PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Doctorado
en Informática

Heterogeneous Verification of Model Transformations

Daniel Calegari García

2014

Heterogeneous Verification of
Model Transformations
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT 14-08
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2014

Universidad de la República
PEDECIBA Informática
Uruguay

Heterogeneous Verification of Model Transformations

Daniel Calegari García

Tesis presentada en cumplimiento parcial de los
requisitos para el grado de **Doctor en Informática**
Universidad de la República - Pedeciba Informática

Montevideo, Uruguay, Abril de 2014

Director de Tesis: Dra. Nora Szasz
Universidad ORT Uruguay

Heterogeneous Verification of
Model Tranformations

Daniel Calegari García

ISSN 0797–6410
Tesis de Doctorado en Informática
Reporte Técnico
PEDECIBA
Universidad de la República
Montevideo, Uruguay, Abril de 2014

tan fácil, fácil no es
horizonte lejano
correr y correr
el dia que no llega
dura es la noche, en soledad
pero el hombre que mira lejos no aprende a ver

Carretera Perdida - Buitres (2001)

Resumen

Esta tesis trata sobre la verificación formal en el contexto de la Ingeniería Dirigida por Modelos (MDE por sus siglas en inglés). El paradigma propone un ciclo de vida de la ingeniería de software basado en una abstracción de su complejidad a través de la definición de modelos y en un proceso de construcción (semi)automático guiado por transformaciones de estos modelos. Nuestro propósito es abordar la verificación de transformaciones de modelos la cual incluye, por extensión, la verificación de sus modelos.

Comenzamos analizando la literatura relacionada con la verificación de transformaciones de modelos para concluir que la heterogeneidad de las propiedades que interesa verificar y de los enfoques para hacerlo, sugiere la necesidad de utilizar diversos dominios lógicos, lo cual es la base de nuestra propuesta. En algunos casos puede ser necesario realizar una verificación heterogénea, es decir, utilizar diferentes formalismos para la verificación de cada una de las partes del problema completo. Además, es beneficioso permitir a los expertos formales elegir el dominio en el que se encuentran más capacitados para llevar a cabo una prueba formal. El principal problema reside en que el mantenimiento de múltiples representaciones formales de los elementos de MDE en diferentes dominios lógicos, puede ser costoso si no existe soporte automático o una relación formal clara entre estas representaciones.

Motivados por esto, definimos un entorno unificado que permite la verificación formal transformaciones de modelos mediante el uso de métodos de verificación heterogéneos, de forma tal que es posible automatizar la traducción formal de los elementos de MDE entre dominios logicos. Nos basamos formalmente en la Teoría de Instituciones, la cual proporciona una base sólida para la representación de los elementos de MDE (a través de *instituciones*) sin depender de ningún dominio lógico específico. También proporciona una forma de especificar traducciones (a través de *comorfismos*) que preservan la semántica entre estos elementos y otros dominios lógicos. Nos basamos en estándares para la especificación de los elementos de MDE. De hecho, definimos una institución para la buena formación de los modelos especificada con una versión simplificada del MetaObject Facility y otra institución para transformaciones utilizando Query/View/Transformation Relations. No obstante, la idea puede ser generalizada a otros enfoques de transformación y lenguajes.

Por último, demostramos la viabilidad del entorno mediante el desarrollo de un prototipo funcional soportado por el Heterogeneous Tool Set (HETS). HETS permite realizar una especificación heterogénea y provee facilidades para el monitoreo de su corrección global. Los elementos de MDE se conectan con otras lógicas ya soportadas en HETS (por ejemplo: lógica de primer orden, lógica modal, entre otras) a través del Common Algebraic Specification Language (CASL). Esta conexión se expresa teóricamente mediante comorfismos desde las instituciones de MDE a la institución subyacente en CASL.

Finalmente, discutimos las principales contribuciones de la tesis. Esto deriva en futuras líneas de investigación que contribuyen a la adopción de métodos formales para la verificación en el contexto de MDE.

Palabras clave: Ingeniería Dirigida por Modelos, verificación, métodos formales, Teoría de Instituciones, MOF, QVT-Relations, Heterogeneous Tool Set

Abstract

This thesis is about formal verification in the context of the Model-Driven Engineering (MDE) paradigm. The paradigm proposes a software engineering life-cycle based on an abstraction from its complexity by defining models, and on a (semi)automatic construction process driven by model transformations. Our purpose is to address the verification of model transformations which includes, by extension, the verification of their models.

We first review the literature on the verification of model transformations to conclude that the heterogeneity we find in the properties of interest to verify, and in the verification approaches, suggests the need of using different logical domains, which is the base of our proposal. In some cases it can be necessary to perform a heterogeneous verification, i.e. using different formalisms for the verification of each part of the whole problem. Moreover, it is useful to allow formal experts to choose the domain in which they are more skilled to address a formal proof. The main problem is that the maintenance of multiple formal representations of the MDE elements in different logical domains, can be expensive if there is no automated assistance or a clear formal relation between these representations.

Motivated by this, we define a unified environment that allows formal verification of model transformations using heterogeneous verification approaches, in such a way that the formal translations of the MDE elements between logical domains can be automated. We formally base the environment on the Theory of Institutions, which provides a sound basis for representing MDE elements (as so called *institutions*) without depending on any specific logical domain. It also provides a way for specifying semantic-preserving translations (as so called *comorphisms*) from these elements to other logical domains. We use standards for the specification of the MDE elements. In fact, we define an institution for the well-formedness of models specified with a simplified version of the MetaObject Facility, and another institution for Query/View/Transformation Relations transformations. However, the idea can be generalized to other transformation approaches and languages.

Finally, we evidence the feasibility of the environment by the development of a functional prototype supported by the Heterogeneous Tool Set (HETS). HETS supports heterogeneous specifications and provides capabilities for monitoring their overall correctness. The MDE elements are connected to the other logics already supported in HETS (e.g. first-order logic, modal logic, among others) through the Common Algebraic Specification Language (CASL). This connection is defined by means of comorphisms from the MDE institutions to the underlying institution of CASL.

We carry out a final discussion of the main contributions of this thesis. This results in future research directions which contribute with the adoption of formal tools for the verification in the context of MDE.

Keywords: Model-Driven Engineering, verification, formal methods, Theory of Institutions, MOF, QVT-Relations, Heterogeneous Tool Set

Acknowledgements

A thesis must be a long and winding road, but fortunately this was far from lonely. Therefore, I want to express my gratitude to several people. First, I want to thank my supervisor Nora Szasz for her guidance and support during the whole development of this thesis. I also want to thank Till Mossakowski and Christian Maeder for their guidance and support during my stay at the University of Bremen. Moreover, I want to thank María Victoria Cengarle and Alexander Knapp for their review of the thesis as well as for the many fruitful discussions we had on this topic, and to Héctor Cancela, Alberto Pardo and Claudia Pons for been part of the evaluation committee.

I also want to thank to all those who have worked with me in the different projects related to this thesis, in particular to Carlos Luna and Alvaro Tasistro which were coauthors of some papers at early stages of this thesis, to those undergraduate students of Computer Engineering which have participated in several projects related to this subject, and to the anonymous referees for their comments and suggestions concerning submitted papers and projects.

Several institutions have contributed with the development of this thesis with financial support to specific projects and grants. I want to thank Agencia Nacional de Investigación e Innovación (ANII), Comisión Académica de Posgrado of Facultad de Ingeniería, and Comisión Sectorial de Investigación Científica of Universidad de la República. I also want to thank Programa de Desarrollo de las Ciencias Básicas (PEDECIBA) in which this PhD was held, and in particular María Inés Sánchez for her help with formal matters.

Last but not least I want to thank my mates of Instituto de Computación, and in particular my "cell mates" Andrea Delgado, Mónica Martínez and Marcos Viera which share everyday suffering with me. Finally, I will not thank my friends or my family, they deserve better than that... all my loving.

Daniel, April 2014

Contents

1 Introduction **1**

2 A Quick Look at MDE **7**
2.1 The MDE Paradigm . . . 7
2.1.1 Models, Metamodels and MOF . . . 8
2.1.2 Model Transformations and QVT-Relations . . . 11
2.2 Class to Relational Example . . . 16

3 Verification of Model Transformations **21**
3.1 What to Verify . . . 21
3.1.1 Language-Related Properties . . . 22
3.1.2 Transformation-Related Properties . . . 22
3.1.3 Summary of Verification Properties . . . 24
3.2 How to Verify . . . 25
3.2.1 Summary of Verification Approaches . . . 26
3.3 Verification by Example . . . 29

4 An Introduction to Institutions **33**
4.1 Institutions . . . 33
4.2 Institution Morphisms . . . 36

5 An Environment for Verification **41**
5.1 Defining the Environment . . . 41
5.2 Representing SW-Models and Metamodels . . . 44
5.3 Representing QVT Transformations . . . 45
5.4 A Proof-Theoretic View . . . 46
5.5 Benefits of the Environment . . . 47
5.6 Related Work . . . 49

6 An Institution for CSMOF **51**
6.1 Preliminaries . . . 51
6.2 Signatures and Formulas . . . 52
6.3 Models . . . 55
6.4 Satisfaction Relation and Satisfaction Condition . . . 57
6.5 Related Work . . . 60
6.6 Model Typing . . . 61

7 An Institution for QVTR **63**
7.1 Preliminaries . . . 63
7.1.1 Recursive Model Transformations . . . 64
7.1.2 Expressions Language . . . 64
7.2 Signatures and Formulas . . . 65
7.3 Models . . . 70
7.4 Satisfaction Relation and Satisfaction Condition . . . 72
7.5 Related Work . . . 77

8 Extending the Institutions **79**
8.1 An Institution for SW-Models . . . 79
8.2 Extending CSMOF and QVTR . . . 83
8.3 Discussion . . . 87

9 Connecting the Institutions with CASL **89**
9.1 Borrowing an Entailment System . . . 89
9.1.1 Common Algebraic Specification Language . . . 90
9.2 Encoding CSMOF into CASL . . . 91
9.3 Encoding QVTR into CASL . . . 97
9.4 Related Work . . . 103

10 Tool Support with HETS **105**
10.1 Heterogeneous Tools Set . . . 105
10.2 Implementation of the Environment . . . 108
10.3 How the Environment Works . . . 112
10.4 Verification Properties . . . 119

11 Conclusions and Further Research **123**
11.1 Summary and Contributions . . . 123
11.2 Discussions and Open Problems . . . 125
11.2.1 On the Formal Definition of the Environment . . . 125
11.2.2 An Institution for OCL . . . 127
11.2.3 Model Transformations & Comorphisms . . . 128
11.2.4 Bridging Technological Spaces . . . 129
11.2.5 Evolution of the Prototype . . . 130

Appendix A Publication List **133**

Appendix B Proofs :: Institution for CSMOF **135**

Appendix C Proofs :: Institution for QVTR **141**

Appendix D Proofs :: CSMOF and QVTR Extensions **149**

Appendix E Proofs :: Comorphisms **151**

Appendix F Code Artifacts for the Running Example **157**

References **179**

1 Introduction

> "Explain to me again what you are doing, I do not understand anything but it sounds great"
>
> My mom

Every traditional software development life-cycle is supported by a number of artifacts (e.g. requirements specifications, analysis and design documents, source code) which are mostly used as guides for the development as well as communication tools with the stakeholders. Human involvement creating, maintaining, interpreting and transforming these artifacts makes this process highly complex and error-prone. In fact, there are studies suggesting that overall software assurance costs account for 30 to 50 percent of total project costs for most software projects [HS02].

These costs may be reduced by adopting a model-centric approach in which different views of the system to be constructed are provided by models. Modeling is the abstraction of the system to be built (or some aspects of it) to deal with its intrinsic complexity in a simplified manner. A (semi)automatic construction process takes place in which models are transformed from higher abstraction levels until an executable system is built. The use of automated mechanisms for the construction of the system improves efficiency on the whole process.

The Model-Driven Engineering (MDE, [Ken02, Sch06]) paradigm is based on these practices. It envisions a software development life-cycle driven by models representing different views of the system to be constructed and model transformations providing a (semi)automatic construction process. It also encompasses other engineering efforts of a complete software engineering process, such maintenance or reverse engineering. Since automation is desirable, models must be formally defined. In this sense, models are defined from metamodels, i.e. a model which introduces the syntax and semantics of certain domain-specific kind of models. The relation between a model and its metamodel is called *conformance* [Béz05]. A model transformation is basically the automatic generation of a target model from a source model,

according to a transformation definition, i.e. a set of rules that describe how certain elements in the source model can be transformed into certain others in the target model [KWB03]. A model transformation can also be considered as a model conforming to a metamodel.

There are many approaches for defining and executing model transformations [CH06, Men10]. In particular, the Object Management Group (OMG) has conducted a standardization process of languages for MDE. They defined the MetaObject Facility (MOF, [OMG03b]) as the language for metamodeling, as well as three transformation languages with different transformation approaches. The Query/View/Transformation Relations (QVT-Relations, [OMG09]) is one of those languages and follows a relational approach which consists of defining transformations as mathematical relations between source and target elements. In this thesis we will rely on these standards for the specification of MDE elements.

The quality of the whole MDE process strongly depends on the quality of the models and model transformations. The reliability of the resulting products is increased by verification of the generated models at early development stages, and of the model transformations to avoid cascading errors. From now on we address the verification of model transformations, which includes by extension the verification of their models and metamodels. In particular, we focus on the use of formal methods (i.e. mathematically based techniques [HJC+08]) for verification of these MDE elements.

The specification and verification of a MDE-built system has some parallelism with traditional software systems. The formal treatment of a problem requires some notation with formal semantics, along with a deductive system for reasoning. This is often considered difficult to apply and requires significant mathematical experience, which leads to a mismatch problem between software engineering expectations and formal methods possibilities. To cope with this situation, a separation of duties between software developers is usually proposed.

On the one side there are those experts in the MDE domain, and on the other, those in formal methods. This gives rise to different technological spaces [KBA02], i.e. working contexts with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. In general terms, MDE experts define models and transformations, while formal experts conduct the verification process, often aided by some (semi)automatic generation process which translates the MDE elements to their formal representation into the domain used for verification purposes. The formal representation is usually defined in a unified semantic domain (e.g. [CLST10b, LR12, BHM09, RDV09]), having tools for conducting the verification process and for retrieving some feedback to the MDE experts. However, the use of a mathematical formalism serving as a unique semantic basis for verification can be quite restrictive.

The minimal requirement to be verified on a model transformation is that the transformation and the source and target models are well-formed, i.e. that they conform to their respective metamodels. However, there are multiple other properties of interest to verify and there is a plethora of verification approaches in the literature [ALS+12, ARW13, CS13d]. In this context, the "heterogeneity problem" arises, which can be defined as the need of using different logical domains for verification. As with traditional software, "heterogeneous multi-logic

specifications are needed, since complex problems have different aspects that are best specified in different logics" [Mos05]. Even if we did not need to use heterogeneous approaches at the same time, it could be useful to allow formal experts to choose the domain in which they feel more confident, or in which they are more skilled to address a formal proof.

In both cases, it is useful to have a representation of MDE elements in different logical domains. However, the maintenance of multiple formal representations of the same elements (one for each domain) can be expensive if there is no automated assistance. One possibility is to generate a partial or complete formal representations of elements from the MDE technological space in different formal domains, and then "connect" these specifications via translations between the logical domains. These connections may be useful to perform a heterogeneous verification (i.e. using different domains for the verification of each part of the whole problem), and also to integrate MDE elements with the specification and verification of other traditional software artifacts.

This introduces an interesting problem to solve: how to create a unified environment in which each technique is used to solve the problems that fits it best, minimizing the burden of switching from one technique to another?

We can follow a heterogeneous specification approach [CKTW08, Mos05] consisting in having different mathematical formalisms for expressing parts of the overall problem and defining semantic-preserving mappings in order to allow "communication" between the formalisms. This approach uses as a basis the Theory of Institutions [GB92] which allows defining *institutions* giving a formal definition of the MDE elements, and formal translations (called *comorphisms*) from these institutions to other logics, also represented as institutions, which will be used for verification. We can also provide an implementation of the environment using the Heterogeneous Tool Set (HETS, [MML07]), which is a tool that supports heterogeneous multi-logic specifications and provides proof management capabilities for monitoring the overall correctness of the process. HETS supports several interconnected logics (e.g. first-order logic, modal logic, rewriting logic, among others) with the Common Algebraic Specification Language (CASL, [MHST03]) as its core language.

Goals and Contributions

The main goal of this thesis is to define a unified environment that allows formal verification of a model transformation using heterogeneous verification approaches. We base the environment on the Theory of Institutions which provides a sound basis to tackle with the "heterogeneity problem" defined before, and to represent MDE elements in some consistent and interdependent way without depending on any specific logical domain. Although we build on the MOF and QVT-Relations standards for the specification of the MDE elements, we follow an idea general enough to be extended to other transformation approaches and languages. We also rely on the existence of HETS to demonstrate the practical benefits of such a formal definition.

We are not concerned with the proposal of a new verification method, with the comparison of verification approaches or with the identification of the "best" verification approach for each kind of problem. We assume that MDE and formal methods practitioners have enough knowledge to select the appropriate strategy for verification. We are providing them with the "glue" they need for connecting their technological spaces and the logical domains needed for specification and verification.

This thesis contributes to the improvement of the quality and reliability of the products developed using the MDE approach by adopting formal tools for the verification of model transformations. It also helps bridging still separate domains such as formal methods and MDE. More specifically, our work makes the following contributions:

- We provide a comprehensive literature review on the verification of model transformations which extends and complements other existing works in the literature [ALS+12, ARW13], and focuses on the use of formal methods for verification issues. We detect the "heterogeneity problem" and conclude about the need of a heterogeneous approach to verification.

- We define a unified environment for the heterogeneous verification of model transformations based on the Theory of Institutions. Compared to other approaches, the environment defines a generic formal representation of the MDE elements in such a way that it is possible to specify semantic-preserving translations from these elements to potentially several logical domains for verification. The environment has many benefits from a software engineering perspective, e.g. it is possible to automate the translations, so the environment is scalable in terms of the transformation specification.

- We develop a solid foundation for the formal representation of MDE elements by defining institutions for both the conformance relation between models and metamodels specified with a simplified version of MOF (CSMOF), and the QVT-Relations transformations (QVTR). These definitions improve existent knowledge on the semantics of MDE and constitute a contribution on the use of institutions for the formalization of specification languages.

- We illustrate how MDE elements can be formally translated into other logics by defining comorphisms from extensions of these institutions to CASL, the core logic of the HETS system, and proving that in such translations the semantics are preserved. The existent connections between CASL and other logics broadens the spectrum of logical domains in which the verification of model transformations can be addressed.

- We evidence the feasibility of the environment by developing a functional prototype supported by HETS. There is no evidence about the using of the Theory of Institutions in practice for the verification of model transformations. In particular, Hets does not support the MDE paradigm, i.e. it does not have specific languages (as CSMOF and QVTR) for the specification of MDE elements. A discussion about the scope of this implementation opens several research directions.

The research work undertaken during the development of this thesis has been presented and published in the following instances. These papers cover most of the work: the comprehensive literature review on the verification of model transformations [CS13d], the definition of the heterogeneous environment [CS13a], and the definition of the institutions [CS13b].

- [CS13d] Daniel Calegari, Nora Szasz: Verification of Model Transformations: A Survey of the State-of-the-Art. Proc. Conf. Latinoamericana de Informática, Medellín, Colombia, 2012. ENTCS 292: 5-25. Elsevier (2013) **Best Paper Award**
- [CS13a] Daniel Calegari, Nora Szasz: Bridging Techological Spaces for the Verification of Model Transformations. Proc. Conf. Iberoamericana de Software Engineering, Montevideo, Uruguay (2013)
- [CS13b] Daniel Calegari, Nora Szasz: Institution-Based Semantics for MOF and QVT-Relations. Proc. 16th Brazilian Symp. Formal Methods, Brasilia, Brasil. LNCS 8195: 34-50. Springer. (2013) **2nd Best Paper Award**

There are other papers not completely related to the content of this thesis but produced under the same context. These papers are summarized in Appendix A.

Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 is dedicated to an overview of MDE. We present the main idea behind this software engineering approach and introduce its main elements (models, metamodels, the conformance relation between them, and model transformations). We also present a simplified version of the well-known Class to Relational transformation [OMG09], which is used throughout this thesis as a proof of concepts. We continue dipping into MDE in Chapter 3, in which we summarize a comprehensive literature review on the verification of model transformations. This review will provide a better understanding on what we call "heterogeneity problem" which motivates this proposal. In Chapter 4 we provide background information on the Theory of Institutions on which our proposal is formally based. In Chapter 5 we introduce the environment presenting how it is defined and we summarize its benefits from a software engineering perspective. In Chapter 6 we provide a formal definition of an institution for the conformance relation (CSMOF), in Chapter 7 we define an institution for model transformations (QVTR), and in Chapter 8 we define an extension of these institutions in order to be used within a proof environment. In Chapter 9 we present the definition of formal translations from these extended institutions into CASL by means of institution comorphisms. The importance of such translations relies in that we can give our environment tool support with HETS. The implementation of a prototype of the environment is detailed in Chapter 10. Finally, Chapter 11 concludes with a summary of our results, a discussion of the main contributions of our work, and an outline of some possible directions of future work.

Some aspects were sent to appendices in order to improve readability of this document. In Appendix A there is the complete publication list produced under the context of this thesis. In Appendix B we give complete formal proofs of properties for the CSMOF institution defined in Chapter 6, and in Appendix C the corresponding proofs for the QVTR institution defined in Chapter 7. In Appendix D we give complete formal proofs of properties for the extended institutions defined in Chapter 8. In Appendix E we give complete proofs for comorphisms properties defined in Chapter 9. Finally, in Appendix F we show the code artifacts and the complete CASL theories generated by the comorphisms for the running example.

How to Read this Thesis

It is possible to read this thesis straight through from start to finish. However, since the target audience is both MDE and formal methods practitioners, we recommend different paths depending on their background and interests. Chapter 2 can be skipped if the reader is familiar with the MDE paradigm, but we recommend to read Chapter 3 if she has no deep understanding on verification issues within such paradigm. Readers with a background in institutions will probably prefer to skip Chapter 4. Chapter 5 should not be skipped since it defines the environment proposed in this work. Chapters 6, 7, 8, and 9 are very technical and not essential for the superficial understanding of the environment, but they provide important foundational issues on its formal definition. Readers without a special interest in formal details can skip them, as well as Chapter 4. Each one of the chapters, from Chapter 5 to Chapter 9, has its own related work section. Chapter 10 can be skipped if there is no interest in technical details on the implementation of the prototype. However, a quick glance at it will provide a better understanding on how the environment works from a practical perspective. Finally, Chapter 11 should not be skipped since it discusses conclusions and future work.

2 A Quick Look at MDE

As models are not intended to represent all aspects of reality, they allow us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality [RWLN89]. The Model-Driven Engineering paradigm is a software engineering paradigm based on the construction of models of a system and their evolution through so called model transformations.

In this chapter we provide an overview of the Model-Driven Engineering paradigm which encloses the general context of our work. In Section 2.1 we describe the MDE paradigm and we introduce its main elements: metamodels, models and model transformations. These concepts are clarified in Section 2.2 with a basic example of a model transformation that will be used throughout the thesis.

2.1 The MDE Paradigm

The Model-Driven Engineering (MDE, [Ken02, Sch06]) refers to the systematic use of models as primary engineering artifacts throughout the engineering process, encompassing all engineering efforts such as development, maintenance or reverse engineering. In particular, the paradigm envisions a development process driven by models (known as Model Driven Development, MDD) representing different views of the system to be constructed. Its feasibility is based on the existence of a (semi)automatic construction process driven by model transformations, starting from abstract models of the system and transforming them until an executable model is generated. This approach tends to improve efficiency on the construction process and the reliability of the resulting products. It also focuses on ensuring software quality attributes by verification of the generated models at early development stages.

Model-Driven Architecture (MDA, [OMG03a]) is an implementation of MDD, defined by the Object Management Group (OMG). MDA introduces different levels of abstraction of the models. These range from models independent from any idea of computation (CIM), e.g. the specification of a business process, to models tightly coupled to a specific implementation language (ISM), e.g. the code itself and configuration scripts. MDA also describes model transformations driving business requirements models to implementations to provide a conceptual framework, as shown in Figure 2.1.

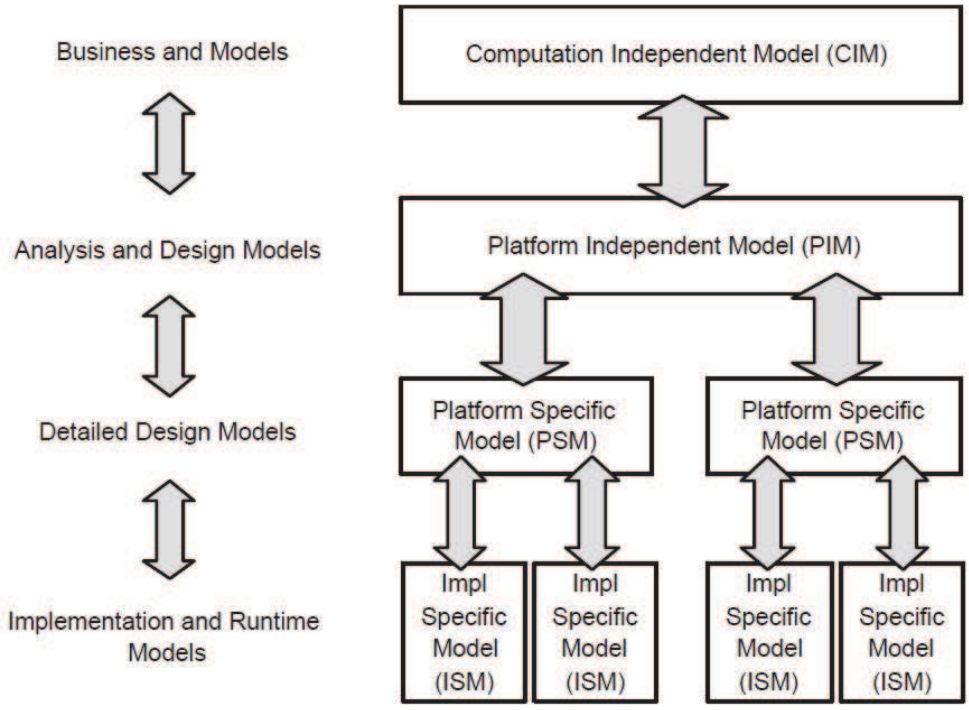


Figure 2.1: Levels and transformations of MDA

MDA involves multiple standards, as the MetaObject Facility (MOF, [OMG03b]) as a language for the definition of models, and the Query/View/Transformation Relations language (QVT-Relations, [OMG09]) for specifying model transformations. These languages will be subject of further explanation in the following sections.

2.1.1 Models, Metamodels and MOF

In the MDE ecosystem everything is a model, even the code is considered as a model. In this context, a model is an abstraction of the system or its environment. Every model *conforms* [Béz05] to a metamodel, i.e. a model which introduces the syntax and semantics of

certain domain-specific kind of models. In the same way, a metamodel conforms to some metametamodel. A metametamodel is usually self-defined, which means that it can be specified by means of its own semantics. These elements and relations introduce a set of layers [Béz05] depicted in Figure 2.2. At the bottom level, the M0 layer is the real system. In the other layers M1 to M3 are the three different kinds of models:

- Terminal models (level M1): conform to metamodels and are representations of real-world systems.
- Metamodels (level M2): conform to metametamodels and define domain-specific concepts.
- Metametamodels (level M3): conform to themselves and provide generic concepts for metamodel specification.

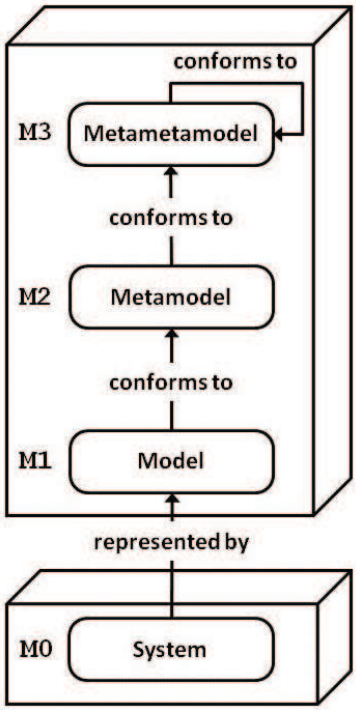


Figure 2.2: The 3+1 architecture of MDE

Metamodels are usually graphically depicted using UML Class Diagrams [OMG05]. However, there are several other specific languages for this purpose, e.g. the MOF, the Ecore metametamodel defined for the Eclipse Modeling Framework [FSM+03], and KM3 [ATL05] defined for the ATL [JK05] transformation language.

The MOF is a standard language for metamodeling. To "simplify the MOF so as to facilitate ease of implementation and conformance while maximizing interoperability and convenient interchange" [OMG03b], the standard defines Essential MOF (EMOF) which is a subset of MOF allowing simple metamodels to be defined. In Figure 2.3 there is a simplified version of the EMOF metametamodel, without operations on classes or packages.

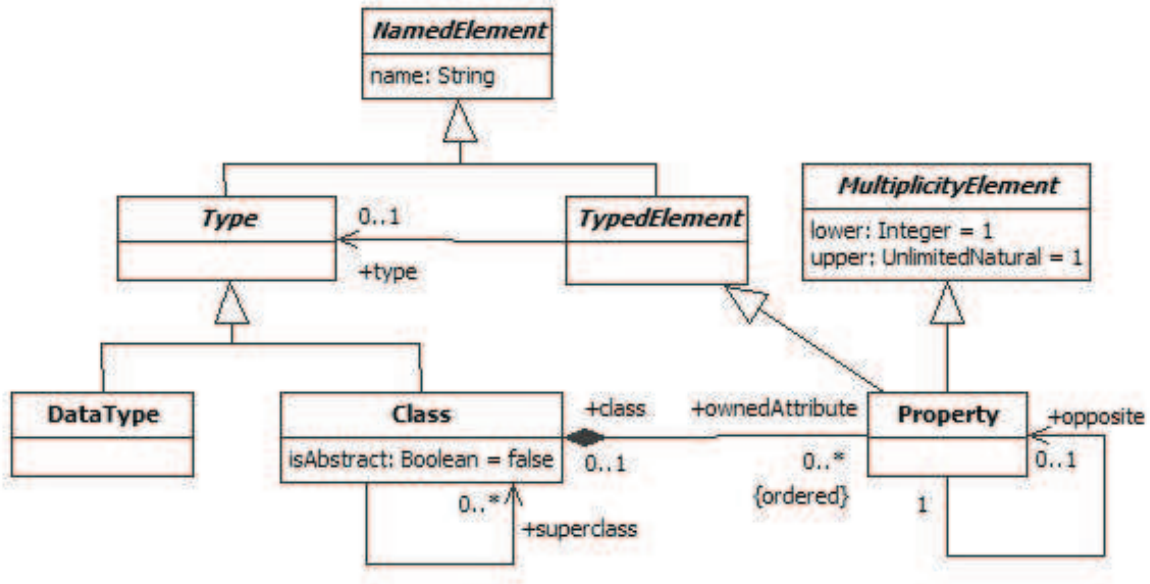


Figure 2.3: A simplified EMOF metametamodel

In few words, an EMOF metamodel defines classes which can belong to a hierarchical structure. Some of them may be defined as abstract (there are no instances of them). Any class has properties which can be attributes (named elements with an associated type which can be a primitive type or another class) and associations (relations between classes in which each class plays a role within the relation). Every property has a multiplicity which constraints the number of elements that can be related through the property, and it can be related with another property (known as its opposite) if the property corresponds to a bidirectional association between two classes. In some cases, there are conditions (called invariants) that cannot be captured by the structural rules of this language, in which case the language is supplemented with another one, e.g. the Object Constraint Language (OCL, [OMG10]). OCL can be used with any MOF metamodel, e.g. within any UML language.

These considerations allow defining the conformance relation in terms of *structural* and *non-structural* conformance. Structural conformance with respect to a MOF metamodel means that in a given SW-model: every object has a type defined within the metamodel, every property link has a corresponding property, such that objects connected with such property link are

well-typed according to the corresponding property, and the SW-model also respects the multiplicity constraints. Non-structural conformance means that a given SW-model respects the invariants specified with the supplementary constraints language. Sometimes some structural constraints, e.g. multiplicity constraints, can also be represented using the supplementary constraint language.

Although a metamodel usually defines a modeling language which has a concrete syntax, it is possible to represent a model using the same languages as for metamodels. Moreover, for representing models and instances of models, there is the graphical representation provided by UML Object Diagrams [OMG05]. Every element in this schema can be represented using XML Metadata Interchange (XMI, [OMG11]), a XML-based language for exchanging metadata information whose metamodel can be expressed in MOF.

2.1.2 Model Transformations and QVT-Relations

The second building block of the MDE paradigm is the model transformations, which can also be considered as models. As pointed out in [CH06], model transformations are closely related to program transformations. "Their differences occur in the mindsets and traditions of their respective transformation communities, the subjects being transformed, and the sets of requirements being considered. While program transformation systems are typically based on mathematically oriented concepts such as term rewriting, attribute grammars, and functional programming, model transformation systems usually adopt an object-oriented approach for representing and manipulating their subject models."

As summarized in Figure 2.4, a model transformation (or just transformation from now on) basically takes as input a source model conforming to a given source metamodel and produces as output another model conforming to a given target metamodel. The model transformation can be defined as well as a model which itself conforms to a model transformation metamodel. This last metamodel, along with the source and target metamodels, must conform to a metametamodel (such as MOF). The transformation definition is executed by a transformation engine.

This schema defines model-to-model transformations. There are also model-to-text and text-to-model transformations where the target and source models, respectively, are just strings not conforming to any specific metamodel. Without loss of generality we will only consider model-to-model transformations.

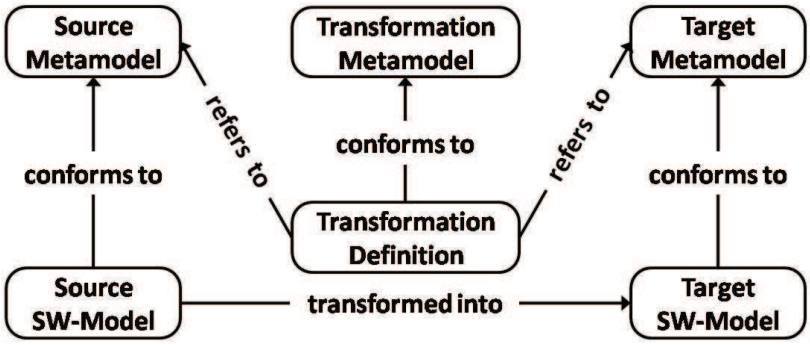


Figure 2.4: General schema of a model transformation

As pointed out in [MCG04, Men10], "this definition is very general, and covers a wide range of activities for which model transformation can be used: automatic code generation, model synthesis, model evolution, model simulation, model execution, model quality improvement (e.g. through model refactoring), model translation, model-based testing, model checking, model verification." However, this schema can be extended as is exhaustively studied in [CH06]. Leaving aside the details, the authors identify multiple variabilities on a model transformation, e.g. it can be bidirectional, it can take more than one source model as input and/or produce multiple target models as output, its rule application strategy can be deterministic, non-deterministic or interactive, the source and target models can be at different abstraction levels (horizontal versus vertical transformations) or not, and the source and target models may conform to the same metamodel (endogenous versus exogenous transformations) or not.

There are several approaches for model transformations [CH06, Men10], among others:

Operational (Imperative). It is similar to direct manipulation but offers more dedicated support. A typical solution is to extend the used metamodeling formalism with facilities for expressing computations. Examples of systems in this category are QVT-Operational mappings [OMG09], and Kermeta [MFJ05].

Relational (Declarative). It consists on defining transformation rules as mathematical relations between source and target elements. Its execution can be seen as a form of constraint solving. Examples of this approach are QVT-Relations [OMG09], and Tefkat [LS05].

Graph-transformation-based. It consists on considering models as typed, attributed, labeled graphs and then applying graph transformations. Examples of this include the following tools: AGG [Tae03], and VIATRA [CHM+02].

The OMG defines three languages with different transformation approaches. In particular, the Query/View/Transformation Relations (QVT-Relations, [OMG09]) is a relational language which defines transformation rules as mathematical relations between source and target elements. In Figure 2.5 there is an excerpt of the QVT-Relations metametamodel.

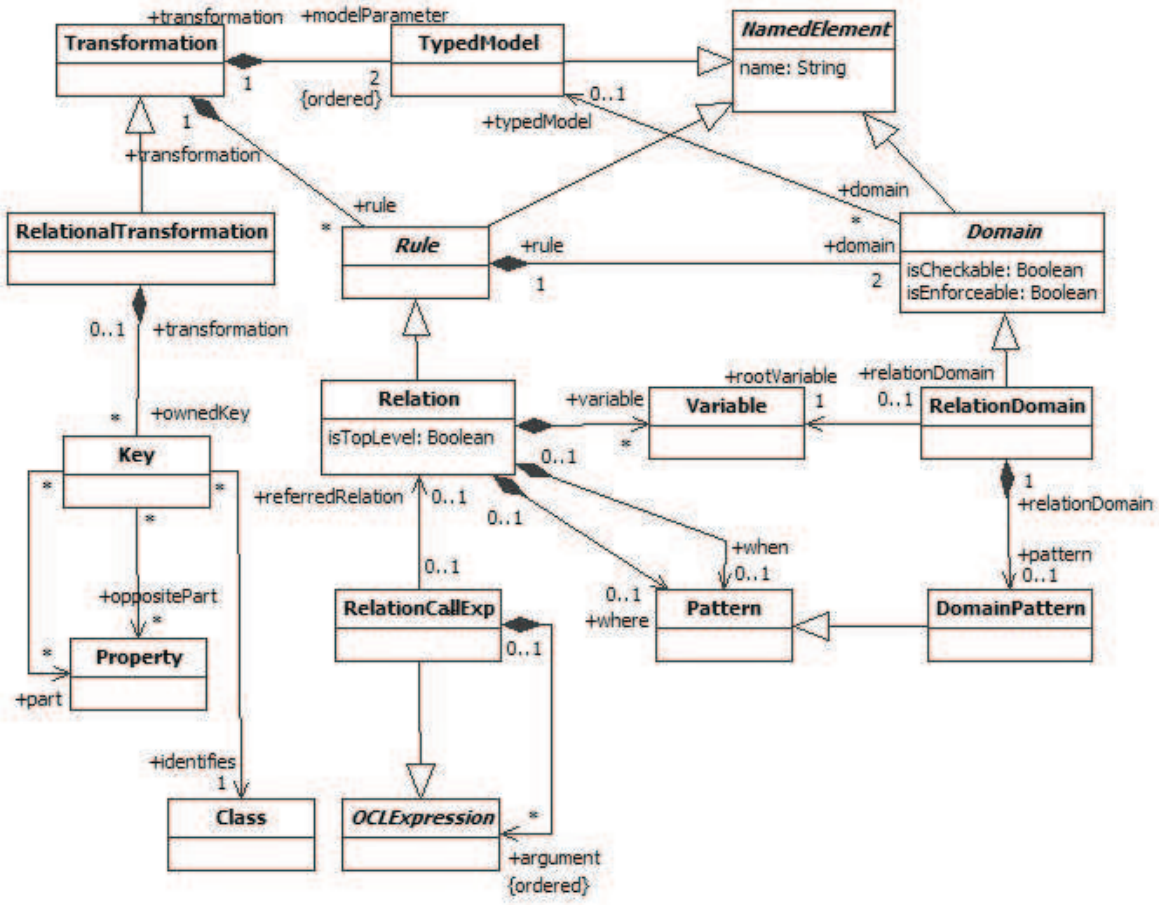


Figure 2.5: An excerpt of the QVT-Relations metametamodel

A relational transformation can be viewed as a set of interconnected relations (we will refer to them as rules indistinctly) together with keys on metamodel elements of the form `key Class {prop1,..., propN}`. Keys are a definition of which properties of a MOF class, in combination, can uniquely identify an instance of that class. They are used in practice at the time of object creation. Notice that they are non-structural constraints on the source and target metamodels, thus they can also be represented using the supplementary constraint language. The identifying properties can also refer to non-navigable opposite roles, e.g. `key Class {prop1, opposite(Class2.property)}`.

Relations are of two kinds: top-level relations which must hold in any transformation execution, and non-top-level relations which are required to hold only when they are referred from another relation. We can view a relation as having the following abstract structure [OMG09] (we consider only a source and a target metamodel).

```
[top] relation R {
    <R_var_set> <R_par_set>
    Domain { <domain_k_var_set> <domain_k_pat> } //k = 1,2
    [when <when_var_set> <when_cond>]
    [where <where_cond>]
}
```

Every relation has a set `<R_var_set>` of variables occurring in the relation, which are particularly used within the domains (`<domain_k_var_set>`) and in the `when` clause (`<when_var_set>`). Relations define source/target domain patterns `<domain_k_pat>` which are used to find matching sub-graphs in a model. A pattern can be viewed as a graph of typed elements (which will be matched by objects) and relations (which will be matched by links), together with a predicate (boolean expression) which must hold. The predicate may refer to variables other than the pattern elements; these are the free variables of a pattern. A pattern can be represented as:

```
e1: <classname1>, e2: <classname2> ... en:<classnameN>
l1 : <assoc1> (ei, ej) ... lm:<assocM>(eu, ew)
where <predicate>
```

Relations can also contain `when` (`<when_cond>`) and `where` (`<where_cond>`) clauses. A `when` clause specifies the conditions under which the relationship holds, whilst the `where` clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. The `when` and `where` clauses, as well as the predicate of a pattern, may contain arbitrary boolean OCL expressions in addition to the relation invocation expressions.

Finally, any relation can define a set of primitive domains which are data types used to parameterize the relation (`<R_par_set>`). In this sense, top-level relations can be parametric when called from a `when` clause, whereas non-top-level relations are always parametric since they are called for given source and target domains elements.

The execution of a transformation requires that all its top-level relations hold, as well as the keys (if any). However the effect of a transformation depends on the direction of its execution, and on the flags that can be attached to domains. Domains can be defined as **checkonly** or **enforced**. As the standard says "When a transformation is enforced in the direction of a checkonly domain, it is simply checked to see if there exists a valid match in the relevant model that satisfies the relationship. When a transformation executes in the direction of the model of an enforced domain, if checking fails, the target model is modified so as to satisfy the relationship." [OMG09].

These two execution modes are determined by two different semantics of a rule: the checking and the enforcement semantics. The first allows determining *what* relations must exists between source and target models (useful for verification), whereas the second determines *how* a model transformation needs to execute. Moreover, as we consider only two metamodels, we also consider that the transformation is executed in the direction of the second domain. In this context, we customize the standard checking semantics and take the first and second patterns as the source and target patterns, respectively. Using `|<var_set>|` as a binding of variables of the set `<var_set>`, and `<exc_domain_k_var_set>` as the variables of domain k that do neither occur in the other domain nor the `when` clause, the standard states that a rule holds if:

$$
\begin{aligned}
&\forall\, |< \text{when\_var\_set} >|, \\
&\quad (\, < \text{when\_cond} > \rightarrow \\
&\qquad \forall\, |< \text{R\_var\_set} > \backslash (< \text{when\_var\_set} > \cup < \text{exc\_domain\_2\_var\_set} >)|, \\
&\qquad\quad (\, < \text{domain\_1\_pat} > \rightarrow \\
&\qquad\qquad \exists\, |< \text{exc\_domain\_2\_var\_set} >|, \\
&\qquad\qquad\quad (\, < \text{domain\_2\_pat} > \, \wedge < \text{where\_cond} >)))
\end{aligned}
$$

This formula states that a rule holds if for each valid binding of variables of the `when` clause and variables of domains other than the target domain, that satisfy the `when` condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of the target domain that satisfies the target domain pattern and `where` condition.

2.2 Class to Relational Example

Let us consider the following example which is a simplified version of the well-known Class to Relational transformation [OMG09] that will be used throughout this thesis. The metamodel on top of Figure 2.6 defines UML class diagrams, where classifiers (classes and primitive types as string, boolean, integer, etc.) are contained in packages. Classes can contain attributes and may be declared as persistent, whilst attributes have a type that is a primitive type.

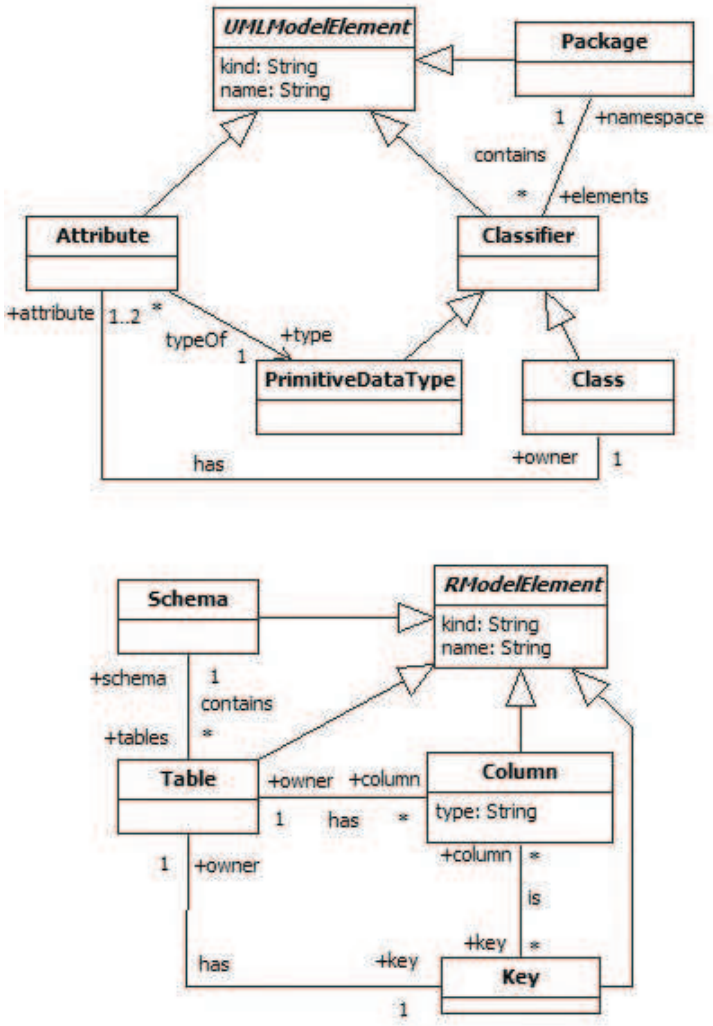


Figure 2.6: Metamodels for the Class to Relational transformation

Notice that a class must contain only one or two attributes, and also that the Classifier class is not abstract. We decided to handle these aspects differently from UML class diagrams in order to have a more complete example. On the other side, relational models conform to the second metamodel in Figure 2.6. Every schema contains a number of tables and each table has a number of columns. Each column has a name and a kind, which can be the primary keys of the corresponding table.

The transformation basically describes how persistent classes within a package are transformed into tables within a schema.

```
transformation uml2rdbms  ( uml : UML , rdbms : RDBMS ) {

  key RDBMS::Table {name, schema};
  key RDBMS::Column {name, owner};
  key RDBMS::Key {name, owner};

  top relation PackageToSchema {
    pn : String;

    checkonly domain uml p : UML::Package {
      name = pn
    };
    enforce domain rdbms s : RDBMS::Schema {
      name = pn
    };
  }

  top relation ClassToTable {
    cn, prefix : String;

    checkonly domain uml c : UML::Class {
      namespace = p : UML::Package {},
      kind = 'Persistent',
      name = cn
    };
    enforce domain rdbms t : RDBMS::Table {
      schema = s : RDBMS::Schema {},
      name = cn,
      column = cl : RDBMS::Column {
        name = 'TID',
        typeT = 'NUMBER'
      },
      keyK = k : RDBMS::Key {
        name = 'PK',
        column = cl
      }
    };
```

```
    when {
      PackageToSchema(p, s);
    }

    where {
      AttributeToColumn(c, t, prefix);
      prefix = '';
    }
  }

  relation AttributeToColumn {
    an, pn, cn, sqltype : String;

    primitive domain prefix : String;

    checkonly domain uml c : UML::Class {
      attribute = a : UML::Attribute {
        name = an,
        typeT = p : UML::PrimitiveDataType {
          name = pn
        }
      }
    };

    enforce domain rdbms t : RDBMS::Table {
      column = cl : RDBMS::Column {
        name = cn,
        typeT = sqltype
      }
    };


    where {
      cn = if prefix = '' then an else prefix + an endif;
      sqltype = if pn = 'INTEGER'
                then 'NUMBER'
                else if pn = 'BOOLEAN'
                     then 'BOOLEAN'
                     else 'VARCHAR' endif endif;
    }
  }
}
```

The rule `PackageToSchema` states that UML packages are mapped into RDBMS schemas. The rule `ClassToTable` states that classes marked as persistent are mapped into tables with the same name, a primary key and an identifying column, such that the package to

which the class belongs is in the relation with the schema to which the table belongs. The rule `AttributeToColumn` is called from the where clause of `ClassToTable`. This rule states that primitive attributes of the persistent class map to columns of the table, such that the name of a column is determined by a formula according to the value of the primitive domain `prefix`, and the type of that column is determined by another formula depending on the name of the primitive attribute.

In Figure 2.7 there is an example of a source model and its corresponding target model. The source model is composed by a persistent class of name `ID` within a package of name `Package`. The class has an attribute of name `value` and type `String` which is a primitive type. The forward execution of the transformation gives the target model which contains a schema of name `Package` with a table of name `ID`, which corresponds to the package and class in the source model. The table has two columns, one of name `value` and type `VARCHAR` corresponding to the string attribute in the source class, and another which is a default primary key without any correspondence in the source model.

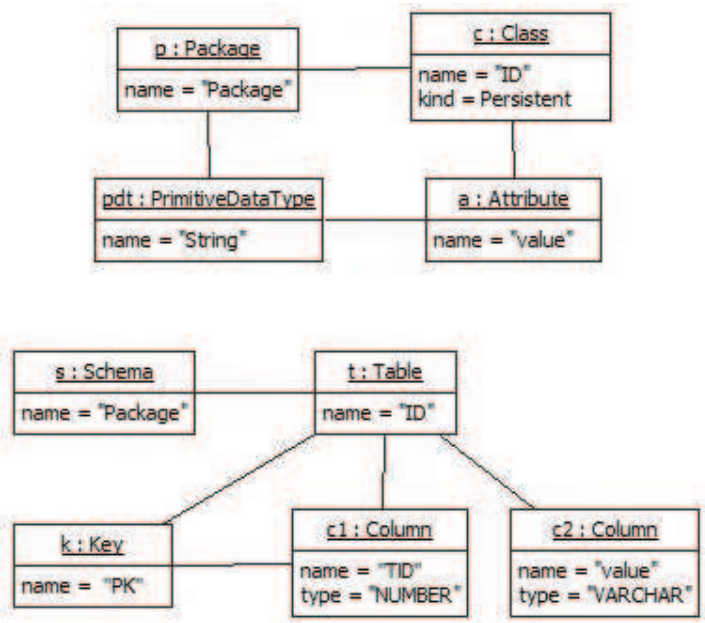


Figure 2.7: Models for the Class to Relational transformation

3 Verification of Model Transformations

The verification of a model transformation –which includes by extension the verification of its models and metamodels– is defined in [ALS+12] as a tri-dimensional problem consisting of: the transformation involved, the properties of interest addressed, and the formal verification techniques used to establish the properties. We developed a comprehensive literature review [CS13d] (extended in [CS12]) on the verification of model transformations extending [ALS+12] and focusing on the last two dimensions.

In this chapter we summarize the main aspects of the verification of model transformations contained in our review. In Section 3.1 we introduce the different aspects of a transformation that must be verified, and in Section 3.2 we review how these aspects are verified in the literature. Finally, in Section 3.3 we define some cases to exemplify verification properties and discuss how they can be verified.

3.1 What to Verify

As we have seen in Chapter 2, there are many transformation approaches. However, these approaches do not define what things have to be verified but how in some cases the verification must be done. Consequently, transformation properties can be explored without talking about any transformation approach in particular. Verification properties can be classified, according to [ALS+12], in language-related and transformation-related properties, the first ones referring to the computational nature of transformations and target properties of transformation languages, and the second ones referring to the modeling nature of transformations. When following a MDE-based approach, formal verification is mostly focused on the second category of properties (transformation-related), whilst those within the first category (language-related) are in general assumed to be somehow automatically verified by the development tools.

3.1.1 Language-Related Properties

This category refers to the computational nature of transformations and target properties of transformation languages. As introduced in [ALS+12], a transformation specification conforms to a transformation language which can possess properties on its own.

The first two properties are identified as execution-time properties. They are the **Termination** property which guarantees the existence of a target model, i.e. that the transformation execution finishes for any well-formed transformation specification, and the **Determinism** (a.k.a. Confluence) property which ensures uniqueness of the target model for a given source model and transformation specification. These properties are related to undecidable problems for sufficiently expressive (i.e. Turing-complete) transformation languages. In these cases, either the transformation language is kept as general as possible, making these properties undecidable, but the transformation framework provides capabilities for checking sufficient conditions ensuring them to hold on a particular transformation; or these properties are ensured by construction by the language, generally by sacrificing its expressive power [ALS+12]. According to this, in the first case, the properties may also be identified as transformation-related properties (as in the next section) when proved for a specific transformation specification.

The third property, identified as a design-time property, is **Typing**, i.e. ensuring the well-formedness of the transformation specification with respect to its transformation language. The process of type checking may occur either at compile or run-time. Since model transformations are models, and models have metamodels (defining the transformation language), solutions to this problem are strongly related to Conformance and Model Typing as will be introduced in the next section.

Finally, there is a **Preservation of Execution Semantics** property. This execution-time property states that the transformation execution must behave as expected according to the definition of the transformation language semantics. Related to this, and in strong contact with the Typing property, there are consistency needs between transformation rules which must also hold. For example, some languages do not allow an element of the input model to be matched more than once (redundancy problem). If this property does not hold, contradictory rules may be applied, e.g. two rules applied to the same element implying different things. Moreover, it is possible that a rule applied to an element of a hierarchy may be more restrictive than other one applied to an element in a lower level of the same hierarchy. In this case, there will be some models not matched by the second rule.

3.1.2 Transformation-Related Properties

This category refers to the modeling nature of transformations. As introduced in [ALS+12], a transformation refers to source/target models for which dedicated properties need to be ensured for the transformation to behave correctly.

There is a first subcategory of properties known as **On the Source/Target Model(s)** that concerns the source and/or target model(s) a transformation refers to. As pointed out in [VP03], the minimal requirement is to guarantee that the generated model is a syntactically well-formed instance of the target language. This introduces a first group of properties known as **Conformance and Model Typing**. Conformance is nowadays well understood and automatically checked within modeling frameworks. In this context, type systems derive from those for object-based languages which are reasonably well-understood [SJ07]. There are also typing requirements with respect to the artifacts handled at an architecture level, i.e. transformation chains, repositories and other model-related services [VJBB13]. There is a second group known as **N-Ary Transformations Properties**: transformations operating on several models at the same time, e.g. model composition, merging, or weaving, require dedicated properties to be checked. But verification interests go beyond this kind of problems. When verifying a model transformation we want to consider its elements as a whole and not individually. In this sense, some authors, as in [CCGdL10], use the notion of a *transformation model* [BBG+06b], i.e. a model composed by the source and target metamodel, the transformation specification and the well-formedness rules. This transformation model can be implicit (i.e. we assume that every element is connected), or explicit (i.e. we really construct a unified structure with different purposes, e.g. tracing or verification).

There are properties known as **Model Syntax Relations** that relate metamodel elements of the source and the target metamodels trying to ensure that certain elements or structures of any input model will be transformed into other elements or structures of the output model. This problem arises when these relations cannot be inferred by just looking at the individual transformation rules, or when the transformation language does not allow expressing some relations, and another constraint language must be used. This is also known as preservation of transformation invariants or structural correspondence. Beyond structural relationships between source and target models, there are semantic properties that must be preserved, known as **Model Semantics Relations** and also as *semantic correctness* or *dynamic consistency* [VP03]. These properties generally depend on the metamodels semantics or on the kind of transformation. Some properties of interest are semantic equivalence, (weak) bi-similarity and preservation of properties, temporal properties, refactoring, and refinement.

Finally, there is a fourth category called **Functional Behavior** which refers to identifying if a transformation behaves as a mathematical function [CCGdL10]. In particular, a transformation may be injective, surjective, bijective, or at least, executable (i.e. there exists a valid pair of source and target models that satisfy the transformation). It is also possible to analyze these properties considering individual rules within a transformation. Moreover, there is a specific property known as **Syntactic Completeness** which refers to completely covering the metamodels by transformation rules. This is also presented in [KAER06] as *metamodel coverage* introducing the problem that if the transformation does not cover the entire metamodel, then this leads to some input models which cannot be transformed. From a functional point of view, syntactic completeness means that the transformation is a total function. When considered for a specific transformation, determinism is also a functional property. In fact, when a transformation is *total* and *deterministic*, it is called *functional* [CCGdL10].

3.1.3 Summary of Verification Properties

We have seen a classification of the properties of interest addressed by the verification of a model transformation. This classification identifies language-related and transformation-related properties, the first ones referring to the computational nature of transformations and target properties of transformation languages, and the second ones referring to the modeling nature of transformations. When following a MDE-based software project, formal verification is mostly focused on the second category of properties (transformation-related), whilst those within the first category (language-related) are in general assumed to be somehow automatically verified by the development tools. A summary of the properties addressed in the literature can be found in Table 3.1.

Table 3.1: Summary of properties addressed in the literature

Language-Related Properties		
Termination	[BLA+10][Bru08][EEdL+05][Küs06] [LR10][VVGE+06][WKK+09]	
Determinism	[BLA+10][CCGdL10][HEOG10][HKT02] [Küs06][LEO06][LR10][WKK+09]	
Typing	[KMS+09][LLM09],[SJ07]	
Preservation of Exec. Sem.	[ABGR10][CCGdL10][GdLW+13] [LD10][PCG11][WKK+09]	
Transformation-Related Properties		
Source/Target	Conformance	[ABGR10][AKP03][ALL10] [CBBD09][CLST10b][GM07][Küs04] [LD10][LMAL10][LR10][LR11][Poe08] [Sch10][SMR11][WKK+09][VJBB13]
	N-Ary	[CNS12][Kat06][MSJ+10]
Syntax Relations	[AKP03][ALL10][CBBD09][CLST10b][GM07] [CHM+02][GdLW+13][LBA10][LD10][LMAL10][LR10] [NK08a][OW09][Poe08][Sch10][SK08]	
Semantics Relations	General	[CLST10b][LR11][Poe08][Sch10][SMR11]
	Sem. Eq.	[BEH06][BKMW08][CCGT09][GGL+06] [HHK10][LR10][NK08b] [PGE97][RLK+08][VP03]
	Temporal	[BBG+06a][BHM09][CHM+02][EKHL03]
	Refactoring	[HT05]
	Refinement	[CBBD09][MGB05][PG08]
Functional Behavior	General	[CCGdL10]
	Synt. Comp.	[CHM+02][GdLW+13][KAER06] [LR10][PCG11][WKC06]

3.2 How to Verify

As pointed out in [EKHL03], a property can be either verified or validated, leading to the well-known distinction between verification and validation. Formally, verification is addressed to "determine whether the products [...] satisfy the conditions imposed" whilst validation is addressed to "determine whether it [the product] satisfies specified requirements" [IEE90]. In other words, verification is the process of proving that we are building the product in the right way, while validation is the process of proving that we are building the right product.

We are focused on verification, and in particular on formal verification, i.e. in the act of verifying using formal methods. Although formal verification techniques may be expensive, they can be helpful in guaranteeing the correctness of critical applications where no other verification technique is acceptable. In contrast to formal verification, there are other techniques which may detect errors or improve confidence, but they cannot prove any property in a definite way.

Verification techniques can be classified in different categories referring to: (a) the kind of technique used for verification, (b) the abstraction level with respect to the elements involved in the transformation, (c) the abstraction level with respect to the implementation of the transformation, and (d) the dependency/independency with respect to the transformation specification. It is worth saying that these categories are orthogonal, i.e. there are verification techniques which correspond to more than one category.

Inference, model checking, testing, static analysis or by construction This category refers to the kind of technique used for verification. *Logical inference* (a.k.a. *theorem proving*) consists of using a mathematical representation of a system and the properties that must be verified, as well as a logic in that semantic domain which allows reasoning about that representation, leading from premises to conclusions. This process is usually carried out using theorem proving software and it is usually only partially automated. *Model checking* also consists of using a mathematical representation of a system and proofs consist of a systematic exhaustive exploration of the mathematical model. With the first approach there is usually a high verification cost, whilst with the second one there are well-known limitations such as the state-explosion problem. On the other hand, *testing* relies on the construction of test strategies for a property including subsequent execution of (either parts or all of) the system according to these strategies. As it is popularly said, testing can only show the presence of errors and not their absence. Finally, we can find strategies based on *static analysis*, i.e. on the analysis of a model transformation that is performed without actually executing it. Static analysis typically consists on semi-decision techniques. In this sense, they are efficient but they cannot assure the overall correctness of the design. For the sake of completeness we also consider the satisfaction of properties that hold *by construction* of the transformation, e.g. those achieved by using special transformation languages as DSLTrans [BLA+10].

Metamodel or model level This category consists of the abstraction level with respect to the elements involved in the transformation, and it is also referred as "offline and online" verification [ALL10], and as "input independent and input dependent" verification [ALS+12]. *Metamodel-level* verification uses the metamodel information for verifying properties for any well-formed model instance while *model-level* verification uses arbitrary source models. As pointed out in [VP03], the first level typically requires the use of sophisticated theorem proving techniques and tools with a huge verification cost. For this reason, the second is in many cases a practical and valuable aid, but it cannot ensure the zero-fault level of quality since it checks a finite number of specific cases. However, as model-level verification takes place on a lower level of abstraction, the range of properties that can be validated is much greater than when using metamodel-level verification.

Specification or implementation As introduced in [EKHL03], verification can either be done on the model (specification) level or on the implementation level. *Specification-level* verification involves only the specification of the transformation in some transformation language, and in consequence the semantics defined for that transformation language. In contrast, *implementation-level* verification means also considering the way a transformation is executed by a transformation engine. As far as we know, verification techniques found in the literature are of the first type, since it is assumed that any transformation engine conforms with the semantics of the transformation language and properties do not depend on how exactly the transformation is executed (including the case of Determinism, Termination and Preservation of Execution Semantics properties).

Transformation independent or dependent This is introduced in [ALS+12]. *Transformation independent* techniques are those techniques which prove properties for any transformation, and in consequence they assure that no assumption is made on the specific source model. In contrast, *transformation dependent* techniques rely on a specific model transformation. Transformation independency is achieved either by a transformation language that preserves the properties by default, or by ensuring a property by construction of the transformation.

3.2.1 Summary of Verification Approaches

We have shown how verification techniques can be classified in different categories referring to: (a) the kind of technique used for verification, (b) the abstraction level with respect to the elements involved in the transformation, (c) the abstraction level with respect to the implementation of the transformation, and (d) the dependency/independency with respect to the transformation specification. It is worth saying that these categories are orthogonal, i.e. there are verification techniques which correspond to more than one category. A summary of the verification techniques addressed in the literature within this categorization can be found in Table 3.2 and Table 3.3.

Table 3.2: Summary of verification techniques addressed in the literature

	inf	mod chk	sta ana	tes	con	met	mod	spe	imp	tra ind	tra dep
[ABGR10]	√						√	√			√
[BBG+06a]	√						√	√			√
[ALL10]	√					√		√			√
[LMAL10]	√					√		√			√
[GGL+06]	√					√		√			√
[Bru08]	√					√		√		√	
[Cha06]	√					√		√		√	
[CLST10b]	√					√		√		√	
[LR10]	√					√		√		√	
[LD10]	√					√		√		√	
[Poe08]	√					√		√		√	
[Sch10]	√					√		√		√	
[SMR11]	√					√		√		√	
[AKP03]	√						√	√		√	
[CNS12]	√						√	√		√	
[PG08]	√	√		√			√	√			√
[CCGdL10]		√				√		√		√	
[LEO06]		√				√		√		√	
[EEdL+05]		√				√		√		√	
[MSJ+10]		√				√		√		√	
[HEOG10]		√				√		√		√	
[HKT02]		√				√		√		√	
[OW09]		√				√		√		√	
[VVGE+06]		√				√		√		√	
[WKK+09]		√				√		√		√	
[LBA10]		√				√		√			√
[VP03]		√				√		√			√

Table 3.3: Summary of verification techniques... [Continuation]

	inf	mod chk	sta ana	tes	con	met	mod	spe	imp	tra ind	tra dep
[BEH06]		√					√	√		√	
[BHM09]		√					√	√		√	
[CBBD09]		√					√	√		√	
[CHM+02]		√					√	√		√	
[EKHL03]		√					√	√		√	
[GdLW+13]		√					√	√		√	
[GM07]		√					√	√		√	
[HHK10]		√						√		√	
[HT05]		√				√	√	√		√	√
[MGB05]		√				√	√	√		√	√
[RLK+08]		√				√	√	√		√	√
[Küs04]		√	√			√		√		√	
[Küs06]		√	√			√		√		√	
[NK08b]		√				√	√	√			√
[NK08a]		√				√	√	√			√
[Kat06]			√					√		√	
[VR11]			√					√		√	
[PCG11]			√			√		√		√	
[WKC06]				√			√	√		√	
[KAER06]				√			√	√		√	
[BKMW08]					√	√		√		√	
[BLA+10]					√	√		√		√	
[KMS+09]					√	√		√		√	
[LLM09]					√	√		√		√	
[PGE97]					√	√		√		√	
[SJ07]					√	√		√		√	
[CCGT09]					√	√		√			√

3.3 Verification by Example

In this section we illustrate several verification properties and discuss how the verification can be addressed using the example introduced in Section 2.2.

Beyond the basic conformance needs, there are usually invariants that cannot be captured by the structural rules of the modeling language. Invariants are well-formedness rules that must hold at all time for any model conforming to a metamodel. In the example the following invariants must hold:

- The owned attributes of a class are uniquely named within their owner class
- All tables have distinct names
- All columns have distinct names within a table
- Every table must have at least one primary column

Invariants can be expressed using a constraint language like the OCL, and as it was said in Section 3.1, this conformance checking is nowadays automatically addressed within modeling frameworks using automated checkers. These checkers can be based on SAT solvers or model-checking, as in [ABGR10, GM07]. This verification is at a model level. But there are other alternatives, for example performing the verification using logical inference. In this case, we can formalize metamodels, models and invariants in some formal language, like B or directly FOL, and then use a proof assistant, like Coq and IsabelleHOL, as done in [LD10, LR10, Sch10, SMR11].

Moreover, using this strategy, one can perform the verification at a metamodel level, forgetting models and considering transformations. Then, postconditions are proved assuming that both the pre-conditions and the transformation rules hold, as done in [CLST10b]. For our running example, a simple property that can be proven is that the length of the `Columns` within a `Table` must be greater than zero. This property holds by the fact that every `Attribute` is transformed into a `Column` and because every `Class` has at least one `Attribute`. This information is given in the transformation rules and in the source invariants, respectively.

A step further of this approach is the one presented in [Poe08] where the author proposes a representation of models, metamodels and transformations in the Calculus of Inductive Constructions [PPM89], and then a correct transformation can be extracted. This requires specifying a transformation as types of the form

$$\forall x \ : \ Pil.I(x) \ \rightarrow \ (\exists y \ : \ Psl.O(x, y))$$

where Pil and Psl are source and target metamodel types, $I(x)$ specifies a precondition on the source model x for the transformation to be applied, and $O(x, y)$ specifies the required properties of the output model y. A proof of this expression implies the automatic

construction of a function f such that, given any x satisfying the precondition $I(x)$, then the postcondition $O(x, fx)$ will be satisfied.

Finally, a complementary approach in [LMAL10] proposes a language for assertions based on FOL that describes some characteristics of a model under transformation. Then, they can derive how an assertion evolves when applying transformation rules using SWIProlog [WSTL10] as an inference system. If the assertions to be verified can be derived from the final assertion, thus they hold in the target model.

Model Syntax Relations

A transformation model gives useful information about the relation between the elements connected by a transformation. With this, some other relations can be inferred which are not evident by just looking at the individual elements.

If we consider the complete example in the QVT standard [OMG09] in which there is a hierarchy between classes and attributes can be primitive or other classes, we can state that if `c` is a subclass of `d`, then all columns of `c`'s table are included in `d`'s table. Since we need three different rules for transforming attributes to columns, this property cannot be trivially inferred.

There are many alternatives for proving this property. First, we can use a formal language to state it and a proof assistant to prove it. We can also use the language for assertions and prove that this property can be derived from the final assertion. We explored these alternatives when discussed conformance and model typing.

Another option is to define a transformation contract stating the pre and post conditions of a transformation (or an individual rule), and check whether this contract holds. This contract can be written in OCL and then verified using an OCL checker or some other model-checker, as in [CBBD09, GM07]. It can also be written using a dedicated tool and then verified using some specific algorithm, as in [CHM+02, GdLW+13].

Functional Behavior

As defined in [CCGdL10], it is possible to check whether some properties hold considering either a specific rule or the whole model transformation. In our example both top rules are executable since there exists a valid pair of models (those in Figure 2.7) that satisfy them. Since they are the only top rules in the transformation, we can derive that the whole transformation is also executable.

Following the same ideas, neither the rule nor the transformation is total (Syntactic Completeness) since they apply only for persistent classes and then non-persistent classes will never be transformed. In this case it is clear that syntactic completeness is not desirable.

These properties can be verified by encoding them as UML/OCL consistency problems on the transformation model, as defined in [CCGdL10]. Then, an OCL-checker can be used. Another alternative is to verify them by static analysis of the transformation rules and the underlying metamodels, as in [PCG11].

Determinism and Termination

As we already said in the previous section, these two properties are the hardest to prove since they are related to undecidable problems. One alternative to achieve them is to use some language which guarantees both by construction, as introduced in [BLA+10]. However, this option clearly reduces expressive power.

Other alternatives are representing the transformation model in some formal language to perform logical inference, as in [LR10], or using static analysis, as in [Küs06]. However, the most referred alternative is expressing the transformation as a graph-rewriting problem which allows performing critical pair analysis, as in [Bru08, HEOG10].

Preservation of Execution Semantics

The preservation of the execution semantics is matter of verification during the development of a transformation engine. However, when defining a model transformation there are consistency needs that must be addressed. As an example, we may not want redundant rules, and indeed our example does not have redundancy. An example of a redundant rule would be one mapping attributes to columns but applicable only to persistent classes.

As before, we can encode these needs as a consistency problem on the transformation model and use a model-checker, as in [ABGR10], or use static analysis, as in [PCG11]. Moreover we can use logical inference as in [LD10].

Another possible approach is the followed in [WKK+09], where the transformation is expressed as a CPN which allows the formal exploration of CPN properties. In particular, the authors can verify whether there are transitions which are never enabled during execution, so called Dead Transition Instances or L0-Liveness.

4 An Introduction to Institutions

The concept of *Institution* was introduced in order to deal with the "population explosion among the logical systems used in computer science" [GB83]. It formalizes the notion of "logical system", which can be seen as a set of principles for some form of sound reasoning [MGDT05]. Many different logics, as first-order, modal and rewriting, have been shown to be institutions [ST12]. An interesting aspect is that within most specification formalisms there is a logical system allowing the user to write axioms describing the properties of the software system to be developed. In this sense, the notion of institution can be used to represent any specification language since it provides ways of representing the syntax and semantics of the language, as well as the relation between them by means of a satisfaction relation. Examples of this are the institutions defined for UML languages [CKTW08, CS11b].

In this chapter we introduce the Theory of Institutions which gives the formal foundations of our work. In Section 4.1 we present the main aspects of the Theory of Institutions needed for a deep understanding of the definitions in the following chapters. Then, in Section 4.2 we present an important result of this theory which is the possibility of using the proof calculus associated to an institution by another one through the definition of formal translations between them.

4.1 Institutions

An institution consists of vocabularies (called signatures) for constructing sentences in a logical system. A model (or interpretation) provides semantics by assigning interpretations to the elements in the signature. We can allow a change of interpretation defining the notion of homomorphism, which consists of a mapping of elements between two models. Institutions also define formal translations (called signature morphisms) between signatures, allowing many different vocabularies at once. Since signatures can change through signature mor-

phisms, we need to translate sentences and models accordingly. Sentences are translated along signature morphisms since symbols must be replaced in each sentence conforming to the signature morphism. In the case of models, they are translated in the opposite direction of signature morphisms. In particular, models are reduced since they provide an interpretation of elements in the signature and the signature morphism can translate elements to a bigger signature. Finally, there is a satisfaction relation of sentences by models, such that when a signature is changed (by a signature morphism), satisfaction of sentences by models changes consistently.

The formal definition of an institution relies on Category Theory [Lan98] in order to "capture structure arising from signature morphisms, as well as forcing an appropriate level of generality and abstraction" [ST12]. A *category* has objects and arrows, which associate two objects. Arrows can be composed, composition is associative, and there exist identity arrows for every object. For example, signatures and signature morphisms define a category $\mathbf{Sign}_{\mathcal{I}}$ such that signatures are the objects and the signature morphisms are the arrows. Moreover, a *functor* relates two categories, mapping objects to objects and arrows to arrows such that domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, and composition and identities are preserved. For example, there is a functor $Sen_{\mathcal{I}}$ in which each signature is mapped to a set of sentences (the set of possible sentences defined by the signature) and signature morphisms are mapped to morphisms between sentences (what we call the extension of a signature morphism to sentences).

Definition 4.1.1 (**Institution**)
An institution $\mathcal{I}$ (as defined in [ST12]) consists of:

1. a category $\mathbf{Sign}_{\mathcal{I}}$[1] of *signatures*;
2. a functor $Sen_{\mathcal{I}} : \mathbf{Sign}_{\mathcal{I}} \to \mathbf{Set}$, giving a set $Sen(\Sigma)$ of Σ-*sentences* for each signature $\Sigma \in |\mathbf{Sign}_{\mathcal{I}}|$ [2] and a function $Sen_{\mathcal{I}}(\sigma)$:$Sen_{\mathcal{I}}(\Sigma_1) \to Sen_{\mathcal{I}}(\Sigma_2)$ translating Σ_1-sentences to Σ_2-sentences for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$;
3. a functor $\mathbf{Mod}_{\mathcal{I}} : \mathbf{Sign}_{\mathcal{I}}^{op} \to \mathbf{Cat}$ [3] , giving a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature $\Sigma \in |\mathbf{Sign}_{\mathcal{I}}|$ and a functor $\mathbf{Mod}_{\mathcal{I}}(\sigma)$:$\mathbf{Mod}_{\mathcal{I}}(\Sigma_2) \to \mathbf{Mod}_{\mathcal{I}}(\Sigma_1)$ translating Σ_2-models to Σ_1-models (and Σ_2-morphisms to Σ_1-morphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$;
4. for each signature $\Sigma \in |\mathbf{Sign}_{\mathcal{I}}|$, a *satisfaction relation* $\models_{\mathcal{I},\Sigma} \subseteq |\mathbf{Mod}_{\mathcal{I}}(\Sigma)| \times Sen_{\mathcal{I}}(\Sigma)$;

such that for any signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ the translation $\mathbf{Mod}_{\mathcal{I}}(\sigma)$ of models and $Sen_{\mathcal{I}}(\sigma)$ of sentences preserve the satisfaction relation, that is, for any $\varphi \in Sen_{\mathcal{I}}(\Sigma)$ and $M_2 \in |\mathbf{Mod}_{\mathcal{I}}(\Sigma_2)|$:

$$M_2 \models_{\mathcal{I},\Sigma_2} Sen_{\mathcal{I}}(\sigma)(\varphi) \text{ iff } \mathbf{Mod}_{\mathcal{I}}(\sigma)(M_2) \models_{\mathcal{I},\Sigma_1} \varphi$$

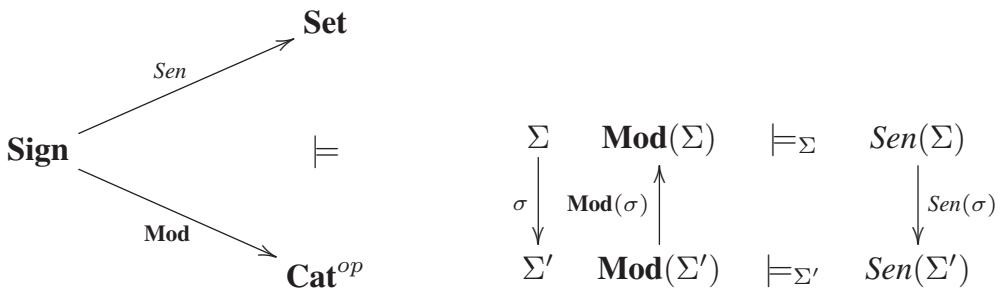
[1] We often omit the subscript $\mathcal{I}$
[2] $|C|$ is the collection of objects of a category C
[3] $\mathbf{Sign}^{op}$ is the opposite category of the category **Sign**

Example 4.1.2

An institution for first-order logic (FOL, [ST12]) is defined as follows. A FOL signature contains function symbols and predicates, and sentences are first-order formulas built out from atomic formulas (with variables and the logical constants true and false) using the standard propositional connectives ($\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$,$\neg$) and quantifiers ($\forall$, $\exists$). FOL models are structures giving an interpretation for variables, functions and predicates. Signature morphisms allow functions and predicates to change their names, and the extension of a signature morphism to sentences is the replacement of every function and predicate according to the signature morphism. In the case of models, the morphism induces a mapping called the *reduct*, defined as the interpretation of variables, functions and predicates in the target signature restricted to those reached from the source signature according to the signature morphism. Finally, the satisfaction relation is the usual satisfaction of a first-order sentence.

□

In Figure 4.1 there is a graphical representation of the elements of an institution and its relations. On the left side there is a representation of the categories defined for signatures (**Sign**), formulas (**Set**) and models ($\mathbf{Cat}^{op}$) and the functors relating them, as well as the satisfaction relation ($\models$) which relates formulas and models. On the right side there is a signature morphism σ allowing a change of notation between signatures Σ and Σ'. Sentences translate in the same direction as the change of notation, whereas models translate in the opposite direction. Because reversing the direction of morphisms gives a contravariant functor, the definition below uses $\mathbf{Cat}^{op}$, the opposite of the category of categories. The satisfaction condition states that *truth is invariant under change of notation* [GB92].



(a) Categories and functors (b) The satisfaction condition

Figure 4.1: Graphical view of an institution

The definition of an institution can be extended to consider not only an individual signature, but a *theory*, i.e. a pair $T = \langle \Sigma, \Psi \rangle$ consisting of a signature Σ and an arbitrary set of axioms, which are Σ-sentences (we set $Sig(T) = \Sigma$ and $Ax(T) = \Psi$). Moreover, it is possible to define a *theory morphism* $\sigma : \langle \Sigma, \Psi \rangle \to \langle \Sigma_2, \Psi_2 \rangle$ as the signature morphisms $\sigma : \Sigma_1 \to \Sigma_2$ for which $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$, i.e. those signature morphisms that map axioms to logical consequences. It is possible to extend *Sen* and **Mod** to start from the category

Th of theories by putting $Sen(\langle \Sigma, \Psi \rangle) = Sen(\Sigma)$ and letting $\mathbf{Mod}^{\mathrm{Th}}(\langle \Sigma, \Psi \rangle)$ be the full subcategory of $\mathbf{Mod}(\Sigma)$ induced by the class of those models satisfying Ψ. In this way, we get the institution of theories $\mathcal{I}^{\mathrm{Th}} = (\mathrm{Th}, Sen, \mathbf{Mod}^{\mathrm{Th}}, \models)$ over $\mathcal{I}$ [GB83].

4.2 Institution Morphisms

From a model-theoretic point of view, it is possible to define the notion of *logical consequence* or *semantic entailment* as follows.

Definition 4.2.1 (**Semantic entailment**)
Given a institution $\mathcal{I} = \langle \mathbf{Sign}, Sen, \mathbf{Mod}, \models_\Sigma \rangle$, a set of Σ-sentences Ψ and a Σ-sentence φ, we say $\Psi \models_\Sigma \varphi$ iff for all Σ-models M, we have

$$M \models_\Sigma \Psi \text{ implies } M \models_\Sigma \varphi$$

Here, $M \models_\Sigma \Psi$ means that $M \models_\Sigma \psi$ for each $\psi \in \Psi$. Moreover, it is possible to extend an institution from a proof-theoretic point of view by defining a *logic* in some way compatible with semantic entailment.

Definition 4.2.2 (**Logic**)
A logic $\mathcal{LOG} = \langle \mathbf{Sign}, Sen, \mathbf{Mod}, \models, \vdash \rangle$ is an institution $\langle \mathbf{Sign}, Sen, \mathbf{Mod}, \models_\Sigma \rangle$ equipped with an *entailment system* $\vdash$ that is, a relation between sentences $\vdash_\Sigma \subseteq P(Sen(\Sigma)) \times Sen(\Sigma)$ for each $\Sigma \in |\mathbf{Sign}|$, such that the following properties are satisfied.

- reflexivity: for any $\varphi \in Sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$
- monotonicity: if $\Psi \vdash_\Sigma \varphi$ and $\Psi' \supseteq \Psi$ then $\Psi' \vdash_\Sigma \varphi$
- transitivity: if $\Psi \vdash_\Sigma \varphi_i$ for $i \in I$ and $\Psi \cup \{\varphi_i | \, i \in I\} \vdash_\Sigma \psi$, then $\Psi \vdash_\Sigma \psi$
- $\vdash$-translation: if $\Psi \vdash_\Sigma \varphi$, then for any $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sign}$, $\sigma(\Psi) \vdash_{\Sigma'} \sigma(\varphi)$
- soundness: $\Psi \vdash_\Sigma \varphi$ implies $\Psi \models_\Sigma \varphi$, i.e. all provable statements are true

In some cases the entailment system can be complete: $\Psi \models_\Sigma \varphi$ implies $\Psi \vdash_\Sigma \varphi$, i.e. all true statements are provable. Entailment is typically defined via a system of finitary derivation rules, giving the notion of proof that is absent when the institution is considered on its own, even if $\vdash$ coincides with semantic entailment.

In order to use an institution for verification purposes, there are two alternatives: the definition of a logic, or the translation of the institution into another logic. Two institutions may be related through *institution morphisms* [GR02] which come in several flavors. In particular, there are so-called *institution comorphisms* which capture how a *weaker* and *poorer* institution can be represented in a *stronger* and *richer* one.

Definition 4.2.3 (**Institution comorphism**)
Given arbitrary institutions $\mathcal{I} = \langle \mathbf{Sign}^{\mathcal{I}}, \mathit{Sen}^{\mathcal{I}}, \mathbf{Mod}^{\mathcal{I}}, \models^{\mathcal{I}}_{\Sigma} \rangle$ and
$\mathcal{J} = \langle \mathbf{Sign}^{\mathcal{J}}, \mathit{Sen}^{\mathcal{J}}, \mathbf{Mod}^{\mathcal{J}}, \models^{\mathcal{J}}_{\Sigma} \rangle$, an *institution comorphism* $\rho : \mathcal{I} \to \mathcal{J}$ consists of:

- a functor $\rho^{Sign} : \mathbf{Sign}^{\mathcal{I}} \to \mathbf{Sign}^{\mathcal{J}}$
- a natural transformation $\rho^{Sen} : \mathit{Sen}^{\mathcal{I}} \to \rho^{Sign}; \mathit{Sen}^{\mathcal{J}}$
- a natural transformation $\rho^{Mod} : (\rho^{Sign})^{op}; \mathbf{Mod}^{\mathcal{J}} \to \mathbf{Mod}^{\mathcal{I}}$

such that for any signature $\Sigma \in |\mathbf{Sign}^{\mathcal{I}}|$ the translations ρ^{Sen}_{Σ} of sentences, and ρ^{Mod}_{Σ} of I-models, preserve the satisfaction relation, that is, for any $\varphi \in \mathit{Sen}^{\mathcal{I}}(\Sigma)$ and
$M \in \mathbf{Mod}^{\mathcal{J}}(\rho^{Sign}(\Sigma))$:

$$M \models^{\mathcal{J}}_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\varphi) \iff \rho^{Mod}_{\Sigma}(M) \models^{\mathcal{I}}_{\Sigma} \varphi$$

The functor ρ^{Sign} translates signatures and morphisms from one institution into the other. The natural transformation ρ^{Sen} translates sentences from one signature to the other in the same direction as the translation of signatures (as happens with signature morphisms). Moreover, the natural transformation ρ^{Mod} translates models from one institution into the other in the opposite direction (as happens with models of an institution). The satisfaction condition of a comorphism states that a translated model satisfies a sentence in the source institution only if the original model satisfies the translated sentence into the other institution.

Definition 4.2.4 (**Natural transformation**)
A natural transformation $\tau : F \Rightarrow G$ between two functors $F, G : \mathcal{C} \to \mathcal{D}$ is a family of arrows $\tau_A : FA \Rightarrow GA$, one for each object A of $\mathcal{C}$, such that, for every $f : A \to B$ it holds: $Gf \circ \tau_A = \tau_B \circ Ff$.

The naturality requirement amount to the facts that ρ^{Sen} and ρ^{Mod} are families of functions $\rho^{Sen}_{\Sigma} : \mathit{Sen}^{\mathcal{I}}(\Sigma) \to \mathit{Sen}^{\mathcal{J}}(\rho^{Sign}(\Sigma))$ and $\rho^{Mod}_{\Sigma} : \mathbf{Mod}^{\mathcal{J}}(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}^{\mathcal{I}}(\Sigma)$, respectively, such that for $\sigma : \Sigma_1 \to \Sigma_2$ the diagrams in Figure 4.2 commute.

The first diagram states that applying the extension of a signature morphism to a sentence in the institution $\mathcal{I}$ and then translating it to the institution $\mathcal{J}$ is the same as first translating the sentence from $\mathcal{I}$ and the applying the translated morphism in $\mathcal{J}$. The second diagram states that translating a model (or homomorphism) from the institution $\mathcal{J}$ and then applying the reduct in the institution $\mathcal{I}$ is the same as first applying the reduct in the institution $\mathcal{J}$ and then translating the resulting model (or homomorphism) to the institution $\mathcal{I}$.

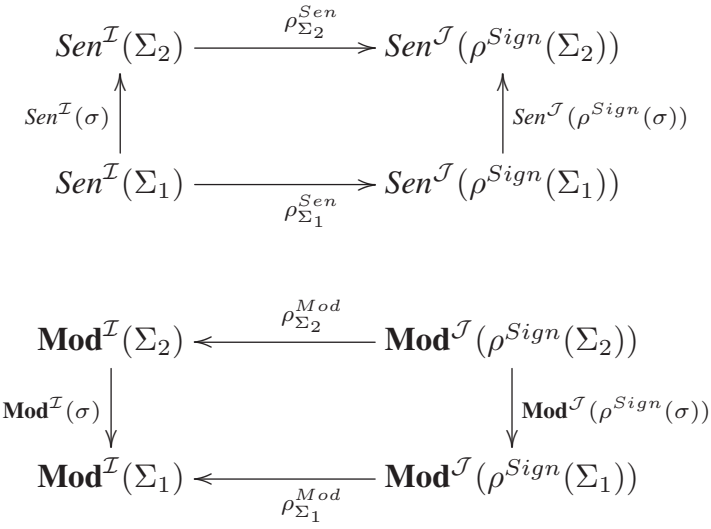


Figure 4.2: Naturality diagrams for an institution comorphism

The importance of comorphisms is such that it is possible (in some cases) to re-use the entailment systems of an institution in another one via a comorphism. This is possible thanks to the *borrowing* technique which can be stated as follows.

Definition 4.2.5 (**Borrowing of entailment**)
Let $\mathcal{I}$ and $\mathcal{J}$ be two institutions, $\rho = (\rho^{Sign}, \rho^{Sen}, \rho^{Mod}) : \mathcal{I} \to \mathcal{J}$ an institution comorphism, and $\mathcal{T}$ a class of $\mathcal{I}$-theories. We say that μ *admits borrowing of entailment for* $\mathcal{T}$, if for any theory $T = \langle \Sigma, \Psi \rangle \in \mathcal{T}$ and any Σ-sentence $\varphi \in \mathcal{I}$, we have

$$\Psi \vdash^{\mathcal{I}}_{\Sigma} \varphi \text{ iff } \rho^{Sen}(\Psi) \vdash^{\mathcal{J}}_{Sig(\rho^{Sign}(\Sigma))} \rho^{Sen}(\varphi).$$

In conclusion, as pointed out in [Mos05], "if we have a sound proof calculus for entailment in $\mathcal{J}$, and if we have an institution comorphism $\rho : \mathcal{I} \to \mathcal{J}$ admitting borrowing of entailment for $\mathcal{T}$, we can use the proof calculus also for proving entailment concerning $\mathcal{I}$-specifications in $\mathcal{T}$: we just have to translate our proof goals using ρ^{Sign} and ρ^{Sen}. If, moreover, the proof calculus is complete for proving entailment in $\mathcal{J}$, then also its re-use for proving entailment in $\mathcal{I}$ is complete."

There is a final result which states that if there is an institution comorphism $\rho : \mathcal{I} \to \mathcal{J}$ admitting *model expansion*, then it admits borrowing of entailment and refinement for theories. The comorphism admits model expansion if ρ^{Mod} is point-wise surjective on objects (i.e., each ρ^{Mod}_{Σ} is surjective on objects). This means that each model of the source institution has a model representing it in the target institution. If the comorphism does not admit model expansion, then the proof calculus of the target institution can be borrowed only for disproving entailment (not for proving it).

These last results also hold for other variants of comorphisms. In particular there is a variant that maps signatures to theories (known as *simple theoroidal comorphisms* [GR02]), i.e. an institution comorphism $(\rho^{Sign}, \rho^{Sen}, \rho^{Mod})$ from $\mathcal{I}$ to $\mathcal{J}^{th}$, where $\mathcal{J}^{th}$ is the institution of theories over $\mathcal{J}$. Simple theoroidal institution comorphisms capture the encoding of a "richer" institution into a "poorer" one.

However, there are cases when logic translations cannot be formalized as (theoroidal) comorphisms, e.g. when the target signature depends on the translation of sentences, thus the notion of *generalized theoroidal comorphisms* [Cod08] is defined, which is basically a theoroidal comorphism with no isolated sentence translation component. The following definition is a condensed version of the original one in [Cod08].

Definition 4.2.6 (**Generalized theoroidal institution comorphism**)
Given arbitrary institutions $\mathcal{I} = \langle \mathbf{Sign}^{\mathcal{I}}, \mathit{Sen}^{\mathcal{I}}, \mathbf{Mod}^{\mathcal{I}}, \models^{\mathcal{I}}_{\Sigma} \rangle$ and
$\mathcal{J} = \langle \mathbf{Sign}^{\mathcal{J}}, \mathit{Sen}^{\mathcal{J}}, \mathbf{Mod}^{\mathcal{J}}, \models^{\mathcal{J}}_{\Sigma} \rangle$, a generalized theoroidal institution comorphism $\mu : \mathcal{I} \to \mathcal{J}$ consists of:

- a functor $\Phi : \mathrm{Th}^{\mathcal{I}} \to \mathrm{Th}^{\mathcal{J}}$, with Th the category of theories and theory morphisms
- a natural transformation $\beta : (\Phi)^{op}; \mathbf{Mod}^{\mathcal{J}} \to \mathbf{Mod}^{\mathcal{I}}$, with $\mathbf{Mod} : \mathrm{Th} \to \mathbf{Cat}$ the functor giving the category of models of a theory.

5 An Environment for Verification

The myriad of alternatives summarized in Chapter 3 motivates a heterogeneous approach for the formal verification of model transformations as well as for their corresponding models and metamodels. With this motivation, we define an environment theoretically based on the Theory of Institutions, introduced in Chapter 4, and supported in practice within the Heterogeneous Tool Set ([MML07, Mos05]), as it is described in Chapter 10.

In this chapter we present an overview of the environment following the ideas presented in [CS13a]. In Section 5.1 we present our heterogeneous environment without giving formal details about the definition of its main components. Those details will be the subject of the following chapters. Then, in Section 5.2 we define how models, metamodels and the conformance relation between them can be described by an institution, and in Section 5.3 we do the same for model transformations. In Section 5.4 we define extensions of these institutions to be used for the definition of formal translations into other logics for verification purposes. Finally, in Section 5.5 we summarize the benefits of the environment and in Section 5.6 we close this chapter with a discussion of related work.

5.1 Defining the Environment

In Chapter 1 we explained how a separation of duties in different technological spaces is carried out for dealing with the specification and formal verification of model transformations, as is graphically depicted in Figure 5.1. On the one side there are those experts in the MDE domain specifying models and transformations, and on the other, those in formal methods (FM) conducting the verification process. Our proposal is to exploit the Theory of Institutions as a sound basis for constructing a FM technological space in which several logics can be used for verification.

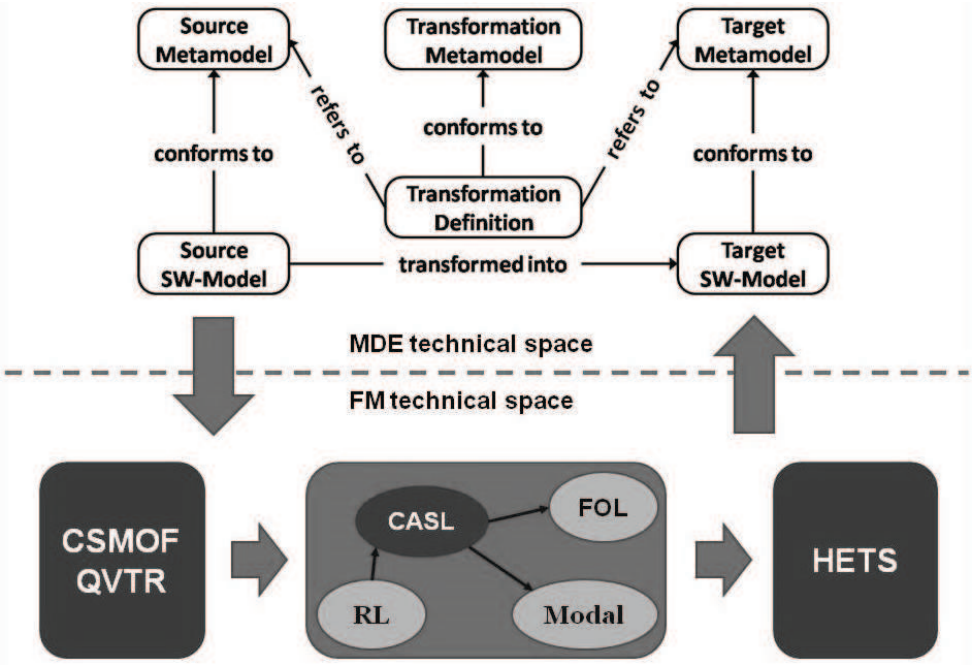


Figure 5.1: An overview of the environment

The idea is to represent models (from now on SW-models), metamodels, the conformance relation, transformations and verification properties in some consistent and interdependent way, following the heterogeneous specification approach [CKTW08, Mos05]. This approach is based on providing institutions for the languages which are part of the environment. Although this idea can be potentially formalized for any transformation approach and language, our proposal is aligned with the OMG standards, in particular: the MetaObject Facility (MOF, [OMG03b]), and the Query/View/Transformation Relations (QVT-Relations, [OMG09]) language.

In particular, we provide an institution for QVT-Relations, for expressing check-only unidirectional transformations (which we called QVTR), which will be introduced in Section 5.3. This kind of transformations verifies whether a target SW-model is the result of transforming the source SW-model, according to the transformation rules. To define this institution we need a representation of SW-models and metamodels. Therefore we first define an institution expressing the structural conformance relation between metamodels and SW-models (as presented in Section 2.1.1) specified with a simplified version of MOF (which we called CSMOF), that will be introduced in Section 5.2. These two institutions (represented in the leftmost box of the FM technological space in Figure 5.1) will provide a generic representation of the MDE elements with formal semantics in both cases.

As mentioned in Chapter 4, in order to use an institution for verification purposes, there are two alternatives: either the definition of a logic (defining a proof calculus for the institution), or the translation of the institution into another logic through an institution comorphism (borrowing an existent proof calculus). We take the second alternative and define comorphisms from an extended version of our institutions, as discussed in Section 5.4, to a host logic.

The definition of a comorphism not only gives us the chance of using some proof calculus, but also of supplementing the former specification of MDE elements with additional properties using the host logic. To the extent that there are many logics connected through comorphisms, the capabilities of the environment increase. As a very simple example, consider that we have a specification of the UML class diagrams metamodel in Figure 2.6 which is expressed with the institution CSMOF. It can be noticed that this specification does not have non-structural constraints. However, we can translate this specification into another logic and supplement it with a property stating that there cannot be two `Classifiers` with the same name in the `UMLMetamodel` specification. This constraint cannot be structurally stated using MOF. Thus using this translation we can perform a non-structural conformance checking.

In particular, we define comorphisms (generalized theoroidal comorphisms indeed) from our extended institutions to the Common Algebraic Specification Language (CASL, [MHST03]), a general-purpose specification language (in the center box of the FM technological space in Figure 5.1). The institution underlying CASL is the sub-sorted partial first-order logic with equality and constraints on sets $SubPCFOL^{=}$, a combination of first-order logic and induction with subsorts and partial functions.

The importance of CASL relies in that it is the main language within the Heterogeneous Tool Set (HETS, [MML07, Mos05]), which is a tool that supports heterogeneous multi-logic specifications (represented in the rightmost box of the FM technological space in Figure 5.1). HETS allows defining institutions and comorphisms, and also provides proof management capabilities for monitoring the overall correctness of a heterogeneous specification, while different parts of it are verified using (possibly different) proof systems. HETS already supports several interconnected logics as shown in Figure 5.2.

We provided HETS with specific institutions for the specification of MDE elements. These institutions are included as logics in such a way that it is possible to transform any specification into CASL or another logic reachable from it. A developer can import any MDE element, use the logics within HETS to specify additional verification properties which must be addressed, and perform the verification assisted by the tool.

With this approach we have only one generic representation of the MDE elements which is formally (and automatically) translated into other logics when needed, and those logics can also be used to specify additional verification properties. In some way, this is related with the slogan created by Sun Microsystems to illustrate the cross-platform benefits of the Java language: “write once, run anywhere”. In our case, we are trying to “specify once, prove anywhere”.

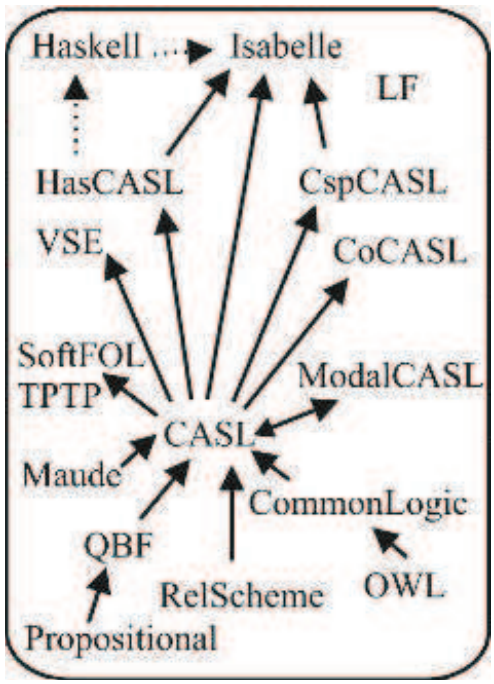


Figure 5.2: Logic graph of HETS

5.2 Representing SW-Models and Metamodels

Since we need to consider SW-models and metamodels for the definition of a model transformation, we define, in Chapter 6, an institution $\mathcal{I}^{\mathrm{M}}$ for a MOF-based structural conformance relation between these elements. We base our proposal on the institution defined for UML class diagrams in [CK08, JKMR12], but adapting the definitions with the purpose of representing metamodels.

For the definition of the institution we follow the schema in Figure 5.3, and we consider the simplified version of EMOF in Figure 2.3 for the representation of metamodels. From any metamodel we can derive a signature with a representation of types and properties (attributes and associations). Formulas represent multiplicity constraints determining whether the number of elements in a property end is bounded (upper and/or lower). A model contains a semantic representation of a SW-model. Given a model representing a SW-model, the satisfaction relation applied to set of multiplicity constraints derived from the metamodel answers the following question: does the SW-model structurally conforms to the metamodel?

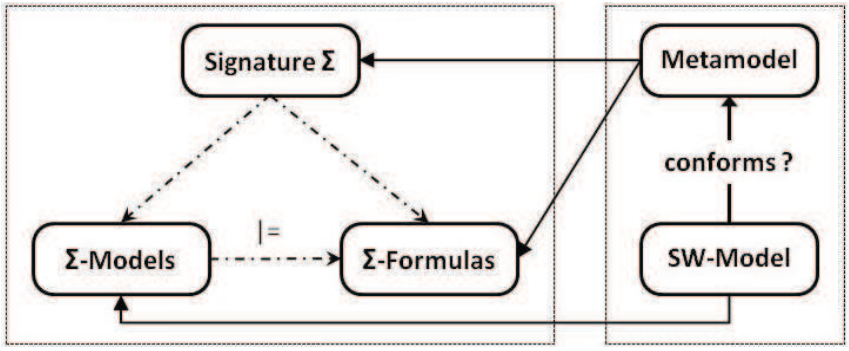


Figure 5.3: The conformance relation as an institution

It can be noticed that SW-models are always well-typed with respect to the metamodel, since any model is, by construction, well-typed with respect to the signature. The satisfaction relation only checks the multiplicity requirements of the structural conformance relation, as defined in Section 2.1.1. In this sense, type-checking is addressed in the construction of models, i.e. if the SW-model is not well-typed, then it is impossible to define the corresponding model. In order to fully understand the conformance requirements, we discuss in Chapter 6 how to address type-checking from an institutional perspective.

Moreover, non-structural conformance is not addressed by this institution, since no other supplementary language is considered, as discussed in Section 11.2.2. However, it is possible to consider other kind of constraints by translating (through comorphisms) this institution into a more expressive logic, as done in Chapter 9 and put in practice in Chapter 10.

5.3 Representing QVT Transformations

In Chapter 7 we define an institution $\mathcal{I}^{\mathrm{Q}}$ for QVT-Relations. For the definition of this institution we follow the schema shown in Figure 5.4. The institution is an extension of the CSMOF institution. The source and target metamodels that are involved in a specific model transformation are represented within the signature, and the source and target SW-models are represented in the model. Formulas represent the two basic conditions which must hold in a model transformation: keys defined on source and target metamodel elements, and transformation rules stating relations between source and target elements. The satisfaction relation checks if the keys hold in the corresponding SW-models, and it answers the following question: is the target SW-model the result of transforming the source SW-model according to the transformation rules?

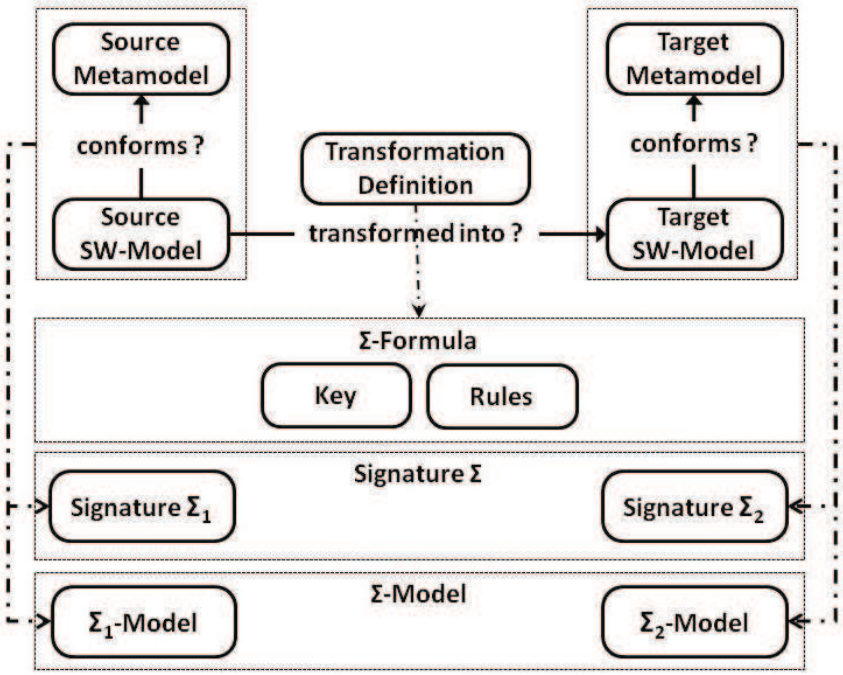


Figure 5.4: A model transformation as an institution

5.4 A Proof-Theoretic View

From a proof-theoretic point of view, we need to use (or define) an entailment system, as defined in Section 4.2, such that it is possible to prove that multiplicity constraints (or any other constraint) hold in a SW-model, which is the context in which the verification must be done. For this we extend the definition of CSMOF formulas to represent SW-models, as shown in Figure 5.5 and defined in Chapter 8.

Formulas representing SW-models are a syntactic representation of CSMOF models, and thus well-typed with respect to the signature (every object has a corresponding type in the signature and links are constrained by properties). The satisfaction relation needs to change in order to state whether a SW-model formula is satisfied by a model (if there is an isomorphism between them). By extending the CSMOF institution in this way, we can prove that for any model satisfying a SW-model formula, the same model also satisfies the multiplicity constraints.

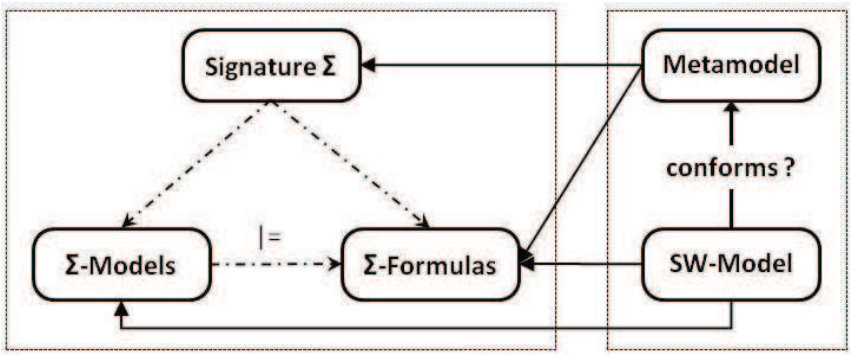


Figure 5.5: An extended CSMOF institution

In the case of QVTR the extension is similar. We extend QVTR formulas, as shown in Figure 5.6, and defined in Chapter 8.

Formulas include extended CSMOF formulas, i.e. now there is a representation of multiplicity constraints and SW-models, indexed by the institutions in which they are defined. The satisfaction relation is thus extended to consider these cases. The extension allows proving that for any pair of models satisfying the source and target SW-models and multiplicity constraints, such pair also satisfies the top transformation rule and key formulas.

These extended institutions are the ones used for the definition of the comorphisms to CASL and borrowing of its entailment system.

5.5 Benefits of the Environment

In general terms, our proposal has many benefits from a software engineering perspective. The first quality attribute we can remark is usability. The MDE expert specifies a model transformation and such specification is taken by the formal verification expert who defines the formal properties to be verified. HETS provides a graphical user interface that can be used for visualizing the whole proof and selecting a prover for the corresponding logic. If the proof is truly heterogeneous (i.e. there is more than one logic involved in the specification) the tool performs automatic translations of proof obligations into other logics and allows selecting the corresponding prover to be used. In order to do this there must be a comorphism linking the current logic with the others involved. The verification process is thus performed hiding the theoretical details of the environment from the developer. Moreover, since there is no need of rewriting the MDE building blocks in each logic involved, the environment is scalable in terms of the transformation specification.

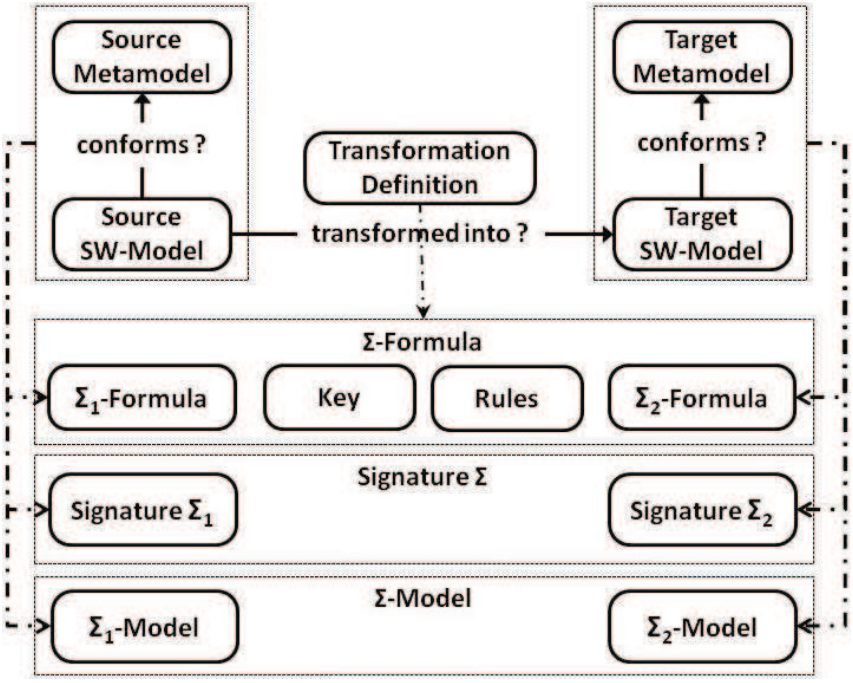


Figure 5.6: An extended QVTR institution

Another benefit is that the environment supports a separation of duties between software developers such that a formal perspective is available whenever it is required. This is a flexible approach which is enhanced by the explicit representation of every element involved in the verification of a transformation, which allows reasoning at different levels of abstraction. When the representation of MDE elements is translated into a specific domain, it conforms a *transformation model* [BBG+06b], based on the semantics of QVT-relations. This SW-model states how elements are related, and these relations allow answering different syntactic and semantic questions as defined in Chapter 3.

The environment is also extensible since it potentially supports any kind of logic and allows the inclusion of new logics. This allows a wide variety of verification approaches so that developers can work in the domain in which they feel more confident, or choose the tool they prefer, if they work in only one domain. The use of several domains is useful not only for verification but also for specification purposes, e.g. for the connection of MDE elements with more traditional software artifacts in a non-full MDE development.

Moreover, although our proposal is aligned with OMG standards, this idea can be potentially formalized for any transformation approach and language, which allows extending the approach as far as necessary.

Finally, the environment is reliable since it is supported by a well-founded theory and by a mature tool in which there are several logics already defined.

5.6 Related Work

There are some works that define environments for the comprehensive verification of MDE elements based on a unified mathematical formalism. In [RDV09, BHM09] rewriting logic is used to analyze MOF-like and QVT-like elements. Since rewriting logic was integrated into HETS [CMRM10], we can use these representations instead of using our comorphism into CASL. Nevertheless, since our institution is logic-independent it provides more flexibility for the definition of further specific comorphisms into other logics and languages (e.g. UML).

First-order logic [BEC12] and constructive type theory [CLST10b] have been used for the verification of ATL transformations. In [Sch10] the authors present how SW-models, metamodels and declarative transformation rules can be directly represented into Isabelle/HOL. Although these works do not refer to QVT-Relations, they are based on relational transformation languages with some similarities.

A different approach is in [Poe08, PT10] in which the authors outline how constructive type theory can be used to provide a uniform formal foundation for representing MDE elements (relational model transformations, but using any specific language). What is interesting about this proposal is that, given a proof, the proof-as-programs approach can be applied to extract a correct model transformation. The second paper presents how this approach could be implemented by using the Coq proof assistant. This is also achieved, but in the concrete case of ATL in [CLST10b, For13].

In [LR12] the authors define a language-independent representation of metamodels and model transformations supporting many transformation languages. They also define mappings to the B and Z3 formalisms. Since they use only one generic language, only one semantic mapping needs to be defined for each target formalism. However, the semantic mapping should be semantics-preserving, and this aspect is not formally addressed in such work. In our case, comorphisms already preserve the semantics with respect to the satisfaction relation. Moreover, our comorphism into CASL and the corresponding implementation in HETS, provides the possibility of connecting our institutions to several logics and tools.

Other works are based on the construction of a *transformation model* [BBG+06b] which is a unified representation of the MDE elements using OCL contracts. In [BCG11, CBBD09, CCGdL10] the authors present how to extract OCL contracts from several model transformation languages to conform a transformation model. These SW-models can be easily checked and analyzed using readily available OCL model finders. In [ALL10] the authors propose a similar technique but create a new language called the assertion description language (ADL) to specify the transformation model.

As far as we know, there is no other comprehensive institutional approach, apart from our work. However, there are some works in which algebraic specifications and institutions are involved. In [BKMW08] the authors propose to represent metamodels as institutions, and the correctness of a transformation is stated in terms of the existence of an intermediate institution that relates the source and target institutions through some institution morphism (which is restrictive). In [OW09] the authors provide an algebraic representation of MDE elements in which metamodels are represented as algebras and transformations as triple patterns. They devise two options for the implementation of this approach: either using a tool like Maude that would allow them to directly work on algebras, or specializing the approach to the case of graphs and using some graph transformation tool. In [CKTW08] the authors define a heterogeneous approach to the semantics of UML. In this work the institution of UML class diagrams was defined, on which our CSMOF institution is based.

Although several approaches to heterogeneous specification have been developed for traditional software development, there is little tool support. CafeOBJ [DF98] is a prominent approach based on the theory of institutions. However it provides a fixed cube of eight logics and twelve projections (formalized as institution morphisms), not allowing logic encodings (formalized as so called comorphisms). Thus, it is not an option for the definition of our environment. Moreover, HeteroGenius [GMLF14] is a framework, based on institutions, allowing the interaction between different external tools giving the user the possibility of performing hybrid analysis of a specification. However, the framework is not formally defined or available to be used as a basis for our environment.

6 An Institution for CSMOF

In this chapter we present the formal definition of an institution $\mathcal{I}^{\mathrm{M}}$ for the structural conformance relation between SW-models and metamodels specified with a simplified version of MOF (which we call CSMOF). This definition is based on the institutions for UML class diagrams presented in [CK08, JKMR12], but adapted for representing metamodels within the scope of model transformations. The institution is an updated version of the one presented in [CS13b, CS13c].

In Section 6.1 we present some preliminary considerations with respect to the MOF standard and the previous works. Then, in Section 6.2 we define the syntactic aspects of the institution (signatures and formulas). In Section 6.3 we define the notion of institution model, which allows us to state, in Section 6.4, the satisfaction relation and the corresponding satisfaction condition of the institution. Finally, in Section 6.5 we present some related work, and in Section 6.6 we close this chapter discussing how to address typing requirements of the structural conformance relation. Along the definition we illustrate the main concepts with the example introduced in Chapter 2. For the sake of readability we do not include complete proofs of institution properties, which are given in Appendix B.

6.1 Preliminaries

As mentioned in Chapter 5, this institution is based on the institutions for UML class diagrams presented in [CK08, JKMR12]. Unlike [CK08], in our definition there are no derived relations, the signature has an explicit representation of abstract classes, and there are only binary properties. Derived relations and n-ary properties are not used within transformations, and abstract classes where not considered in the former works.

Moreover, unlike MOF, we do not consider aggregation, uniqueness and ordering properties within a property end, operations on classes, or packages. Aggregation and operations are not used within transformations, whilst packages are just used for organizing metamodel elements (since they can be considered syntactic sugar). Although uniqueness and ordering properties are neither commonly used, their future inclusion will improve the institution.

Finally, as explained in Section 5.2, this institution only considers the multiplicity requirements of the structural conformance relation. Neither typing requirements (discussed in Section 6.6) nor non-structural conformance (addressed by means of a comorphism into a more expressive logic, as done in Chapter 9) are addressed by the satisfaction relation.

6.2 Signatures and Formulas

A signature represents a metamodel, i.e. it defines hierarchical related classes, primitive types and type constructors. For such reason we first introduce class hierarchies.

Definition 6.2.1 (**Class hierarchy and type extension**)
A class hierarchy is represented as a partial order $\boldsymbol{C} = (C, \leq_C)$ where C is a set of class names, and $\leq_C \subseteq C \times C$ is the subclass (inheritance) relation. By $\boldsymbol{T}(\boldsymbol{C})$ we denote the *type extension* of $\boldsymbol{C}$ by primitive types (e.g. Boolean and String) and type constructors (e.g. List and Set). $\boldsymbol{T}(\boldsymbol{C})$ is likewise a class hierarchy $(T(C), \leq_{T(C)})$ with $C \subseteq T(C)$ and $\leq_C \subseteq \leq_{T(C)}$, which is closed with respect to types, and "downwards" closed with respect to type constructors.

We can consider a fixed set of primitive types and type constructors similar to those defined for the OCL [OMG10]. As in [JKMR12], in order to provide generic access to primitive types and type constructors, we treat these as built-in with a standard meaning. All other classes are assumed to be inhabited, i.e., to contain at least one object. However, unlike [JKMR12] in which the existence of an object $null$ is assumed, we impose that if c is abstract then there exists another c' in the hierarchy such that $c' \leq_{T(C)} \ldots \leq_{T(C)} c$ and c' has at least one object.

Definition 6.2.2 (CSMOF **signature**)
A CSMOF signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ declares:

- a finite class hierarchy $\boldsymbol{C} = (C, \leq_C)$ extended with a subset $\alpha \subseteq C$ denoting abstract classes
- a properties declaration (attributes and associations) $\boldsymbol{P} = (R, P)$ where R is a finite set of role names with a default role name "_", and P is a finite set of properties of the form $\langle r_1 : c_1, r_2 : c_2 \rangle$ with $r_1, r_2 \in R$, $c_1, c_2 \in T(C)$, such that for any class or type name $c \in T(C)$, the role names of the properties in which any $c' \leq_{T(C)} c$ is involved are all different, i.e. if $\langle r_1 : c_1, r_2 : c_2 \rangle$ and $\langle s_1 : d_1, s_2 : d_2 \rangle$ are properties in P and $c_k = d_l \in T(C)$, then $r_i \neq s_j$ for any $i \neq k$ and for any $j \neq l$ ($1 \leq i \leq 2$, $1 \leq j \leq 2$)

Any property declaration $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$ represents a MOF property and its opposite (if any, as explained in Section 2.1.1), such that the type c_i attached to the role r_i represents the type of the property, as well the type in the opposite side represents its owned class. The default role name "_" is used if a property has no opposite. For example, in the case of an attribute r of type d in the type c, the property declaration will be $\langle \_ : c, r : d \rangle$, and in the case of an unidirectional association, the role in the opposite side of the arrow must also be the default role name "_".

Example 6.2.3

From the class metamodel in Figure 2.6 we derive the signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, such that:

$$
\begin{array}{lcl}
C & = & \{\text{UMLModelElement}, \text{Package}, \text{Classifier}, \text{PrimitiveDataType}, \\
 & & \ \text{Attribute}, \text{Class}\} \text{ and } T(C) \text{ also contains type } \text{String} \\
\leq_C & = & \{\text{Package} \leq_C \text{UMLModelElement}, \\
 & & \ \text{Attribute} \leq_C \text{UMLModelElement}, \\
 & & \ \text{Classifier} \leq_C \text{UMLModelElement}, \\
 & & \ \text{Class} \leq_C \text{Classifier}, \text{PrimitiveDataType} \leq_C \text{Classifier}\} \\
\alpha & = & \{\text{UMLModelElement}\} \\
R & = & \{\text{namespace}, \text{elements}, \text{type}, \text{owner}, \text{attribute}, \text{name}, \text{kind}\} \\
P & = & \{\ \langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier} \rangle, \\
 & & \ \langle \_ : \text{UMLModelElement}, \text{name} : \text{String} \rangle, \\
 & & \ \langle \_ : \text{UMLModelElement}, \text{kind} : \text{String} \rangle, \\
 & & \ \langle \text{attribute} : \text{Attribute}, \text{owner} : \text{Class} \rangle, \\
 & & \ \langle \_ : \text{Attribute}, \text{type} : \text{PrimitiveDataType} \rangle\}
\end{array}
$$

□

Formulas represent multiplicity constraints, i.e. determining whereas the number of elements in a property end is bounded (upper and/or lower).

Definition 6.2.4 (CSMOF **formula**)

Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, a Σ-formula representing a multiplicity constraint is defined by the following grammar:

$$
\begin{array}{lcl}
\Phi & ::= & \#\Pi = \mathbb{N} \mid \mathbb{N} \leq \#\Pi \mid \#\Pi \leq \mathbb{N} \\
\Pi & ::= & C \bullet R
\end{array}
$$

The $\#$-expressions return the number of links in a property when some role is fixed. We use $\bullet$ as an operator representing the selection of the elements linked with an element of class $c \in C$ through role $r \in R$; there must exist a property $\langle r' : c, r : d \rangle$ or $\langle r : d, r' : c \rangle$ in P.

Example 6.2.5

The set of formulas φ corresponding to the metamodel in Figure 2.6 is defined by:

$$\begin{aligned}\varphi \quad = \quad & \{\, \#(\text{UMLModelElement} \bullet \text{name}) = 1, \#(\text{UMLModelElement} \bullet \text{kind}) = 1, \\ & 1 \leq \#(\text{Class} \bullet \text{attribute}), \#(\text{Class} \bullet \text{attribute}) \leq 2, \\ & \#(\text{Attribute} \bullet \text{type}) = 1, \#(\text{Attribute} \bullet \text{owner}) = 1, \\ & \#(\text{Classifier} \bullet \text{namespace}) = 1\}\end{aligned}$$

The formulas allow representing any kind of multiplicity, e.g. that the number of attributes of a class has a lower bound of one element, and also and upper bound of two. In the other cases there are only lower bounds defined. If a property end has no associated formula, it means that there is no bound defined for it.

□

Definition 6.2.6 (CSMOF **signature morphism**)

Given signatures $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1, 2)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ is a tuple of maps $\langle \sigma_T, \sigma_R \rangle$ between class and role names, such that hierarchical relations and properties change consistently, i.e.

- $a \in C_1$ implies $\sigma_T(a) \in C_2$ (the extension of σ_T to $T(C)$ leaves built-in types unchanged),
- $a, b \in C_1$ with $a \leq_{C_1} b$ implies $\sigma_T(a) \leq_{C_2} \sigma_T(b)$,
- $a \in \alpha_1$ implies $\sigma_T(a) \in \alpha_2$,
- $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ implies $\langle \sigma_R(r_1) : \sigma_T(c_1), \sigma_R(r_2) : \sigma_T(c_2) \rangle \in P_2$

Signature morphisms extend to formulas over Σ_1 as follows. Given a Σ_1-formula φ, $\sigma(\varphi)$ is the canonical application of the signature morphism to the type and role in the formula such that $\sigma(c \bullet r) = \sigma_T(c) \bullet \sigma_R(r)$.

The following lemmas state basic properties of the institution: that signatures and signature morphisms define a category **Sign**, and also that there is a functor *Sen* from this category to the category of sets of formulas and their translations. For the sake of readability we only show a proof sketch. The complete proofs are given in Appendix B.

Lemma 6.2.7. *Signatures and signature morphisms define a category* ***Sign****. The points of the category are the signatures and the arrows are the signature morphisms.*

Proof sketch. Given signatures $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1..3)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$, and $\boldsymbol{P}_i = (R_i, P_i)$, and signature morphisms $\sigma_i : \Sigma_i \to \Sigma_{i+1}$ $(i{=}1..2)$, we can define the composition of signature morphisms as the tuple $\langle \sigma_T, \sigma_R \rangle$ such that $\sigma_T(c) = \sigma_{T_2}(\sigma_{T_1}(c))$ for any $c \in C_1$, and $\sigma_R(r) = \sigma_{R_2}(\sigma_{R_1}(r))$ for any $r \in R_1$. This composition defines a signature morphism which is also associative. Moreover, we can define the identity signature morphism as a tuple $\langle id_T, id_R \rangle$ such that $id_T(c) = c$, and $id_R(c) = c$. This morphism also satisfies the signature morphism conditions. Finally, with these elements the category **Sign** can be defined. □

Lemma 6.2.8. *There is a functor Sen giving a set of formulas ψ (object in the category **Set**) for each signature Σ (object in the category **Sign**), and a function $\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$ (arrow in the category **Set**) translating formulas for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**).*

Proof sketch. The signature morphism in Definition 6.2.6 changes types and roles consistently. In this sense, its application to any formula in $Sen(\Sigma_1)$ gives a formula in $Sen(\Sigma_2)$ with the types and roles translated with respect to the signature morphism $\sigma : \Sigma_1 \to \Sigma_2$. In this sense, the domain and codomain of the image of an arrow in **Set** are the images of domain and codomain, respectively, of the arrow in **Sign**. Moreover, since types and roles are translated consistently with respect to a signature morphism, and signature morphism can be composed (as defined in Lemma 6.2.7), the composition with respect to formulas is also preserved. For the same reason, identities are preserved. Finally, the functor *Sen* is defined.

□

6.3 Models

An interpretation (or model) contains a semantic representation for a SW-model, i.e. objects and links. For such reason we need to define the notion of object domain with respect to a class hierarchy.

Definition 6.3.1 (**Object domain and value extension**)
Given a class hierarchy $\boldsymbol{C} = (C, \leq_C)$, a $\boldsymbol{C}$-object domain $\boldsymbol{O}$ is a family $(O_c)_{c \in C}$ of sets of object identifiers verifying $O_{c_1} \subseteq O_{c_2}$ if $c_1 \leq_C c_2$. Given a type extension $\boldsymbol{T}$, the value extension of a $\boldsymbol{C}$-object domain $\boldsymbol{O} = (O_c)_{c \in C}$ by primitive values and value constructions, which is denoted by $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O})$, is a $\boldsymbol{T}(\boldsymbol{C})$-object domain $(V_c)_{c \in T(C)}$ such that $V_c = O_c$ for all $c \in C$. We consider disjoint sets of objects within the same hierarchical level, in particular, if $c_1 \leq_C c$ and $c_2 \leq_C c$, then $O_{c_1} \cap O_{c_2} = \emptyset$.

Definition 6.3.2 (CSMOF **interpretation**)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, a Σ-interpretation $\mathcal{I}$ consists of a tuple $(\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$ where

- $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}) = (V_c)_{c \in T(C)}$ is a $\boldsymbol{T}(\boldsymbol{C})$-object domain
- $\boldsymbol{A}$ contains a relation $\langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}} \subseteq V_{c_1} \times V_{c_2}$ for each relation name $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$ with $c_1, c_2 \in T(C)$
- $c_2 \in \alpha$ implies $O_{c_2} = \bigcup_{c_1 \leq_C c_2} O_{c_1}$

Example 6.3.3

An interpretation $\mathcal{I}$ can have one element for each type in the signature as follows:

- A $\boldsymbol{T}(\boldsymbol{C})$-object domain consisting of

$$\begin{array}{rclrcl} V_{\text{Class}} & = & \{\texttt{c1}\} & V_{\text{PrimitiveDataType}} & = & \{\texttt{pdt1}\} \\ V_{\text{Package}} & = & \{\texttt{p1}\} & V_{\text{Classifier}} & = & V_{\text{Class}} \cup V_{\text{PrimitiveDataType}} \\ V_{\text{Attribute}} & = & \{\texttt{a1}\} & V_{\text{UMLModelElement}} & = & V_{\text{Classifier}} \cup V_{\text{Package}} \cup V_{\text{Attribute}} \\ & & & V_{\text{String}} & = & \{\texttt{Pac}, \texttt{Str}, \texttt{Per}, \texttt{nul}, \texttt{ID}, \texttt{val}\} \end{array}$$

- A set $\boldsymbol{A}$ consisting of relations:

$$\begin{array}{lcl} \langle \_ : \text{UMLModelElement}, \text{name} : \text{String} \rangle^{\mathcal{I}} & = & \{(\texttt{p1}, \texttt{Pac}), (\texttt{c1}, \texttt{ID}), \\ & & \ (\texttt{c2}, \texttt{nul}), (\texttt{a1}, \texttt{val})\} \\ \langle \_ : \text{UMLModelElement}, \text{kind} : \text{String} \rangle^{\mathcal{I}} & = & \{(\texttt{p1}, \texttt{nul}), (\texttt{c1}, \texttt{Per}), \\ & & \ (\texttt{a1}, \texttt{nul}), (\texttt{pdt1}, \texttt{nul})\} \\ \langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier} \rangle^{\mathcal{I}} & = & \{(\texttt{p1}, \texttt{c1}), (\texttt{p1}, \texttt{pdt1})\} \\ \langle \text{attribute} : \text{Attribute}, \text{owner} : \text{Class} \rangle^{\mathcal{I}} & = & \{(\texttt{a1}, \texttt{c1})\} \\ \langle \_ : \text{Attribute}, \text{type} : \text{PrimitiveDataType} \rangle^{\mathcal{I}} & = & \{(\texttt{a1}, \texttt{pdt1})\} \end{array}$$

□

Given a Σ-interpretation, the evaluation of an expression $c_i \bullet r_i$, associated to a property $\langle r_1 : c_1, r_2 : c_2 \rangle$, with respect to the interpretation gives a set of sets of pairs of semantic elements connected through property $\langle r_1 : c_1, r_2 : c_2 \rangle$, grouped by the semantic elements with type c_i. Note that this set can be empty if the element with type c_i is not connected with any other. Formally:

Definition 6.3.4 (Evaluation of properties)

Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, a Σ-interpretation $\mathcal{I} = (\boldsymbol{V}_{\boldsymbol{C}}^{\boldsymbol{T}}(\boldsymbol{O}), \boldsymbol{A})$, and a property $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, we define the evaluation of $c_i \bullet r_j$ as follows:

$$(c_i \bullet r_j)^{\mathcal{I}} = \{\{t \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}} \mid \pi_i(t) = o\} \mid o \in V_{c_i}\} \ (i = 1, 2).$$

Example 6.3.5

Consider the property $\langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier} \rangle$ representing that a package contains classifiers. This interpretation evaluates $(\text{Classifier} \bullet \text{namespace})^{\mathcal{I}}$ as the set $\{\{(\texttt{p1}, \texttt{c1})\}, \{(\texttt{p1}, \texttt{pdt1})\}\}$ since there is only one object with role namespace which is the package object p1, and there are two elements (c1 and pdt1) in the opposite side of the property to group by.

□

Definition 6.3.6 (CSMOF **homomorphism**)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, Σ-interpretations $\mathcal{I} = (\boldsymbol{V_C^T}(\boldsymbol{O}), \boldsymbol{A})$ and $\mathcal{I}' = (\boldsymbol{V_C^T}(\boldsymbol{O})', \boldsymbol{A}')$, a Σ-homomorphism $h : \mathcal{I} \to \mathcal{I}'$ is a family of maps $(h_c)_{c \in T(C)}$ with $h_c : V_c \to V_c'$ such that:

- $h_c(v) \in O_c'$ forall $v \in O_c$
- $h_c(v) \in V_c' \backslash O_c'$ forall $v \in V_c \backslash O_c$
- $(v_1, v_2) \in p^{\mathcal{I}}$ iff $(h_{c_1}(v_1), h_{c_2}(v_2)) \in p^{\mathcal{I}'}$ for any $v_i \in V_{c_i}$ (i=1,2), $p = \langle r_1 : c_1, r_2 : c_2 \rangle \in P$.

We can prove now that interpretations and homomorphisms define a category **Mod**(Σ). Again here we just provide a sketch of the proof. The complete proof is given in Appendix B.

Lemma 6.3.7. *For any signature Σ, the Σ-interpretations and Σ-homomorphisms define a category **Mod**(Σ). The points of the category are the Σ-interpretations, and its arrows are the Σ-homomorphisms.*

Proof sketch. We need to show that homomorphisms can be composed, that the composition of homomorphisms is associative, and that there exists identity homomorphisms. Let $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$, and $\boldsymbol{P} = (R, P)$ be a signature, let $\mathcal{I}_i = (\boldsymbol{V_C^T}(\boldsymbol{O}_i), \boldsymbol{A}_i)$ $(i{=}1..3)$ be Σ-interpretations, and let $h_i : \mathcal{I}_i \to \mathcal{I}_{i+1}$ $(i{=}1..2)$ be Σ-homomorphisms. It is possible to define an associative composition $h_2 \circ h_1$ as a family of maps $(h_c)_{c \in T(C)}$ with $h_c : V_{c_1} \to V_{c_3}$ such that $h_c(v) = h_{c_2}(h_{c_1}(v))$ for all $v \in O_{c_1}$. The composition is a homomorphism. Moreover, there exists an identity homomorphism $id_{\mathcal{I}} : \mathcal{I} \to \mathcal{I}$ consisting of a family of maps $(id_c)_{c \in T(C)}$ with $id_c : V_c \to V_c$ such that: $id_c(v) = v$, $v \in O_c$. Finally, interpretations and homomorphisms define a category.

□

6.4 Satisfaction Relation and Satisfaction Condition

As explained in Section 5.2, we must express that a SW-model conforms to a metamodel if it is well-typed and it also satisfies its multiplicity constraints. Well-typing holds by construction, since the interpretation representing a SW-model respects the signature which defines types within the metamodel. The satisfaction of multiplicity constraints (formulas) by a SW-model (interpretation) is thus the main concern of the satisfaction relation.

Definition 6.4.1 (CSMOF **satisfaction relation**)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, a Σ-formula φ representing a multiplicity constraint and a Σ-interpretation $\mathcal{I}$, the interpretation satisfies φ, if one of the following holds:

- φ is $\#(c \bullet r) = n$ and $|S| = n$ for all $S \in (c \bullet r)^{\mathcal{I}}$
- φ is $n \leq \#(c \bullet r)$ and $n \leq |S|$ for all $S \in (c \bullet r)^{\mathcal{I}}$
- φ is $\#(c \bullet r) \leq n$ and $|S| \leq n$ for all $S \in (c \bullet r)^{\mathcal{I}}$

This means that for any object of class c, the number of elements within $\mathcal{I}$ related through the role r (of a property of the class c) satisfies the multiplicity constraints.

This definition can be trivially extended for a set of formulas Φ, as follows: $\mathcal{I} \models_\Sigma \Phi$ iff $\mathcal{I} \models_\Sigma \varphi. \forall \varphi \in \Phi$.

Example 6.4.2
We can check that $\mathcal{I} \models_\Sigma \varphi$ for every formula φ representing a multiplicity constraints defined before.

- $\#(\text{UMLModelElement} \bullet \text{name}) = 1$ and $|S| = 1$
  for all $S \in (\text{UMLModelElement} \bullet \text{name})^{\mathcal{I}}$ =
  $\{\{(p1, Pac)\}, \{(c1, ID)\}, \{(a1, val)\}, \{(pdt1, Str)\}\}$
- $\#(\text{UMLModelElement} \bullet \text{kind}) = 1$ and $|S| = 1$
  for all $S \in (\text{UMLModelElement} \bullet \text{kind})^{\mathcal{I}}$ =
  $\{\{(c1, Per)\}, \{(pdt1, nul)\}, \{(a, nul)\}, \{(p, nul)\}\}$
- $\#(\text{Classifier} \bullet \text{namespace}) = 1$ and $|S| = 1$
  for all $S \in (\text{Classifier} \bullet \text{namespace})^{\mathcal{I}}$ = $\{\{(p1, c1)\}, \{(p1, pdt1)\}\}$
- $\#(\text{Attribute} \bullet \text{owner}) = 1$ and $|S| = 1$
  for all $S \in (\text{Attribute} \bullet \text{owner})^{\mathcal{I}}$ = $\{\{(a1, c1)\}\}$
- $\#(\text{Attribute} \bullet \text{type}) = 1$ and $|S| = 1$
  for all $S \in (\text{Attribute} \bullet \text{type})^{\mathcal{I}}$ = $\{\{(a1, pdt1)\}\}$
- $1 \leq \#(\text{Class} \bullet \text{attribute})$ and $1 \leq |S|$
  for all $S \in (\text{Class} \bullet \text{attribute})^{\mathcal{I}}$ = $\{\{(a1, c1)\}\}$
- $\#(\text{Class} \bullet \text{attribute}) \leq 2$ and $|S| \leq 2$
  for all $S \in (\text{Class} \bullet \text{attribute})^{\mathcal{I}}$ = $\{\{(a1, c1)\}\}$

□

Definition 6.4.3 (CSMOF **reduct**)
Given signatures $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$ $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, and a Σ_2-interpretation $\mathcal{I} = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$, the reduct $\mathcal{I}|_\sigma$ of $\mathcal{I}$ along σ is the Σ_1-interpretation $\mathcal{I}_1 = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}|_\sigma), \boldsymbol{A}|_\sigma)$ with

- $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}|_\sigma) = (V_{\sigma(c)})_{c \in T(C_1)}$
- $\boldsymbol{A}|_\sigma = \{\langle \sigma_R(r_1) : \sigma_T(c_1), \sigma_R(r_2) : \sigma_T(c_2) \rangle^{\mathcal{I}} \mid \langle r_1 : c_1, r_2 : c_2 \rangle \in P_1\}$

Moreover, given Σ_2-interpretations $\mathcal{I}_2 = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}_2), \boldsymbol{A}_2)$ and $\mathcal{I}'_2 = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}_2)', \boldsymbol{A}'_2)$, $\mathcal{I}_1$ denoting $\mathcal{I}_2|_\sigma$ and $\mathcal{I}'_1$ denoting $\mathcal{I}'_2|_\sigma$, and a Σ_2-homomorphism $h_2 : \mathcal{I}_2 \to \mathcal{I}'_2$, the reduct $h_2|_\sigma$ of h_2 along σ is the Σ_1-homomorphism $h_1 : \mathcal{I}_1 \to \mathcal{I}'_1$ defined by $h_{1_c}(v) = h_{2_{\sigma(c)}}(v)$ for any $c \in T(C_1)$, for any $v \in V_c$. It is possible to check that h_1 is indeed a Σ_1-homomorphism, since h_2 is a homomorphism and $h_2|_\sigma$ is defined for elements in $T(C_1)$.

The following lemmas prove that the reduct defines a functor and thus there is a functor **Mod** giving a category of interpretations for each signature and a functor defined by the reduct. Again here we just provide a sketch of the proofs. The complete proofs are given in Appendix B.

Lemma 6.4.4. *The reduct of Σ-interpretations and Σ-homomorphisms is a functor **Mod**(σ) from Σ_2-interpretations (Σ_2-homomorphisms) to Σ_1-interpretations (Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$.*

Proof sketch. By definition, domain and codomain of the reduct of an homomorphism are the reduct of domain and codomain, respectively, of the homomorphism. Also, the reduct of a composition of two homomorphisms gives an interpretation for elements which are reached by the signature morphism, which coincide with the composition of reduced homomorphisms. Moreover, by definition of reduct of a homomorphism, the reduct of an identity Σ_2-homomorphism is an identity Σ_1-homomorphism (only restricts that the identity homomorphism applies to elements in Σ_1). Finally, the reduct of interpretations and homomorphisms is a functor.

□

Lemma 6.4.5. *There is a functor **Mod** giving a category **Mod**(Σ) of Σ-interpretations (object in the category **Cat**) for each signature Σ (object in the category **Sign**), as shown in Lemma 6.3.7, and a functor **Mod**(σ) (arrow in the category **Cat**) from Σ_2-interpretations to Σ_1-interpretations (and Σ_2-homomorphisms to Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**), as shown in Lemma 6.4.4.*

Proof sketch. By Lemma 6.4.4, the image of an arrow $\sigma : \Sigma_2 \to \Sigma_1$ in the category $\mathbf{Sign}^{op}$ is an arrow $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ in the category **Cat**. Also, by Lemma 6.3.7, the image of a signature Σ in the category **Sign** is an object $\mathbf{Mod}(\Sigma)$ in the category **Cat**. Thus, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow. Now, let Σ_i $(i{=}1..3)$ be signatures, let $\sigma_i : \Sigma_i \to \Sigma_{i+1}$ $(i{=}1, 2)$ be signature morphisms. A Σ_3-interpretation reduced with respect to the composed signature

morphism $\sigma_2 \circ \sigma_1$ gives an interpretation for the elements in Σ_3 reached by the composed signature morphism (corresponding to interpretation of elements in Σ_1), which by definition of composition it is the same as reducing the interpretation to the elements reached from Σ_2, and then again reduced to those reached from Σ_1. In the case of homomorphisms the reasoning is similar. Moreover, following the same reasoning it is straightforward to prove that $\mathbf{Mod}(id_\sigma)$ is an identity functor, i.e., it is composed by the identity reduct of Σ-interpretations and the identity reduct of Σ-homomorphisms. Finally, the functor **Mod** is defined.

□

We close the definition of the institution by giving a proof sketch of the satisfaction condition. The complete proof is given in Appendix B.

Theorem 6.4.6 (CSMOF satisfaction condition)**.** *Given signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I}$, and a Σ_1-formula φ, the following satisfaction condition holds.*

$$\mathcal{I}|_\sigma \models_{\Sigma_1} \varphi \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\varphi)$$

Proof sketch. We proceed in the same way as we did for proving the three cases of a formula φ. The point is that the semantic elements giving an interpretation of a property do not change with the reduct. This means that it is the same to evaluate $(c_i \bullet r_j)$ in a reduced interpretation $\mathcal{I}|_\sigma$ as evaluating its translation $(\sigma_T(c_i) \bullet \sigma_R(r_j))$ in the original interpretation $\mathcal{I}$. Thus, the number of elements of $|S|$ (as in Definition 6.4.1) is the same in both sides of the satisfaction condition. Finally the satisfaction condition holds.

□

Given that the satisfaction condition holds we can state that $\mathcal{I}^{\mathrm{M}}$ consisting of signatures, morphisms, formulas, interpretations, reducts, and the satisfaction relation, defines an institution.

6.5 Related Work

There are many works defining the semantics of MOF and the conformance relation in terms of a shallow embedding of the language by providing a syntactic translation into another one, e.g. rewriting logic [BM09, RDV09], constructive type theory [CLST10b], first-order logic [BEC12, SZ09]. Unlike these works, we prefer to define a generic institution not restricted by any logical domain.

There are also some works with an algebraic/institutional approach. In [OW09] the authors propose an algebraic representation of metamodels based on many-sorted algebras. They broadly define how these settings are related to institutions but they do not define a specific

institution for metamodels. Moreover, in [BKMW08] the authors propose to define concrete institutions for any specific metamodel involved in a transformation.

In [LBEE+06] the authors use a graph-based representation for metamodels and SW-models which is commonly used in the field of graph transformations. Graphs can also be bounded in order to represent multiplicities [WMP13]. As it is presented in Section 6.6, the structural conformance relation is given in terms of a graph morphism between an attributed graph (representing a SW-model) and a attributed type graph (representing a metamodel).

In [Zhu12] the author proposes a formal approach for the definition of metamodels (not based on MOF) using a meta-notation called GEBNF (graphic extension of BNF) and the specification of constraints on models in a formal logic language induced from GEBNF. This is also a shallow embedding into GEBNF, but in this case the author provides an institution in which signatures represent metamodels (in GEBNF), interpretations represent SW-models, and formulas are predicates using the language induced from GEBNF (intended to be as expressive as OCL).

In [Fav09] the author proposes an algebraic formalization of MOF metamodels based on the NEREUS language, which can be viewed as an intermediate notation that can be translated to other formal languages. Something similar is presented in [ABGR10] in which the authors define a method for representing MOF metamodels in a formalism called Alloy, based on first-order logic.

In [CK08, JKMR12] the authors define institutions for simple and stereotyped UML Class Diagrams. As mentioned before, we have adapted these works for the purpose of defining the institution for the conformance relation. Finally, in [TBHW99] the authors define the semantics of class diagrams with OCL constraints by defining a translation into CASL. Although this is also a shallow embedding, as explained before, the translation (and the one proposed in [JKMR12]) have some similarities with our comorphism from CSMOF to CASL, as discussed in Section 9.4.

6.6 Model Typing

Structural conformance, as defined in Section 2.1.1, involves both typing requirements and the satisfaction of multiplicity constraints. Typing requirements are not addressed by the CSMOF satisfaction relation. In fact, the CSMOF institution only considers well-typed SW-models, since any interpretation (a SW-model) is, by construction, well-typed with respect to the signature (the metamodel). No matter where, type-checking must be addressed. In this sense, for the matter of completeness of our institutional settings, we discuss how we can address type-checking.

As presented in Section 3.1, type systems derive from those for object-based languages which are reasonably well-understood [SJ07]. There are many works in this sense, but as far as we

know, no one strictly defines an institution for this purpose. However, we can take some related works for the definition of the typing problem. For example, we can consider Attributed Type Graphs with Node Type Inheritance [LBEE+06] which are commonly used in the field of graph transformation for the representation of metamodels and SW-models.

The idea is based on representing elements as graphs, i.e. structures of the form (V, E, s, t) with V a set of vertices (also called nodes), E a set of edges, and $s, t : E \to V$ the source and target functions. In the case of a SW-model, vertices represent objects and edges represent links. We can then assign each element of the graph a type by defining a type graph representing a metamodel, which is a distinguished graph containing all the relevant types (as vertices) and their properties (as edges). Graphs can be related by graph morphisms, mapping the nodes and edges of a graph to those of another one, preserving source and target of each edge. The typing itself is depicted by a graph morphism between the initial graph and the type graph. These basic settings can be improved. In particular, it is possible to define attributed graphs which are graphs with attributes associated to nodes and arrows, and attributed type graphs which are also graphs defining types with a correspondence with many possible attributed graphs. A type graph can be extended with an inheritance relation and a set of abstract node types [LBEE+06]. Graphs can also be bounded [WMP13] in order to represent multiplicities.

In conclusion, in analogy with SW-models which conform to a metamodel, we have attributed graphs typed with respect to an attributed type graph with inheritance. The typing relation within the structural conformance relation can be stated as a graph morphism between an attributed graph and an attributed type graph. In [HET08] there is an example of the formal definition of the abstract syntax of UML class and sequence diagrams based on typed attributed graph transformation with inheritance. We can assume that the typing problem is completely resolved by the existence of such graph morphism.

In order to define an institution for model typing, we need to put metamodels and SW-models at both sides of such relation, for example representing metamodels as interpretations (as in CSMOF) and SW-models as formulas. This is a similar approach to the one followed for the definition of an extension of CSMOF in Chapter 8). However, an important aspect to consider is that metamodels (interpretations) and SW-models (formulas) must not be constrained by a common set of types (within the signature as in CSMOF). In this sense, it can be no typing relation between them, which is exactly what the satisfaction relation must define.

Leaving aside the details, we can represent attributed graphs (SW-models) as formulas, and attributed type graphs with inheritance (metamodels) as interpretations. We have a fixed signature which has the sorts (e.g. for vertices and edges) and functions (e.g. relating vertices and edges) for the definition of any attributed (type) graph. A signature morphism is just a renaming of the fixed sorts and functions. Homomorphisms and reducts are pointless since the signature morphism is just a renaming. Finally, the satisfaction relation is expressed as the existence of a graph morphism between the attributed graph and the type attributed graph with inheritance.

7 An Institution for QVTR

In this chapter we introduce an institution $\mathcal{I}^{\mathrm{Q}}$ for QVT-Relations check-only unidirectional transformations (called QVTR). Since any QVT-relation transformation involves a representation of SW-models and metamodels, we base this institution on the $\mathcal{I}^{\mathrm{M}}$ institution defined in Chapter 6. The institution is an updated version of the one presented in [CS13b, CS13c].

In Section 7.1 we present some preliminary considerations with respect to the QVT standard, and we define an abstract expressions language institution $\mathcal{I}^{\mathrm{E}}$ which will be the constraint language used within model transformations. In Section 7.2 we define the syntactic aspects of the institution, i.e. the signatures and formulas. Then, in Section 7.3 we define the notion of institution model, and in Section 7.4 we define the satisfaction relation and we state the satisfaction condition of the institution. Finally, in Section 7.5 we close this chapter with a discussion of related work. Along the definition we illustrate the main concepts with the example introduced in Chapter 2. For the sake of readability we do not include complete proofs of institution properties, which are given in Appendix C.

7.1 Preliminaries

As for the institution for the conformance relation, we restrict some of the QVT-Relations constructions. As mentioned in Chapter 2, we consider only a source and a target metamodel, and the transformation is executed in the direction of the second domain. Moreover, we do not consider black-box operations or rule and transformation overriding, since they are advanced features not commonly used in practice. We neither consider auxiliary functions and queries since they are syntactic sugar. Finally, we simplify the pattern structure by not considering opposite roles in object templates, since they can be expressed as conditions within a template, and collection templates, since they are advanced features not commonly used in practice. The future inclusion of these elements will improve the institution.

7.1.1 Recursive Model Transformations

We forbid cycles of rule invocations to avoid infinite recursion. Notice that `when` and `where` clauses, as defined in Section 2.1.2, conform a potentially cyclic graph of dependencies between transformation rules. Cycles are however not problematic unless the satisfaction of a rule involving a set of SW-model elements depends recursively on its own satisfaction. In this case we have infinite recursion which cannot be handled by our proposal. We thus assume that recursion is well-founded, i.e. no rule will be called twice in the same chain of dependencies for the same set of elements. This constraint ensures well-foundness since we always have a finite set of element in any SW-model. Another alternative evaluated in [GdL12] is to forbid cycles of dependencies to avoid infinite recursion. However, this alternative is too restrictive in practice.

7.1.2 Expressions Language

As mentioned in Chapter 5, the `when` and `where` clauses, as well as the `<predicate>` of a pattern, may contain arbitrary boolean OCL expressions. From a formal perspective we need an institution for OCL which would allow us to use the language not only for constraining the transformation rules, but also for expressing general constraints on metamodels. Unfortunately there is no institution for OCL, which is left for future work. However, in our work we consider a generic institution $\mathcal{I}^{\mathrm{E}}$ as an expressions language, which can be instantiated for example with an institution for first-order logic with equality ($FOL^{=}$) as defined in [ST12, LR12]. With this decision we are not losing expressive power (there are works [BKS02] with the aim of expressing OCL into first-order logic).

In $FOL^{=}$, signatures are many-sorted algebraic signatures enriched with predicate symbols of the form (S, Ω, Π) where S is a set (of sort names), $\Omega = (\Omega_{w,s})_{w \in S^*, s \in S}$ is a family of sets (of operation names with their arities and result sorts indicated just as in algebraic signatures) and $\Pi = (\Pi_w)_{w \in S^*}$ is a family of sets (of predicate or relation names with their arities indicated). Signature morphisms are as usual between elements in the signatures.

Moreover, sentences are first-order formulas built out from atomic formulas using the standard propositional connectives ($\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$,$\neg$) and quantifiers ($\forall$, $\exists$). The atomic formulas are equalities of the form $t = t'$, where t and t' are (S,Ω)-terms (possibly with variables) of the same sort, atomic predicate formulas of the form $p(t_1, ..., t_n)$, where $p \in \Pi_{s_1 ... s_n}$ and $t_1, ..., t_n$ are terms of sorts $s_1, ..., s_n$, respectively, and the logical constants `true` and `false`.

Models are many-sorted first-order structures, i.e. consisting of a carrier set $|D|_s$ for each sort name $s \in S$, a function f_D for each operation name $f \in \Omega$, and a relation p_D for each $p \in \Pi$. Finally, the satisfaction relation is the usual satisfaction of a first-order sentence in a first-order structure. The formulas can also include variables $X^s = (X^s)_{s \in S}$, so for the satisfaction relation we consider variable assignments $\mu_s : X^s \to |D|_s$ for each $s \in S$.

Specific signatures and models of the $\mathcal{I}^{\mathrm{E}}$ institution can be derived from (and constrained by) signatures and models of the $\mathcal{I}^{\mathrm{Q}}$ institution. For example, in the case of $FOL^{=}$, the signature must contain sorts for every type, predicates for each property declaration, and predefined functions for primitive types and type constructors (as those defined for the OCL). Moreover, we can derive a $FOL^{=}$ first-order structure such that the interpretation of elements must be the same than in each of the models $\mathcal{M}_i^{\mathrm{M}}$ of the $\mathcal{I}^{\mathrm{M}}$ institution. These aspects are more clear in the examples of the next sections.

7.2 Signatures and Formulas

A signature defines the source and target metamodels that are involved in a specific model transformation.

Definition 7.2.1 (QVTR **signature**)
A QVTR signature is a pair $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ of $\mathcal{I}^{\mathrm{M}}$-signatures $\Sigma_i^{\mathrm{M}} = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1, 2)$ representing the source and target metamodels of the transformation.

We can either assume that there are no name clashes (types, roles and properties) between $\mathcal{I}^{\mathrm{M}}$-signatures or that equal names do not introduce an inconsistency.

From this signature, we can derive a $\mathcal{I}^{\mathrm{E}}$-signature of the expressions language which must contain an element for each type (in $\bigcup_i T(C_i)$) and a predicate for each property declaration (in $\bigcup_i P_i$) in the $\mathcal{I}^{\mathrm{M}}$-signatures, as well as predefined predicates and functions for type constants and type constructors.

Example 7.2.2
The signature $\Sigma = \langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ contains the signature Σ_1^{M} of the source metamodel, which is presented in Example 6.2.3, and the signature Σ_2^{M} of the target metamodel, which is not shown here but can be derived in the same way as the other one.

The $\mathcal{I}^{\mathrm{E}}$-signature Σ^{E} is defined as the disjoint union of both signatures, plus other predicates and functions, as for example the type constructor $+$ (append) for strings. In case of using a $FOL^{=}$ signature, there must also be sorts for every type ($\bigcup_i T(C_i) \subseteq S$) and there must be a predicate for each property declaration ($\bigcup_i P_i \subseteq \Pi$).

□

Formulas represent the two basic conditions which must hold in a model transformation: keys defined on source and target metamodel elements and transformation rules stating relations between source and target elements.

Definition 7.2.3 (QVTR **formula**)
Given a signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ such that $\Sigma_i^{\mathrm{M}} = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$, Σ-formulas are defined as follows:

- A formula φ^{K} representing a key constraint of the form $\langle c, \{r_1, ..., r_n\} \rangle$ $(1 \leq n)$ with $c \in C_i$ (j = 1..n) a class in one of the metamodels, $r_j \in R_i$ (j = 1..n) roles defined in properties in which such class participates (having such role or at the opposite side of it), i.e. for each r_j there is a property $\langle r_j : c_j, r_i : c_i \rangle$ or $\langle r_i : c_i, r_j : c_j \rangle \in P_i$ such that $c = c_i$ (the property is non-navigable from c) or $c = c_j$ (r_j is navigable from c). Roles determine the elements within these properties that together can uniquely identify an instance of the class.

- A formula φ^{R} representing a set of interrelated transformation rules, such that, given variables $X^s = (X^s)_{s \in (\bigcup_i T(C_i))}$, the formula is a finite set of tuples representing rules of the form $\langle \mathrm{top}, \mathrm{VarSet}, \mathrm{ParSet}, \mathrm{Pattern}_i\ (i = 1, 2), \mathrm{when}, \mathrm{where} \rangle$, where:

  - $\mathrm{top} \in \{true, false\}$ defines if the rule is a top-level relation or not

  - $\mathrm{VarSet} \subseteq X^s$ is the set of variables used within the rule

  - $\mathrm{ParSet} \subseteq \mathrm{VarSet}$ representing the set of variables taken as parameters when the rule is called from another one (corresponding to the top pattern element in the source and target domains, and the primitive domains defined within the rule)

  - $\mathrm{Pattern}_i$ $(i = 1, 2)$ are the source and target patterns, i.e. tuples $\langle E_i, A_i, Pr_i \rangle$ such that $E_i \subseteq (X^c)_{c \in C_i}$ is a set of class-indexed variables, A_i is a set of elements representing associations of the form $rel(p, x, y)$ with $p \in P_i$ and $x, y \in E_i$, and Pr_i is a $\mathcal{I}^{\mathrm{E}}$-formula over these elements. We denote by $\mathrm{k\_VarSet}$ $(\mathrm{k} = \{1, 2\})$ the variables used in pattern k that do neither occur in the other domain nor in the `when` clause.

  - $\mathrm{when}/\mathrm{where}$ are the `when`/`where` clauses of the rule, respectively. A `when` clause is a pair $\langle \mathrm{when}_c, \mathrm{when}_r \rangle$ such that when_c is a $\mathcal{I}^{\mathrm{E}}$-formula with variables in VarSet, and when_r is a set of pairs of transformation rules (formulas) and set of variables which are the parameters of the rules. We will denote by $\mathrm{WhenVarSet}$ the set of variables occurring in the `when` clause. Finally, a `where` clause is a pair $\langle \mathrm{where}_c, \mathrm{where}_r \rangle$ such that where_c is a $\mathcal{I}^{\mathrm{E}}$-formula with variables in VarSet, and where_r is a set of pairs of transformation rules and set of variables (as before). Only variables used in a `where` clause (as `prefix` in the example) are contained in $\mathrm{2\_VarSet}$.

Definition 7.2.4 (QVTR **signature morphism**)
Given a signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$, a signature morphism is defined as a tuple of signature morphisms of the corresponding institutions $\langle \sigma_1^{\mathrm{M}}, \sigma_2^{\mathrm{M}} \rangle$. The signature morphism σ^{E} is derived from the morphisms defined for types and predicates in σ_1^{M} and σ_2^{M}

Given a set of variables $X_2 = (X_2^s)_{s \in (\bigcup_i T_2(C_i))}$, we define a set $X_2|_\sigma$ as

$X_1 = (X_1^{s_1})_{s_1 \in (\bigcup_i T_1(C_i))}$ by $X_1^{s_1} = X_2^{\sigma(s_1)}$. Signature morphisms extend to formulas over Σ_1 and $X_2\,|_\sigma$ as follows. Given a Σ_1-formula φ, $\sigma(\varphi)$ is the canonical application of the signature morphism to every element in φ.

As shown in the following lemmas, we can prove that signatures and signature morphisms define a category **Sign**, and that there is a functor *Sen* giving a set of formulas for each signature and a function translating sentences for each signature morphism. We provide a sketch of the proofs. The complete proofs are given in Appendix C.

Lemma 7.2.5. *Signatures and signature morphisms define a category* ***Sign****. The points of the category are the signatures and it arrows are the signature morphisms.*

Proof sketch. A signature morphism is defined as a tuple of morphisms of the corresponding institutions. We can define the composition as the componentwise composition of signature morphisms, and the identity signature morphism as the tuple with the identity signature morphisms of the corresponding institutions. Since in those institutions, morphisms are composable, the composition is associative, and there exists an identity signature morphism, we can conclude that in our signature morphism those properties also hold. Finally, signatures and signature morphisms define a category.

□

Lemma 7.2.6. *There is a functor Sen giving a set of formulas ψ (object in the category* ***Set****) for each signature Σ (object in the category* ***Sign****), as shown in the definition of a formula, and a function $\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$ (arrow in the category* ***Set****) translating formulas for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category* ***Sign****), as shown in the extension of the signature morphism to formulas.*

Proof sketch. A signature morphism is, by Definition 7.2.4, a pair of CSMOF signature morphisms changing types and roles consistently. In this sense, its application to any formula in $Sen(\Sigma_1)$ gives a formula in $Sen(\Sigma_2)$ with the types and roles translated with respect to the signature morphism $\sigma : \Sigma_1 \to \Sigma_2$. In this sense, the domain and codomain of the image of an arrow in **Set** are the image of domain and codomain, respectively, of the arrow in **Sign**. Moreover, since signature morphism can be composed (as defined in Lemma 7.2.5), the composition with respect to formulas is also preserved. For the same reason, identities are preserved. Finally, the functor *Sen* is defined.

□

Example 7.2.7

Key definitions within the example are represented by the following formulas:

- $\langle \text{Table}, \{\text{name}, \text{schema}\}\rangle$
- $\langle \text{Column}, \{\text{name}, \text{owner}\}\rangle$
- $\langle \text{Key}, \{\text{name}, \text{owner}\}\rangle$

Property $\langle \text{owner} : \text{Table}, \text{column} : \text{Column}\rangle$ is bidirectional. However, if owner role is non-navigable (i.e. $\langle \_ : \text{Table}, \text{column} : \text{Column}\rangle$), the second key can be represented using the opposite role column as follows:

- $\langle \text{Column}, \{\text{name}, \text{column}\}\rangle$

There is also a formula representing the whole transformation with the following rules.

`PackageToSchema` = $\langle \text{top}, \text{VarSet}, \text{ParSet}, \text{Pattern}_i\ (i = 1, 2), \text{when}, \text{where}\rangle$

- top = *true*
- VarSet = $\{\text{pn}, \text{p}, \text{s}\}$ with $\text{pn} \in X^{String}$, $\text{p} \in X^{Package}$, and $\text{s} \in X^{Schema}$
- ParSet = $\{\text{p}, \text{s}\}$
- $\text{Pattern}_1 = \langle E_1, A_1, Pr_1\rangle$ with $E_1 = \{\text{p}\}$, $A_1 = \emptyset$, and $Pr_1 =$ `name(p,pn)` Remember that $\text{name}(\text{p}, \text{pn})$ is a property in the source metamodel, and thus a predicate in the $\mathcal{I}^{\text{E}}$ signature
- $\text{Pattern}_2 = \langle E_2, A_2, Pr_2\rangle$ with $E_2 = \{\text{s}\}$, $A_2 = \emptyset$, and $Pr_2 =$ `name(s,pn)`
- when $= \langle \emptyset, \emptyset\rangle$
- where $= \langle \emptyset, \emptyset\rangle$

`ClassToTable` = $\langle \text{top}, \text{VarSet}, \text{ParSet}, \text{Pattern}_i\ (i = 1, 2), \text{when}, \text{where}\rangle$

- top = *true*
- VarSet = $\{\text{cn}, \text{prefix}, \text{c}, \text{p}, \text{Persistent}, \text{t}, \text{s}, \text{cl}, \text{NUMBER}, \text{TID}, \text{k}, \text{PK}\}$ with $\text{c} \in X^{Class}$, $\text{p} \in X^{Package}$, $\text{t} \in X^{Table}$, $\text{s} \in X^{Schema}$, $\text{cl} \in X^{Column}$, $\text{k} \in X^{Key}$, and the others in X^{String}
- ParSet = $\{\text{c}, \text{t}\}$
- $\text{Pattern}_1 = \langle E_1, A_1, Pr_1\rangle$ with
  E_1 = $\{\text{c}, \text{p}\}$
  A_1 = $\{rel(\langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier}\rangle, \text{p}, \text{c})\}$
  Pr_1 = `name(c,cn) AND kind(c,Persistent)`

- $\text{Pattern}_2 = \langle E_2, A_2, Pr_2 \rangle$ with
  E_2 = {t, s, cl, k}
  A_2 = {$rel(\langle \text{schema} : \text{Schema}, \text{tables} : \text{Table} \rangle, \text{s}, \text{t})$,
  $rel(\langle \text{owner} : \text{Table}, \text{column} : \text{Column} \rangle, \text{t}, \text{cl})$,
  $rel(\langle \text{column} : \text{Column}, \text{key} : \text{Key} \rangle, \text{cl}, \text{k})$,
  $rel(\langle \text{owner} : \text{Table}, \text{key} : \text{Key} \rangle, \text{t}, \text{k})$}
  Pr_2 = `name(t,cn) AND name(cl,TID) AND type(cl,NUMBER) AND name(k,PK)`

- when = $\langle \emptyset, \{(\texttt{PackageToSchema}, \{p, s\})\} \rangle$

- where = $\langle \text{where}_c, \{(\texttt{AttributeToColumn}, \{c, t, prefix\})\} \rangle$
  with where_c = `prefix = EMPTY`

`AttributeToColumn` = $\langle$top, VarSet, ParSet, Pattern_i $(i = 1, 2)$, when, where$\rangle$

- top = $false$

- VarSet = {an, pn, cn, sqltype, c, a, p, t, cl, prefix, EMPTY, INTEGER, NUMBER, BOOLEAN, VARCHAR} with c $\in X^{Class}$, a $\in X^{Attribute}$, p $\in X^{PrimitiveDataType}$, t $\in X^{Table}$, cl $\in X^{Column}$, and the others in X^{String}

- ParSet = {c, t, prefix}

- $\text{Pattern}_1 = \langle E_1, A_1, Pr_1 \rangle$ with
  E_1 = {c, a, p}
  A_1 = {$rel(\langle \text{attribute} : \text{Attribute}, \text{owner} : \text{Class} \rangle, \text{a}, \text{c})$,
  $rel(\langle \_ : \text{Attribute}, \text{type} : \text{PrimitiveDataType} \rangle, \text{a}, \text{p})$}
  Pr_1 = `name(a,an) AND name(p,pn)`

- $\text{Pattern}_2 = \langle E_2, A_2, Pr_2 \rangle$ with
  E_2 = {t, cl}
  A_2 = {$rel(\langle \text{owner} : \text{Table}, \text{column} : \text{Column} \rangle, \text{t}, \text{cl})$}
  Pr_2 = `name(cl,cn) AND type(cl,sqltype)`

- when = $\langle \emptyset, \emptyset \rangle$.

- where = $\langle \text{where}_c, \emptyset \rangle$ with where_c =

```
((prefix = EMPTY AND cn = an) OR
   (not (prefix = EMPTY) AND (cn = prefix + an)))
AND ((pn = INTEGER AND sqltype = NUMBER) OR
      (pn = BOOLEAN AND sqltype = BOOLEAN) OR
        (((not (pn = INTEGER) AND (not (pn = BOOLEAN))
           AND sqltype = VARCHAR))
```

□

7.3 Models

An interpretation contains a semantic representation for the source and target SW-models.

Definition 7.3.1 (QVTR **interpretation**)
Given a signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$, an interpretation is a tuple $\langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$ of disjoint $\mathbf{Sign}_i^{\mathrm{M}}$-interpretations. Here we can also derive a $\mathcal{I}^{\mathrm{E}}$ model $\mathcal{M}^{\mathrm{E}}$ such that the interpretation of elements in $\mathbf{Sign}_i^{\mathrm{M}}$ must be the same in $\mathcal{M}_i^{\mathrm{M}}$ and $\mathcal{M}^{\mathrm{E}}$.

Example 7.3.2
Assume that we have an interpretation $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$ such that $\mathcal{M}_1^{\mathrm{M}}$ is the one defined in Example 6.3.3, and $\mathcal{M}_2^{\mathrm{M}}$ is an interpretation with a direct correspondence with the SW-model in the bottom of Figure 2.7. The interpretation of elements in $\mathbf{Sign}_i^{\mathrm{M}}$ must be the same in $\mathcal{M}_i^{\mathrm{M}}$ and $\mathcal{M}^{\mathrm{E}}$. If the model is a $FOL^{=}$ structure, this means that $|D|_t = V_t$. for every $t \in \bigcup_i T(C_i)$, and $p_D = p^{\mathcal{I}}$ for every $p \in \bigcup_i P_i$. In the case of $t \in T(C) \backslash C$ (primitive types) we have that $V_t \subseteq |D|_t$ since the model can have more elements than those in the source and target institutions, as type constants (e.g. the empty string) and elements created using type constructors from other elements (e.g. new strings using type constructor $+$).

□

Definition 7.3.3 (**Binding of variables**)
Given a signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ such that $\Sigma_i^{\mathrm{M}} = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$, an interpretation $\langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$, and variables $X^s = (X^s)_{s \in (\bigcup_i T(C_i))}$, the binding of a variable $x^c \in X^c$, denoted by $|x^c|$, is the set of any possible interpretation of such variable, which corresponds to $|x^c| = V_c$ if c is a class, or corresponds to this set together with the elements created using type constructors (in $\mathcal{M}^{\mathrm{E}}$) in the case of c is a primitive type.

Moreover, the binding of a set of variables $(x_1, ..., x_n)$, denoted by $|(x_1, ..., x_n)|$, is defined as $\{(y_1, ..., y_n) \mid y_i \in |x_i| \ (i = 1..n)\}$. We can also view $|(x_1, ..., x_n)|$ as a set of variable assignments. We denote by $\mu[x_1, ..., x_n]$ a function with an assignment for variables $x_1, ..., x_n$. We also denote by $\mu_1 \cup \mu_2$ an assignment unifying the former ones, assuming that if there is variable clash, the assignment takes for those variables the values in μ_2.

Example 7.3.4
Binding of variables depends on the type of elements. For a class variable, we have that the set of possible values coincides with the set of elements within the CSMOF institutions. For example, we have that $|\mathrm{p}| = V_{\mathrm{Package}} = \{p1\}$. However, if the variable is of a primitive type, since transformation rules can use other elements beside those in the CSMOF institutions (for example those strings created using the type constructor $+$), we can have more elements. In the example, we have that $|\mathrm{pn}| = \{Pac, Str, ID, ..., tid, numb, ..., ID{+}tid, ID{+}numb, ...\}$.

□

Definition 7.3.5 (QVTR **homomorphism and reduct**)
Given signatures $\Sigma_i = \langle \Sigma_1^{\mathrm{M}}{}_i, \Sigma_2^{\mathrm{M}}{}_i \rangle$ $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, and Σ_2-interpretation $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$ and $\mathcal{M}_2 = \langle \mathcal{M}_1^{\mathrm{M}}{}_2, \mathcal{M}_2^{\mathrm{M}}{}_2 \rangle$, homomorphisms and reducts are defined componentwise. A Σ_2-homomorphism $h : \mathcal{M} \to \mathcal{M}_2$ is defined as a tuple of homomorphisms $\langle h_1^{\mathrm{M}}, h_2^{\mathrm{M}} \rangle$ of the corresponding institutions. The reduct $\mathcal{M}|_\sigma$ of $\mathcal{M}$ along σ is the Σ_1-interpretation $\langle \mathcal{M}_1^{\mathrm{M}}|_\sigma, \mathcal{M}_2^{\mathrm{M}}|_\sigma \rangle$. Moreover, the reduct $h|_\sigma$ of h along σ is the Σ_1-homomorphism $\langle h_1^{\mathrm{M}}|_\sigma, h_2^{\mathrm{M}}|_\sigma \rangle$.

The following lemmas state that interpretations and homomorphisms define a category, and also that the reduct defines a functor. Thus, there is a functor **Mod** giving a category of interpretations and the reduct functor for each signature. Again here we just provide a sketch of the proofs. The complete proofs are given in Appendix C.

Lemma 7.3.6. *For any signatures, the Σ-interpretations and Σ-homomorphisms define a category **Mod**(Σ). The points of the category are the Σ-interpretations, and its arrows are the Σ-homomorphisms.*

Proof sketch. An interpretations is a tuple of interpretations of the corresponding institutions, and homomorphisms are defined componentwise. We can define the composition of homomorphisms as the componentwise composition of homomorphisms, as well as the identity homomorphism as the tuple with the identity homomorphisms of the corresponding institutions. Since in the corresponding institutions, homomorphisms are composable, the composition is associative, and there exists an identity homomorphism, we can conclude that these properties also hold for homomorphisms. Finally, interpretations and homomorphisms define a category.

□

Lemma 7.3.7. *The reduct of Σ-interpretations and Σ-homomorphisms is a functor **Mod**(σ) from Σ_2-interpretations to Σ_1-interpretations (and Σ_2-homomorphisms to Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$.*

Proof sketch. An interpretations is a tuple of interpretations of the corresponding institutions, and homomorphisms are defined componentwise. In the corresponding institutions, the reduct of interpretations and homomorphisms is a functor. From this we can conclude straightforwardly that domain and codomain of the reduct of an Σ-homomorphism are the reduct of domain and codomain, respectively, the reduct of a composition of two homomorphisms is the composition of the reducts of those homomorphisms, and the reduct of an identity homomorphisms is likewise an identity. Finally, the reduct of interpretations and homomorphisms is a functor.

□

Lemma 7.3.8. *There is a functor **Mod** giving a category **Mod**(Σ) of Σ-interpretations (object in the category **Cat**) for each signature Σ (object in the category **Sign**), as shown in Lemma 7.3.6, and a functor **Mod**(σ) (arrow in the category **Cat**) from Σ_2-interpretations*

*to Σ_1-interpretations (and Σ_2-homomorphisms to Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**), as shown in Lemma 7.3.7.*

Proof sketch. By Lemma 7.3.7, the image of an arrow $\sigma : \Sigma_2 \to \Sigma_1$ in the category $\mathbf{Sign}^{op}$ is an arrow $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ in the category **Cat**. Also, by Lemma 7.3.6, the image of a signature Σ in the category **Sign** is an object $\mathbf{Mod}(\Sigma)$ in the category **Cat**. Thus, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow. Now, reducts (and homomorphisms) are defined as tuples of reducts (and homomorphisms) of the corresponding institutions. Since composition is preserved in isolation for each component, we can directly conclude that this also holds for the tuple. Using the same reasoning, we can conclude that identities are preserved. Finally, the functor **Mod** is defined.

□

7.4 Satisfaction Relation and Satisfaction Condition

As explained in Section 5.3, the satisfaction relation must express that the target SW-model (represented within the interpretation) is the result of transforming the source SW-model (represented within the interpretation) according to the transformation rules and also that key constraints hold (both represented as formulas).

Definition 7.4.1 (QVTR **satisfaction relation**)
Given a signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ such that $\Sigma_i^{\mathrm{M}} = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$, and an interpretation $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$, we define that $\mathcal{M}$ satisfies a formula φ, written $\mathcal{M} \models_\Sigma \varphi$, in one of the following cases:

- A formula $\varphi^{\mathrm{K}} = \langle c, \{r_1, ..., r_n\} \rangle$ with $c \in C_i$ (j = 1..n), $r_j \in R_i$ (j = 1..n), is satisfied in the corresponding metamodel $\mathcal{M}_i^{\mathrm{M}}$ if there are no two elements of type c with the same set of elements related through properties involving roles r_j (they must differ in at least one element). Formally, for each r_j the corresponding property p_j is $\langle r : c, r_j : d \rangle$ if r_j is navigable, or $\langle r_j : c, \_ : d \rangle$ if the opposite role of r_j is non-navigable. We can define that given an element $x \in (V_c)_{c \in C_i}$, the set of semantic elements linked with x in p_j is $\nu(x, p_j) = \{\pi_2(t) |\ \pi_1(t) = x,\ t \in {p_j}^{\mathcal{M}}\}$. The definition is straightforward in the case of c in the second component of the property. The formula is satisfied if for all $x, y \in (V_c)_{c \in C_i}$, $x \neq y$ implies $\bigcup_j \nu(x, p_j) \neq \bigcup_j \nu(y, p_j)$.

- A formula φ^{R} is satisfied if the semantics defined in the standard [OMG09] holds, i.e. if every top-level relation holds, which means that there are matching elements in the source and target SW-models in the relation. Formally, given a $\mathcal{I}^{\mathrm{E}}$ model $\mathcal{M}^{\mathrm{E}}$ constructed from the interpretation $\mathcal{M}$, φ^{R} is satisfied if for every top rule $\mathrm{Rule} \in \varphi^{\mathrm{R}}$, we have that $\mathcal{M}^{\mathrm{E}}, \emptyset \models \mathrm{Rule}$. We use $\emptyset$ as the empty variable assignment which will be filled only in the case of explicit called rules.

A rule $\text{Rule} = \langle \text{top}, \text{VarSet}, \text{ParSet}, \text{Pattern}_i \ (i = 1,2), \text{when}, \text{where} \rangle$ is satisfied with respect to a model $\mathcal{M}^{\text{E}}$ and a variable assignment μ, denoted by $\mathcal{M}^{\text{E}}, \mu \models \text{Rule}$ if

1. If $\text{WhenVarSet} = \emptyset$

$$\begin{aligned}
&\forall\, \mu^1[x_1, ..., x_n] \in\ |\text{VarSet} \backslash \text{2_VarSet}|, \\
&\quad (\ \mathcal{M}^{\text{E}}, (\mu^1[x_1, ..., x_n] \cup \mu) \models \text{Pattern}_1 \rightarrow \\
&\qquad \exists\, \mu^2[y_1, ..., y_m] \in |\text{2_VarSet}|, \\
&\qquad\quad (\ \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu) \models \text{Pattern}_2 \wedge \\
&\qquad\qquad \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu) \models \text{where}))
\end{aligned}$$

2. If $\text{WhenVarSet} \neq \emptyset$

$$\begin{aligned}
&\forall\, \mu^w[z_1, ..., z_o] \in\ |\text{WhenVarSet}|, \\
&\quad (\ \mathcal{M}^{\text{E}}, (\mu^w[z_1, ..., z_o] \cup \mu) \models \text{when} \rightarrow \\
&\qquad \forall\, \mu^1[x_1, ..., x_n] \in\ |\text{VarSet} \backslash (\text{WhenVarSet} \cup \text{2_VarSet})|, \\
&\qquad\quad (\ \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^w \cup \mu) \models \text{Pattern}_1 \rightarrow \\
&\qquad\qquad \exists\, \mu^2[y_1, ..., y_m] \in |\text{2_VarSet}|, \\
&\qquad\qquad\quad (\ \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu^w \cup \mu) \models \text{Pattern}_2 \wedge \\
&\qquad\qquad\qquad \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu^w \cup \mu) \models \text{where})))
\end{aligned}$$

A pattern $\text{Pattern} = \langle E, A, Pr \rangle$ is satisfied with respect to a model $\mathcal{M}^{\text{E}}$ and a variable assignment μ (which must include a valuation for elements in E), denoted by $\mathcal{M}^{\text{E}}, \mu \models \text{Pattern}$ if there is a matching subgraph of elements and the predicate holds, i.e.

- $\forall\ rel(\langle r_1 : c_1, r_2 : c_2 \rangle, x, y) \in A.\ (\mu(x), \mu(y)) \in\ \mathcal{M}^{\text{E}}$ (this means that the model $\mathcal{M}^{\text{E}}$ has a relation (corresponding to the property $\langle r_1 : c_1, r_2 : c_2 \rangle$) connecting elements $\mu(x)$ and $\mu(y)$)
- $\mathcal{M}^{\text{E}}, \mu \models_{\text{E}} Pr$, such that $\models_{\text{E}}$ is the satisfaction relation in $\mathcal{I}^{\text{E}}$

A `when` clause $\langle \text{when}_c, \text{when}_r \rangle$ is satisfied with respect to a model $\mathcal{M}^{\text{E}}$ and a variable assignment μ, denoted by $\mathcal{M}^{\text{E}}, \mu \models \langle \text{when}_c, \text{when}_r \rangle$ if

$$\mathcal{M}^{\text{E}}, \mu \models_{\text{E}} \text{when}_c\ \wedge\ (\forall (r, v) \in \text{when}_r.\ \mathcal{M}^{\text{E}}, \mu[v] \models r)$$

such that $\models_{\text{E}}$ is the satisfaction relation in $\mathcal{I}^{\text{E}}$, and the later is the satisfaction of the parametric transformation rule r (which must be defined in the formula φ^{R}) using the variable assignment $\mu[v]$ as a parameter. The satisfaction of a `where` clause is defined in the same way.

Example 7.4.2

We need to prove that our interpretation satisfies both kinds of formulas.

In the case of keys $\langle \text{Table}, \{\text{name}, \text{schema}\}\rangle$ and $\langle \text{Key}, \{\text{name}, \text{owner}\}\rangle$, we have only one table t and only one key k, thus the condition trivially holds.

In the case of key $\langle \text{Column}, \{\text{name}, \text{owner}\}\rangle$ we have two columns c_1 and c_2 and their sets of elements related through properties are $\{\text{TID}, \text{t}\}$ and $\{\text{value}, \text{t}\}$, correspondingly. The sets differ in one element, and thus the formula holds.

We need to prove now that $\mathcal{M}^{\text{E}}, \emptyset \models$ `PackageToSchema`.
We know that $|\text{pn}| = \{Pac, Str, ID, Per, val, nul, pk, tid, numb, varch, ...\}$, and $|\text{p}| = V_{\text{Package}} = \{p1\}$, thus $|\{\text{pn}, \text{p}\}|$ is $\{(Pac, p1), (Str, p1), (ID, p1), ...\}$. We also have that $|\text{s}| = V_{\text{Schema}} = \{s1\}$. Thus, $\mathcal{M}^{\text{E}}, \emptyset \models$ `PackageToSchema` if

$$
\begin{aligned}
&\forall\, \mu^1[\text{pn}, \text{p}] \in \{(Pac, p1), (Str, p1), (ID, p1), (Per, p1), (val, p1), (nul, p1), ...\},\\
&\quad (\, \mathcal{M}^{\text{E}}|_{\varphi}, \mu^1 \models \text{Pattern}_1 \rightarrow\\
&\qquad \exists\, \mu^2[\text{s}] \in \{s1\},\\
&\qquad\quad (\, \mathcal{M}^{\text{E}}|_{\varphi}, (\mu^1 \cup \mu^2) \models \text{Pattern}_2 \wedge\\
&\qquad\qquad \mathcal{M}^{\text{E}}|_{\varphi}, (\mu^1 \cup \mu^2) \models \text{where}))
\end{aligned}
$$

For every $\mu^1[\text{pn}, \text{p}]$ different from $(Pac, p1)$ we have that Pattern_1 does not hold, since it depends on the predicate `name(p,pn)`. Thus, in these cases the implication holds. Now, in the case of $(Pac, p1)$, we have that Pattern_1 holds, and that the only possible value for s is $s1$. In this case, we also have that $\mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2) \models \text{Pattern}_2$ since the predicate `name(s,pn)` holds. Note at the bottom of Figure 2.7 that the schema has the same name as the package, which is semantically represented as Pac. Moreover, since the `where` clause is empty, $\mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2) \models \text{where}$ trivially holds.
In conclusion, we have that $\mathcal{M}^{\text{E}}, \emptyset \models$ `PackageToSchema` indeed.

Finally, we need to prove that $\mathcal{M}^{\text{E}}, \emptyset \models$ `ClassToTable`. Proceeding in the same way, we have to prove that:

$$
\begin{aligned}
&\forall\, \mu^w[\text{p}, \text{s}] \in \{(p1, s1)\},\\
&\quad (\, \mathcal{M}^{\text{E}}, \mu^w[\text{p}, \text{s}] \models \text{when} \rightarrow\\
&\qquad \forall\, \mu^1 \in |(\text{cn}, \text{c}, \text{Persistent})|,\\
&\qquad\quad (\, \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^w) \models \text{Pattern}_1 \rightarrow\\
&\qquad\qquad \exists\, \mu^2 \in |(\text{prefix}, \text{t}, \text{cl}, \text{NUMBER}, \text{TID}, \text{k}, \text{PK})|,\\
&\qquad\qquad\quad (\, \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu^w) \models \text{Pattern}_2 \wedge\\
&\qquad\qquad\qquad \mathcal{M}^{\text{E}}, (\mu^1 \cup \mu^2 \cup \mu^w) \models \text{where})))
\end{aligned}
$$

We have a `when` clause which is the invocation of the relation `PackageToSchema` with a concrete variable assignment for domain variables p and s. We have proved above that with this assignment $\mathcal{M}^{\mathrm{E}}, \mu^w[\mathrm{p}, \mathrm{s}] \models$ `PackageToSchema` holds.
Now, in the case of $\mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^w) \models \mathrm{Pattern}_1$ we need to prove that $(\mu(p), \mu(c)) \in \mathcal{M}^{\mathrm{E}}$ since $rel(\langle \mathrm{namespace} : \mathrm{Package}, \mathrm{elements} : \mathrm{Classifier} \rangle, \mathrm{p}, \mathrm{c}) \in A$, and also that $\mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^w) \models_{\mathrm{E}}$ `name(c,cn) AND kind(c,Persistent)`. This only holds with the variable assignment $\mu^1[\mathrm{c}, \mathrm{cn}, \mathrm{Persistent}] = (c1, ID, Per)$ and $\mu^w[\mathrm{p}] = p1$. In any other case, $\mathrm{Pattern}_1$ does not hold and thus the rest of the expression holds.

Now, for proving $\mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^2) \models \mathrm{Pattern}_2$ we need to prove that

- $(\mu(\mathrm{s}), \mu(\mathrm{t})) \in \mathcal{M}^{\mathrm{E}}$ since $rel(\langle \mathrm{schema} : \mathrm{Schema}, \mathrm{tables} : \mathrm{Table} \rangle, \mathrm{s}, \mathrm{t}) \in A$
- $(\mu(\mathrm{t}), \mu(\mathrm{cl})) \in \mathcal{M}^{\mathrm{E}}$ since $rel(\langle \mathrm{owner} : \mathrm{Table}, \mathrm{column} : \mathrm{Column} \rangle, \mathrm{t}, \mathrm{cl}) \in A$
- $(\mu(\mathrm{cl}), \mu(\mathrm{k})) \in \mathcal{M}^{\mathrm{E}}$ since $rel(\langle \mathrm{column} : \mathrm{Column}, \mathrm{key} : \mathrm{Key} \rangle, \mathrm{cl}, \mathrm{k}) \in A$
- $(\mu(\mathrm{t}), \mu(\mathrm{k})) \in \mathcal{M}^{\mathrm{E}}$ since $rel(\langle \mathrm{owner} : \mathrm{Table}, \mathrm{key} : \mathrm{Key} \rangle, \mathrm{t}, \mathrm{k}) \in A$

and also that $\mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^2) \models_{\mathrm{E}} Pr_2$. These expressions hold with $\mu^2[\mathrm{prefix}, \mathrm{t}, \mathrm{cl}, \mathrm{NUMBER}, \mathrm{TID}, \mathrm{k}, \mathrm{PK}] = (nul, t1, cl1, num, TID, k1, PK)$.
Finally, with the variable assignment we have at the moment $(\mu^1 \cup \mu^2 \cup \mu^w)[...] = (ID, c1, p1, Per, nul, t1, s1, cl1, num, TID, k1, PK)$, we can prove $\mathcal{M}^{\mathrm{E}}, \mu[c, t, prefix] \models$ `AttributeToColumn`.

As before, we have to prove that:

$$
\begin{aligned}
&\forall\, \mu^1[...] \in |\mathrm{c}, \mathrm{a}, \mathrm{p}, \mathrm{an}, \mathrm{pn}|, \\
&\quad (\, \mathcal{M}^{\mathrm{E}}, (\mu^1[\mathrm{c}, \mathrm{a}, \mathrm{p}, \mathrm{an}, \mathrm{pn}] \cup \mu[\mathrm{c}, \mathrm{t}, \mathrm{prefix}]) \models \mathrm{Pattern}_1 \rightarrow \\
&\qquad \exists\, \mu^2[...] \in |\mathrm{cn}, \mathrm{sqltype}, \mathrm{t}, \mathrm{cl}, \mathrm{prefix}, \mathrm{EMPTY}, \mathrm{INTEGER}, \\
&\qquad\qquad \mathrm{NUMBER}, \mathrm{BOOLEAN}, \mathrm{VARCHAR}, \mathrm{prefix} + \mathrm{an}|, \\
&\qquad\quad (\, \mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^2 \cup \mu) \models \mathrm{Pattern}_2 \wedge \\
&\qquad\qquad \mathcal{M}^{\mathrm{E}}, (\mu^1 \cup \mu^2 \cup \mu) \models \mathrm{where}))
\end{aligned}
$$

For every $\mu^1[\mathrm{c}, \mathrm{a}, \mathrm{p}, \mathrm{an}, \mathrm{pn}]$ different from $(c1, a1, pdt1, val, Str)$ we have that $\mathrm{Pattern}_1$ does not hold, since it depends on the predicate `name(a,an) AND name(p,pn)`. Thus, in these cases the rest of the expression holds. Now, in the case of $(c1, a1, pdt1, val, Str)$, we have that $\mathrm{Pattern}_1$ holds.

In this case, there exists a variable assignment $\mu^2[...] = (val, varch, t1, cl2, nul, nul, int, numb, BOOL, VARC, val)$ such that $\mathcal{M}^{\mathrm{E}}|_{\varphi}, (\mu^1 \cup \mu^2 \cup \mu) \models \mathrm{Pattern}_2$. This can be viewed at the bottom of Figure 2.7, where the table t (semantically represented as $t1$) has a column c2 (semantically represented as $cl2$) with column name value (semantically represented as val) and type VARCHAR (semantically represented as $VARC$) which satisfies the predicate

`name(cl,cn) AND type(cl,sqltype).`
The same variable assignment satisfies the `where` clause since

```
(prefix = EMPTY) AND (cn = an) AND
    (not (pn = INTEGER) AND (not (pn = BOOLEAN)))
              AND (sqltype = VARCHAR)
```

□

We close the definition of the institution by giving a proof sketch of the satisfaction condition. The complete proof is given in Appendix C.

Theorem 7.4.3 (QVTR satisfaction condition)**.** *Given signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$, a set of variables $X_2 = (X_2^s)_{s \in S_2}$, and a Σ_1-formula φ with variables in $X_2|_\sigma$, the following satisfaction condition holds.*

$$\mathcal{M}|_\sigma \models_{\Sigma_1} \varphi \text{ iff } \mathcal{M} \models_{\Sigma_2} \sigma(\varphi)$$

Proof sketch. In the case of a formula $\varphi^{\mathrm{K}} = \langle c, \{r_1, ..., r_n\} \rangle$ with $c \in C_i$ (j = 1..n), $r_j \in R_i$ (j = 1..n), it is satisfied in the corresponding interpretation $\mathcal{M}_i^{\mathrm{M}}|_\sigma$ if there are no two elements of type c with the same set of elements related through properties involving roles r_j (they must differ in at least one element). Since the signature morphism changes types and roles consistently, the interpretation $\mathcal{M}_i^{\mathrm{M}}$ will not have more or less elements of type $\sigma_{T_i}(c)$, or less or more elements related through properties involving roles $\sigma_{R_i}(r_j)$. Thus, the translated formula is also satisfied. This implication also holds backwards. In the case of a formula φ^{R}, the reasoning is similar. The formula holds if every top-level relation holds, which means that there are matching elements in the source and target interpretations within $\mathcal{M}|_\sigma$ satisfying the relations. After the signature morphism, we can find the same matching elements in the source and target interpretations within $\mathcal{M}$, and since the signature morphism can only introduce new types and roles not used within $\sigma(\varphi^{\mathrm{R}})$, the relations between matching elements still hold. Once again, this implication also holds backwards. Finally, the satisfaction condition holds.

□

Given that the satisfaction condition holds we can state that $\mathcal{I}^{\mathrm{Q}}$ consisting of signatures, morphisms, formulas, interpretation, reducts, and the satisfaction relation, defines an institution.

7.5 Related Work

There are works defining the semantics of QVT-Relations in terms of a shallow embedding of the language, e.g. into rewriting logic [BHM09] and coloured petri nets [dLG09]. There are also embeddings into specific tools, as in the case of Alloy [ABK07] and KIV [SMR11], which provide model checking capabilities. In some cases, the tool performs a translation into another formal language, as in [LR12] in which the UML-RSDS language and tool performs a translation into the B or Z3 languages. As said before, we do not follow this approach since a unified mathematical formalism can be quite restrictive.

There are also some works with an algebraic/institutional approach. In [OW09] the authors define model-to-model transformations based on triple algebras (in consonance with relational model transformations). The representation differs from ours since it depends on the formal definition of a triple algebra. In [CGR12] the authors define an institution for graph transformation systems, not completely related to QVT-Relations or any other transformation language. They interpret metamodels and SW-models as graphs, as we presented in Section 6.6. Sentences are (injective) partial morphisms among typed graphs representing transformation rules.

In [BKMW08] transformations are represented as institution comorphisms between two institutions representing the source and target languages. This is related with the heterogeneous approach in [CKTW08] in which UML languages are related through comorphisms. However, this is somehow restrictive since it assumes a semantic relation between metamodels, and not every model transformation is semantic-preserving. A discussion on this topic is given in Chapter 11.

In [GdL12] the authors present a formal semantics for the QVT-Relations check-only scenario based on algebraic specification and category theory. The definition of the institution is much more complex than ours, and the work does not envisions a scenario in which the elements of the transformation are translated to other logics for verification.

Finally, in [Ste13] the authors define game-theoretic semantics of QVT-Relations check-only transformations, based on the semantics in the standard. This semantics is devised for analyzing the implications of minor variations in decisions about what the meaning of a QVT-R transformation should be.

8 Extending the Institutions

In this chapter we introduce extensions of the CSMOF institution in Chapter 6 and the QVTR institution in Chapter 7. As introduced in Section 5.4, the extension allows the inclusion of SW-models as syntactic elements (represented as a formula Ω) to be considered by any possible entailment system $\vdash$ devised to derive the satisfiability of other formulas, represented as a set of formulas Ψ, i.e. $\Omega \vdash_\Sigma \Psi$. In the same way, we also need to verify whether a key constraint (or a set of them), represented as a formula φ^{K}, is derived from the same Ω, i.e. $\Omega \vdash_\Sigma \varphi^{\mathrm{K}}$, or whether a transformation rule φ^{R} (or the whole model transformation) is derived from a pair of SW-models, i.e. $\Omega_1 \cup \Omega_2 \vdash_\Sigma \varphi^{\mathrm{R}}$. A sound entailment system will ensure semantic entailment, i.e. $\Omega \vdash_\Sigma \Psi$ implies $\Omega \models_\Sigma \Psi$. Semantic entailment is defined by the satisfaction relations of the institutions.

In Section 8.1 we define a supporting institution for the extensions of the CSMOF and QVTR institutions, which is basically the CSMOF institution but with SW-models as formulas. Then, in Section 8.2 we define such extensions for the satisfaction of our proof-theoretic needs. Finally, in Section 8.3 we discuss some decisions we made with respect to the inclusion of SW-models as syntactic elements. Along the definition we illustrate the main concepts with the example introduced in Chapter 2. For the sake of readability we do not include complete proofs of institution properties, which are given in Appendix D.

8.1 An Institution for SW-Models

As stated before, we first define a supporting institution $\mathcal{I}^{\Omega}$ by taking the same basic definitions from the CSMOF institution given in Chapter 6 and changing the definition of formulas, as well as the corresponding satisfaction relation. A formula in this institution is a syntactic representation of a SW-model.

Definition 8.1.1 ($\mathcal{I}^{\Omega}$**-formulas**)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$ as defined for the CSMOF institution, and variables $X = (X^c)_{c \in T(C)}$, we define formulas as follows:

$$\Omega \quad ::= \quad x^c \mid \langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle \mid \Omega \oplus \Omega$$

with $x^c \in X^c$, ${x_1}^{c_1} \in X^{c_1}$, ${x_2}^{c_2} \in X^{c_2}$, $c, c_1, c_2 \in T(C)$, $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, $r_1, r_2 \in R$.

A variable x^c represents a typed element, $\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle$ represents a link between two typed elements with their respective roles, and $\Omega \oplus \Omega$ allows to compose these elements to represent a whole SW-model.

Example 8.1.2
The formula Ω corresponding to the class SW-model in Figure 2.7 is defined as follows:

$$\begin{array}{l} \mathrm{p}^{\mathrm{Package}} \oplus \mathrm{c}^{\mathrm{Class}} \oplus \mathrm{a}^{\mathrm{Attribute}} \oplus \mathrm{pdt}^{\mathrm{PrimitiveDataType}} \oplus \\ \quad \mathrm{Package}^{\mathrm{String}} \oplus \mathrm{ID}^{\mathrm{String}} \oplus \mathrm{Persistent}^{\mathrm{String}} \oplus \ldots \oplus \\ \langle \mathrm{namespace}, \mathrm{p}, \mathrm{elements}, \mathrm{c} \rangle \oplus \langle \_ : \mathrm{p}, \mathrm{name}, \mathrm{Package} \rangle \oplus \\ \langle \mathrm{namespace}, \mathrm{p}, \mathrm{elements}, \mathrm{pdt} \rangle \oplus \langle \_ : \mathrm{a}, \mathrm{type}, \mathrm{pdt} \rangle \oplus \ldots \end{array}$$

□

The extension of a signature morphism σ to a formula Ω is still the canonical application of the signature morphism to every type and role in the formula such that $\sigma(x^c) = x^{\sigma_T(c)}$, and $\sigma(\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle) = \langle \sigma_R(r_1), {x_1}^{\sigma_T(c_1)}, \sigma_R(r_2), {x_2}^{\sigma_T(c_2)} \rangle$.

Since signature and signature morphisms are taken from the CSMOF institution, the category **Sign** is still defined, as in Lemma 6.2.7. Nevertheless, we need to prove that we still have a functor *Sen* giving a set of formulas for each signature and a function translating sentences for each signature morphism, as defined by the following lemma. The complete proof is given in Appendix D.

Lemma 8.1.3. *There is a functor Sen giving a set of formulas ψ (object in the category **Set**) for each signature Σ (object in the category **Sign**), as shown in the definition of a formula, and a function $\sigma : Sen(\Sigma_1) \rightarrow Sen(\Sigma_2)$ (arrow in the category **Set**) translating formulas for each signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ (arrow in the category **Sign**), as shown in the extension of the signature morphism to formulas.*

Proof sketch. The signature morphism in Definition 6.2.6 changes types and roles consistently. We also proved in Lemma 6.2.8 that the original functor *Sen* satisfies that domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow. Since formulas are just extended with a new component (Ω), and the extension of the signature morphism to formulas is still the canonical application of the signature morphism, the proof also holds in the case of formulas Ω. Moreover, composition and identities are preserved in the original definition of formulas and signature morphisms, based on the

fact that the extension of signature morphisms to formulas is the canonical application of the signature morphism. In this case, nothing changes, since the the extension of signature morphisms to these formulas is still the canonical application of the signature morphism. Thus, composition and identities are still preserved. Finally, the functor *Sen* is defined.

□

We need to define a reduction of an interpretation with respect to the types used in the definition of the SW-model formula Ω (as discussed in Section 8.3). This reduction defines an explicit scope in which the satisfaction of a formula is checked. We first define the types used within a formula as follows.

Definition 8.1.4 (Types of a $\mathcal{I}^{\Omega}$-formula)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, variables $(X^c)_{c \in T(C)}$, and a Σ-formula Ω representing a SW-model, the function $types$ giving the set of types used within Ω is inductively defined as follows:

- $types(x^c) = \{c\}$
- $types(\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle) = \{c_1, c_2\}$
- $types(\Omega_1 \oplus \Omega_2) = types(\Omega_1) \cup types(\Omega_2)$

Definition 8.1.5 (Explicit scope)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, a Σ-formula Ω representing a SW-model, and a Σ-interpretation $\mathcal{I} = (\boldsymbol{V_C^T}(\boldsymbol{O}), \boldsymbol{A})$, the explicit scope defined by Ω in $\mathcal{I}$, denoted by $\mathcal{I}|_\Omega$, is the Σ-interpretation $(\boldsymbol{V_C^T}(\boldsymbol{O})|_\Omega, \boldsymbol{A}|_\Omega)$ such that:

- $\boldsymbol{V_C^T}(\boldsymbol{O})|_\Omega = (V_c)_{c \in types(\Omega)}$ with $V_c \in \boldsymbol{V_C^T}(\boldsymbol{O})$
- $\boldsymbol{A}|_\Omega$ only contains those relations $\langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}} \in \boldsymbol{A}$ such that $c_1, c_2 \in types(\Omega)$

Example 8.1.6
The types used for the definition of the Σ-formula Ω in Example 8.1.2 coincide with those in the signature Σ in Example 6.2.3. For such reason, the interpretation in Example 6.3.3 cannot be reduced with respect to Ω, i.e. the explicit scope $\mathcal{I}|_\Omega = \mathcal{I}$. However, if we assume that the type extension $T(C)$ also contains type $\mathrm{Integer}$, the $\boldsymbol{T}(\boldsymbol{C})$-object domain of $\mathcal{I}$ will also have integer values, and thus, $\boldsymbol{V_C^T}(\boldsymbol{O})|_\Omega$ will not have any of them. □

We define the satisfaction relation for Ω formulas based on a valuation function $K^{\mathcal{I}}$ determining that there is an isomorphism between Ω and the interpretation $\mathcal{I}$.

Definition 8.1.7 (Valuation of a $\mathcal{I}^{\Omega}$-formula)
Given a signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$, variables $(X^c)_{c \in T(C)}$, a Σ-formula Ω representing a SW-model, and a Σ-interpretation $\mathcal{I} = (\boldsymbol{V_C^T}(\boldsymbol{O}), \boldsymbol{A})$, we define a valuation $K^{\mathcal{I}}(\Omega)$ as a bijective function mapping each $x^c \in \upsilon(\Omega)$ to an element of $(V_c)_{c \in T(C)}$ for each $c \in T(C)$, such that syntactic links in Ω and semantic links in $\mathcal{I}$ coincide, i.e.

- for every $\langle r_1, x^c, r_2, y^d \rangle \in \omega(\Omega)$,

$$(K^{\mathcal{I}}(x^c), K^{\mathcal{I}}(y^d)) \in \langle r_1 : c, r_2 : d \rangle^{\mathcal{I}} \text{ for some } \langle r_1 : c, r_2 : d \rangle \in P$$

- for every $\langle r_1 : c, r_2 : d \rangle \in P$,

$$\langle r_1 : c, r_2 : d \rangle^{\mathcal{I}} \quad = \quad \{(K^{\mathcal{I}}(x^c), K^{\mathcal{I}}(y^d)) \mid \langle r_1, x^c, r_2, y^d \rangle \in \omega(\Omega)\}$$

The function υ gives the set of variables within a formula Ω, i.e.

- $\upsilon(x^c) = \{x^c\}$
- $\upsilon(\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle) = \{{x_1}^{c_1}, {x_2}^{c_2}\}$
- $\upsilon(\Omega_1 \oplus \Omega_2) = \upsilon(\Omega_1) \cup \upsilon(\Omega_2)$

The function ω gives the set of links within a formula Ω, i.e.

- $\omega(x^c) = \emptyset$
- $\omega(\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle) = \{\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2} \rangle\}$
- $\omega(\Omega_1 \oplus \Omega_2) = \omega(\Omega_1) \cup \omega(\Omega_2)$

Finally, we can now define the satisfaction relation of a Ω formula with respect to an interpretation $\mathcal{I}$ as the existence of the isomorphic function between the SW-model and the explicit scope defined by the reduced interpretation with respect to the types in the SW-model.

Definition 8.1.8 ($\mathcal{I}^{\Omega}$ satisfaction relation)
Given a signature Σ, a Σ-formula Ω representing a SW-model, and a Σ-interpretation $\mathcal{I}$, the satisfaction relation is defined as follows:

$$\mathcal{I} \models_{\Sigma} \Omega \text{ if there exists a valuation function } K^{\mathcal{I}|_{\Omega}}(\Omega)$$

This definition can be extended for a set of formulas.

Example 8.1.9
It can be noticed that there is a trivial function $K^{\mathcal{I}|_{\Omega}}(\Omega)$ between the formula Ω representing a SW-model in Example 8.1.2 and the interpretation $\mathcal{I}$ defined in Example 6.3.3.

□

The definition of interpretations, reducts and homomorphisms do not change, thus original properties of the CSMOF institution with respect to these elements still hold: the existence of the category **Mod**(Σ) (Lemma 6.3.7), the functorial properties of reduct (Lemma 6.4.4), and the existence of the functor **Mod** (Lemma 6.4.5). Nevertheless, we need to prove the satisfaction condition, as shown in the following theorem. The complete proof is given in Appendix D.

Theorem 8.1.10 ($\mathcal{I}^{\Omega}$ satisfaction condition)**.** *Given signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I}$, and a Σ_1-formula Ω, the following satisfaction condition holds.*

$$\mathcal{I}|_{\sigma} \models_{\Sigma_1} \Omega \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\Omega)$$

Proof sketch. We know by definition that $\mathcal{I}|_{\sigma} \models_{\Sigma_1} \Omega$ holds if there exists a bijective function $K^{(\mathcal{I}|_{\Omega})|_{\sigma}}(\Omega)$, i.e. a bijective mapping variables in Ω such that syntactic links in Ω and semantic links in $(\mathcal{I}|_{\Omega})|_{\sigma}$ coincide. Since this interpretation is restricted to those types found in the formula Ω, we can find the same elements (no more or less) in $\mathcal{I}|_{\sigma(\Omega)}$. We can have more elements in $\mathcal{I}$ giving an interpretation to types and properties introduced by the signature morphism. However, these types and properties will not be considered in the translated formula $\sigma(\Omega)$ and thus they will not be part of $\mathcal{I}|_{\sigma(\Omega)}$. In conclusion, there is also a bijective function $K^{(\mathcal{I}|_{\sigma(\Omega)})}(\sigma(\Omega))$. This implication also holds backwards, and thus, the satisfaction condition holds.

□

Given that the satisfaction condition holds we can state that $\mathcal{I}^{\Omega}$ consisting of signatures, morphisms, formulas, interpretations, reducts, and the satisfaction relation, defines an institution.

8.2 Extending CSMOF and QVTR

We can easily combine this supporting institution $\mathcal{I}^{\Omega}$ and the original CSMOF and QVTR institutions, since the only differences between them are the definition of formulas and the corresponding satisfaction relation. In this context, we can define an extended CSMOF institution $\mathcal{I}^{\mathrm{M}^+}$ in which formulas are the disjoint union of both kind of formulas.

Definition 8.2.1 (**Extended** CSMOF **formulas**)

Given a CSMOF signature Σ, an extended CSMOF formula is defined as the disjoint union of CSMOF formulas and $\mathcal{I}^{\Omega}$ formulas, i.e.

$$\Psi \quad ::= \quad \Phi \mid \Omega$$

with Φ a CSMOF formula defined in Section 6.2, and Ω a $\mathcal{I}^{\Omega}$ formula defined in Section 8.1.

Since the definition of signature and signature morphisms coincide in both institutions, the category **Sign** is still defined, as in Lemma 6.2.7. Moreover, we proved in Lemma 6.2.8 that the functor *Sen* is defined for CSMOF formulas, and in Lemma 8.1.3 we do the same for $\mathcal{I}^{\Omega}$ formulas. By using the fact that a $\mathcal{I}^{\mathrm{M}^+}$-formula is the disjoint union of both kind of formulas, we can straightforwardly conclude that the functor *Sen* is still defined in this extended CSMOF institution.

Finally, we need to state the satisfaction relation.

Definition 8.2.2 (**Extended** CSMOF **satisfaction relation**)
Given a signature Σ, a Σ-formula and a Σ-interpretation $\mathcal{I}$, the interpretation satisfies the formula in one of the following cases:

- $\mathcal{I} \models_{\Sigma} \Phi$ as in Definition 6.4.1
- $\mathcal{I} \models_{\Sigma} \Omega$ as in Definition 8.1.8

As with the institution $\mathcal{I}^{\Omega}$, the definition of interpretations, reducts and homomorphisms do not change, thus original properties of the CSMOF institution with respect to these elements still hold. Moreover, the satisfaction condition holds.

Theorem 8.2.3 ($\mathcal{I}^{\mathrm{M}^+}$ satisfaction condition)**.** *Given signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I}$, and a Σ_1-formula φ, the following satisfaction condition holds.*

$$\mathcal{I}|_{\sigma} \models_{\Sigma_1} \varphi \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\varphi)$$

Proof. In the case of φ a CSMOF formula, we have that the satisfaction condition holds, as proved by Theorem 6.4.6. Moreover, in the case of φ a $\mathcal{I}^{\Omega}$ formula, we also have that the satisfaction condition holds, as proved by Theorem 8.1.10. By using the fact that a $\mathcal{I}^{\mathrm{M}^+}$-formula is the disjoint union of both kind of formulas, we can straightforwardly conclude that the satisfaction condition holds for the union of them.

□

Given that the satisfaction condition holds we can state that $\mathcal{I}^{\mathrm{M}^+}$ defines an extended CSMOF institution.

Example 8.2.4

Consider the formula Ω in Example 8.1.2 and the multiplicity constraints Φ in Example 6.4.2. We can now use the extended CSMOF institution to prove $\Omega \models_\Sigma \Phi$. We need to find an interpretation $\mathcal{I}$ such that $\mathcal{I} \models_\Sigma \Omega$ implies $\mathcal{I} \models_\Sigma \Phi$. The interpretation $\mathcal{I}$ in Example 6.3.3 satisfies the SW-model Ω, as shown in Example 8.1.9, and the multiplicity constraints Φ, as shown in Example 6.4.2.

But the definition of $\Omega \models_\Sigma \Phi$ is not given for a single interpretation $\mathcal{I}$, i.e. it holds if for all interpretations $\mathcal{I}$, we have that $\mathcal{I} \models_\Sigma \Omega$ implies $\mathcal{I} \models_\Sigma \Phi$. This holds only for multiplicity constraints Φ involving the same types defined in a SW-model Ω (those which are not affected by$|_\Omega$). For any other multiplicity constraint we can find an interpretation such that it satisfies Ω but not Φ, which is correct.

Lets consider an example in which there is a signature with types `A`, `B` and `C`, and associations between them. There is also an interpretation $\mathcal{I}$ with just two related elements, one of type `A` and another of type `B`. We also have two formulas: a multiplicity constraint formula φ = $\mathrm{B} \bullet \mathrm{rc} = 1$, i.e. there must be exactly one element of type C related with any element of type B, and a SW-model formula $\Omega = \mathrm{a}^{\mathrm{A}} \oplus \mathrm{b}^{\mathrm{B}} \oplus \langle \mathrm{ra}, \mathrm{a}, \mathrm{rb}, \mathrm{b} \rangle$.

We can notice that the interpretation provides a semantic representation for the SW-model, i.e. $\mathcal{I} \models_\Sigma \Omega$ but $\mathcal{I} \not\models_\Sigma \varphi$, since there is no semantic element of type `C`. This is the counterexample to prove that $\Omega \not\models_\Sigma \Phi$.

□

The extension of QVTR is very similar. We can define an extended QVTR institution $\mathcal{I}^{\mathrm{Q}^+}$ in which formulas are the disjoint union of QVTR formulas and $\mathcal{I}^{\mathrm{M}^+}$ formulas.

Definition 8.2.5 (**Extended** QVTR **formulas**)

Given a QVTR signature Σ, an extended QVTR formula is defined as the disjoint union of QVTR formulas and extended $\mathcal{I}^{\mathrm{M}^+}$ formulas, i.e.

$$\Pi \quad ::= \quad \Psi_i \mid \varphi^{\mathrm{K}} \mid \varphi^{\mathrm{R}}$$

with Ψ_i ($i = \{1, 2\}$) a $\mathcal{I}^{\mathrm{M}^+}$ formula, as defined before, indexed by the institution in which it is defined; φ^{K} a key constraint, and φ^{R} a set of interrelated transformation rules, both defined in Section 7.2.

Since the definition of signature and signature morphisms coincide with the QVTR institutions, the category **Sign** is still defined, as in Lemma 7.2.5. Moreover, we proved in Lemma 7.2.6 that the functor *Sen* is defined for QVTR formulas, and in the last section we do the same for extended CSMOF formulas. By using the fact that a $\mathcal{I}^{\mathrm{Q}^+}$-formula is the disjoint union of both kind of formulas, we can straightforwardly conclude that the functor *Sen* is still defined in this extended QVTR institution.

Finally, we state the satisfaction relation.

Definition 8.2.6 (Extended QVTR satisfaction relation)
Given a signature Σ, a Σ-formula and a Σ-interpretation $\mathcal{I} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$, the interpretation satisfies the formula in one of the following cases:

- $\langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle \models_\Sigma \Psi_i$ if $\mathcal{M}_i^{\mathrm{M}} \models_\Sigma \Psi_i$ as in Definition 8.2.2.
- $\langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle \models_\Sigma \varphi^{\mathrm{K}}$ as in Definition 7.4.1
- $\langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle \models_\Sigma \varphi^{\mathrm{R}}$ as in Definition 7.4.1

The definition of interpretations, reducts and homomorphisms is the same as with the QVTR institution, thus the properties with respect to these elements still hold. Moreover, the satisfaction condition holds.

Theorem 8.2.7 ($\mathcal{I}^{\mathrm{Q}^+}$ satisfaction condition)**.** *Given signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I}$, and a Σ_1-formula φ, the following satisfaction condition holds.*

$$\mathcal{I}|_\sigma \models_{\Sigma_1} \varphi \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\varphi)$$

Proof. In the case of φ a QVTR formula, we have that the satisfaction condition holds, as proved by Theorem 7.4.3. Moreover, in the case of φ an extended CSMOF formula, we also have that the satisfaction condition holds, as proved by Theorem 8.2.3. By using the fact that a $\mathcal{I}^{\mathrm{Q}^+}$-formula is the disjoint union of both kind of formulas, we can straightforwardly conclude that the satisfaction condition holds for the union of them.

□

Given that the satisfaction condition holds we can state that $\mathcal{I}^{\mathrm{Q}^+}$ defines an extended QVTR institution.

Example 8.2.8

Following with the example, we can consider a set of formulas Ψ_1 such that it contains a formula Ω_1 corresponding to the class SW-model in Figure 2.7, as defined in Example 8.1.2, and the set of multiplicity constraint formulas defined in Example 6.2.5. Moreover, we can consider a set of formulas Ψ_2 composed by a formula Ω_2 corresponding to the relational SW-model in Figure 2.7, and the set of multiplicity constraint formulas derived from the relation metamodel in Figure 2.6, which were not defined in past chapters. We can also have a set of QVTR formulas φ representing keys and transformation rules, as in Example 7.2.7, and an interpretation $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$ as defined in Example 7.3.2, which satisfies these formulas as shown in Example 7.4.2.

With all this information we can prove that the QVTR formulas φ are implied by the SW-models in Ψ_1 and Ψ_2, i.e. $\Psi_1 \cup \Psi_2 \models_\Sigma \varphi$, since there exists the interpretation $\mathcal{M}$ such that $\mathcal{M} \models_\Sigma \Psi_i$ and $\mathcal{M} \models_\Sigma \varphi$. □

8.3 Discussion

In this section we discuss the definition of the satisfaction relation of SW-model formula with respect to an interpretation as the existence of the isomorphic function between the SW-model and the explicit scope defined by the reduced interpretation with respect to the types in the SW-model.

Formulas of the CSMOF and QVTR institutions represent multiplicity constraints, key constraints and transformation rules. These formulas are satisfied, in the corresponding institutions, with respect to a given interpretation representing a specific (or a pair of) SW-model. However, as discussed in Section 5.4, we need to define an entailment system $\vdash$ to verify whether multiplicity constraints, represented as a set of formulas Ψ, are derived from a syntactic representation of a SW-model, represented as a formula Ω, i.e. $\Omega \vdash_{\Sigma} \Psi$. In the same way, we also need to verify whether a key constraint (or a set of them), represented as a formula φ^{K}, is derived from the same Ω, i.e. $\Omega \vdash_{\Sigma} \varphi^{\mathrm{K}}$, or whether a transformation rule φ^{R} (or the whole model transformation) is derived from a pair of SW-models, i.e. $\Omega_1 \cup \Omega_2 \vdash_{\Sigma} \varphi^{\mathrm{R}}$.

A sound entailment system will ensure semantic entailment, i.e. $\Omega \vdash_{\Sigma} \Psi$ implies $\Omega \models_{\Sigma} \Psi$. Semantic entailment is defined by the satisfaction relations of the institutions. As defined in Section 4.2, $\Omega \models_{\Sigma} \Psi$ requires to find an interpretation $\mathcal{I}$ such that $\mathcal{I} \models_{\Sigma} \Omega$ implies $\mathcal{I} \models_{\Sigma} \Psi$. However, if the interpretation $\mathcal{I}$ provides more elements than those used for interpreting the formula representing the SW-model, we can have some contradictory cases, as shown in the following example.

Consider a SW-model with two elements of classes `A` and `B` and a link between them, and a multiplicity constraint of cardinality 1-2 between `A` and `B`. Also consider an interpretation with a semantic interpretation of those elements, plus one object of class `B` with a relation to the object of class `A`. We can notice that the interpretation provides a semantic representation for the SW-model (and other elements) and it also satisfies the multiplicity constraints. However, it is clear that the multiplicity constraint does not hold in this SW-model, since we have only one element of class `B` linked to the element of class `A`.

In this sense, we need an isomorphism between the SW-model and the interpretation, thus any isomorphic interpretation $\mathcal{I}$ will satisfy $\Omega \models_{\Sigma} \Psi$ and the syntactic relation between Ω and Ψ will also make sense.

We also need to ensure that the satisfaction condition with respect to Ω holds, i.e. $\mathcal{I}|_{\sigma} \models_{\Sigma_1} \Omega$ iff $\mathcal{I} \models_{\Sigma_2} \sigma(\Omega)$. The problem is that if Σ_2 has more elements that Σ_1 we can have the case in which $\mathcal{I}|_{\sigma} \models_{\Sigma_1} \Omega$ since there is an isomorphism between $\mathcal{I}|_{\sigma}$ and Ω, but $\mathcal{I} \not\models_{\Sigma_2} \sigma(\Omega)$ since $\mathcal{I}$ can have more elements than $\mathcal{I}|_{\sigma}$ (corresponding to elements in Σ_2 which are not in Σ_1), and thus the isomorphism is not possible.

In the last example, we can have another signature Σ_2 with the same elements as before plus another class C. We can define the signature morphism as the identity, and take as an interpretation the same as before plus another object of class C. In this case, it is clear that $\mathcal{I}|_\sigma \models_{\Sigma_1} \Omega$ since the object of class C does not exists within $\mathcal{I}|_\sigma$. However, it does exists within $\mathcal{I}$, thus $\mathcal{I} \not\models_{\Sigma_2} \sigma(\Omega)$ since there is no possible isomorphism.

To solve this problem we took a similar approach as in [CS13b], in which an interpretation is reduced with respect to the types used in the definition of the SW-model formula Ω, before checking the satisfaction condition. This defines an *explicit scope* in which the formula is verified, and only affects the definition of the satisfaction relation for SW-model formulas, not the original definitions.

9 Connecting the Institutions with CASL

In this chapter we present the definition of generalized theoroidal comorphisms (introduced in Chapter 4) from our extended institutions defined in Chapter 8 to the Common Algebraic Specification Language (CASL, [CoF04]), a general-purpose specification language. The importance of defining such comorphisms is that CASL is the main language within the Heterogeneous Tool Set (HETS, [MML07, Mos05]), a tool meant to support heterogeneous multi-logic specifications. HETS provides tool support for the verification of MDE elements, as introduced in Chapter 10, but also for moving inside the graph of logics within HETS and take advantage of the benefits of each logic.

In Section 9.1 we present CASL and its founding institution. Then, in Section 9.2 we define the encoding of the CSMOF extension into CASL by means of a generalized theoroidal comorphism, and in Section 9.3 we do the same for the QVTR extension. Finally, in Section 9.4 we close this chapter with a discussion of related work. Along the definition we illustrate the main concepts with the example introduced in Chapter 2. For the sake of readability, complete proofs are given in Appendix E.

9.1 Borrowing an Entailment System

In Chapter 8 we introduce extensions of the CSMOF and QVTR institutions for the definition of an entailment system devised to verify whether multiplicity constraints, key constraints and transformation rules are derived from SW-models. However, as described in Section 5.1 we do not define our own entailment system, but borrow an existent one through the definition of an institution comorphism (generalized theoroidal comorphisms indeed) to CASL.

Since CASL has a sound proof calculus for entailment, and our comorphisms admits borrowing of entailment, we can translate our proof goals using the comorphism into CASL and use its proof calculus also for proving entailment concerning our extended CSMOF and QVTR specifications. As explained in Section 8.3, since the entailment system is sound, we also have a connection with the model-theoretic world defined by the satisfaction relation of the extended CSMOF and QVTR institutions ensuring the desired properties.

9.1.1 Common Algebraic Specification Language

The institution underlying CASL is the sub-sorted partial first-order logic with equality and constraints on sets $SubPCFOL^{=}$ [MHST03].

Signatures are many-sorted algebraic signatures enriched with predicate and function symbols of the form $(S, TF, PF, P, \leq_S)$ where S is a set (of sort names), TF and PF are $S^* \times S$-sorted disjoint families of total and partial function symbols, respectively, $P = (P_w)_{w \in S^*}$ is a family of predicate symbols, and $\leq_S$ is a reflexive and transitive subsort relation (embedding) on the set S of sorts. Signature morphisms consist of maps taking sort, function and predicate symbols respectively to a symbol of the same kind, and they must preserve subsorting, typing of function and predicate symbols and totality of function symbols. Signatures and signature morphisms can be extended with a total injection function symbol inj, a partial projection function symbol pr between sorts, and a unary membership predicate $\in^s$.

Sentences are the usual partial many-sorted first-order logic formulas together with sort generation constraints. Many-sorted first-order logic formulas are built out from atomic formulas using the standard propositional connectives ($\wedge$,$\vee$,$\Rightarrow$,$\Leftrightarrow$,$\neg$) and quantifiers ($\forall$, $\exists$). The atomic formulas are applications of qualified predicate symbols to argument terms (variables or application of functions) of appropriate sorts, assertions about the definedness of fully-qualified terms, or existential $t \stackrel{e}{=} t'$ and strong $t \stackrel{s}{=} t'$ equations between fully-qualified terms of the same sort, the logical constants `true` and `false`, and variables $X^s = (X^s)_{s \in S}$. Sort generation constraints are triples (S', F', σ') such that $\sigma' : \Sigma' \to \Sigma$ and S' and F' are respectively sort and function symbols of Σ', stating that a given set of sorts is generated by a given set of functions. Formulas are translated along a signature morphism $\varphi : \Sigma \to \Sigma''$ by replacing symbols as prescribed by φ while sort generation constraints are translated by composing the morphism σ' in their third component with φ.

Models are many-sorted first-order structures, i.e. consisting of a non-empty carrier set $|M|_s$ for each sort name $s \in S$, a partial function (or total) f_M for each function symbol $f \in PF$ (or TF), and a relation p_M for each predicate symbol $p \in P_w$, $w \in S^*$, satisfying some axioms with respect to embedding, projection, and membership. Homomorphisms between models M and M' consist of a function $h_s : |M|_s \to |M'|_s$ for each $s \in S$ preserving not only the values of functions but also their definedness, and preserving the truth of predicates. Reducts are defined by interpreting symbols of the signature in the reduct in the same way that their images under the signature morphism are interpreted.

Finally, the satisfaction relation is basically the usual satisfaction of a partial first-order formula in a first-order structure. A sort generation constraint (S', F', σ') holds in a model M if the carriers of the reduct of M along σ' of the sorts in S' are generated by function symbols in F'. A sentences is satisfied in a model if it is satisfied with respect to all variable valuations.

9.2 Encoding CSMOF into CASL

We define a generalized theoroidal institution comorphism between the extended CSMOF institution $\mathcal{I}^{\mathrm{M}^+}$ defined in Chapter 8 and $\mathcal{I}^{\mathrm{C}}$ for $SubPCFOL^{=}$, i.e.

- a functor $\Phi : \mathrm{Th}^{\mathcal{I}^{\mathrm{M}^+}} \rightarrow \mathrm{Th}^{\mathcal{I}^{\mathrm{C}}}$, with Th the category of theories and theory morphisms
- a natural transformation $\beta : (\Phi)^{op}; \mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}} \rightarrow \mathbf{Mod}^{\mathcal{I}^{\mathrm{M}^+}}$, with $\mathbf{Mod} : \mathrm{Th} \rightarrow \mathbf{Cat}$ the functor giving the category of models of a theory.

We use generalized theoroidal comorphism since we introduce in a CASL signature some functions representing typed elements (of a SW-model) contained within extended CSMOF formulas. This is not mandatory, but helpful in order to construct a more optimal and readable CASL representation, without the need of existential and equality axioms.

Given a theory $T = \langle \Sigma, \Psi \rangle \in \mathrm{Th}^{\mathcal{I}^{\mathrm{M}^+}}$, the functor Φ is defined in three steps: first, Σ is translated into a CASL theory, then a SW-model formula in Ψ is translated into CASL formulas and some functions are added to the CASL signature of the same theory, and finally multiplicity constraint formulas in Ψ are translated into CASL formulas of the same theory.

The class hierarchy represented within a $\mathcal{I}^{\mathrm{M}^+}$ signature is basically translated into a set of sorts complying with a subsorting relation, properties are translated into predicates, and an axiom is introduced to relate predicates derived from bidirectional properties. Formally, every $\mathcal{I}^{\mathrm{M}^+}$ signature $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$ and $\boldsymbol{P} = (R, P)$ is translated into a theory $((S, TF, PF, P, \leq_S), E)$ such that:

- For every class name c in C, there is a sort name $c \in S$.
- For every $c_1 \leq_C c_2$ with $c_1, c_2 \in C$, we have $c_1 \leq_S c_2$ with $c_1, c_2 \in S$.
- For every $c \in \alpha$ there is an axiom in E stating that c is the disjoint embedding of it subsorts (sort generation constraint).
- For every $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, there are two predicates $r_1 : c_2 \times c_1$ and $r_2 : c_1 \times c_2 \in \Pi$, and an axiom in E stating the equivalence of the predicates, i.e. $r_1(x, y)$ iff $r_2(y, x)$ with $x \in S_1, y \in S_2$. In the case of predicates with the default role name _, we only generate the predicate in the opposite direction of the default role, i.e. if $\langle \_ : c_1, r_2 : c_2 \rangle$ or $\langle r_1 : c_1, \_ : c_2 \rangle$ we only have $r_2 : c_1 \times c_2$ or $r_1 : c_2 \times c_1$, respectively.

We consider the existence of a built-in extension of the institution $\mathcal{I}^{\mathrm{C}}$, e.g. the CASL standard library. In this extension, the sets of functions TF and PF contain those functions defined for built-in types (like $\leq$ for integers and $+$ for strings).

Example 9.2.1

The signature in Example 6.2.3 corresponding to the class metamodel in Figure 2.6 is translated into a theory $((S, TF, PF, P, \leq_S), E)$ such that:

$$
\begin{array}{lcl}
S & = & \{\mathrm{UMLModelElement}, \mathrm{Package}, \mathrm{Classifier}, \mathrm{PrimitiveDataType}, \\
 & & \ \mathrm{Attribute}, \mathrm{Class}, \mathrm{String}\} \\
\leq_S & = & \{\mathrm{Package} \leq_S \mathrm{UMLModelElement}, \\
 & & \ \mathrm{Attribute} \leq_S \mathrm{UMLModelElement}, \\
 & & \ \mathrm{Classifier} \leq_S \mathrm{UMLModelElement}, \\
 & & \ \mathrm{Class} \leq_S \mathrm{Classifier}, \mathrm{PrimitiveDataType} \leq_S \mathrm{Classifier}\} \\
P & = & \{\ \mathrm{elements} : \mathrm{Package} \times \mathrm{Classifier}, \mathrm{namespace} : \mathrm{Classifier} \times \mathrm{Package}, \\
 & & \ \mathrm{name} : \mathrm{UMLModelElement} \times \mathrm{String}, \\
 & & \ \mathrm{kind} : \mathrm{UMLModelElement} \times \mathrm{String}, \\
 & & \ \mathrm{attribute} : \mathrm{Class} \times \mathrm{Attribute}, \mathrm{owner} : \mathrm{Attribute} \times \mathrm{Class}, \\
 & & \ \mathrm{type} : \mathrm{PrimitiveDataType} \times \mathrm{Attribute}\}
\end{array}
$$

Since $\mathrm{UMLModelElement} \in \alpha$, there is a sort generation constraint in E stating that $\mathrm{UMLModelElement}$ is the disjoint embedding of its subsorts $\mathrm{Attribute}$, $\mathrm{Classifier}$, and $\mathrm{Package}$. There are also axioms stating the equivalence of the predicates derived from bidirectional properties, e.g. $\forall\, x : \mathrm{Package}, y : \mathrm{Classifier}.\ \mathrm{elements}(x, y) \Leftrightarrow \mathrm{namespace}(y, x)$

□

In the case of a SW-model formula Ω, each variable within the formula (representing an object) is translated into a total function of the corresponding type. We also add several axioms in order to represent implicit constraints in the $\mathcal{I}^{\mathrm{M}^+}$ institution which are not necessarily kept when representing the basic elements in $SubPCFOL^{=}$, as for example the need of distinguishing between two different variables (functions in the target institution) and the specification of the cases in which a property holds (when there is a syntactic link represented within the formula Ω). Formally,

- For every $x^c \in \upsilon(\Omega)$ there is a total function (constant) $x : c \in TF$ with $c \in S$
- For every $\langle r_1, x^{c_1}, r_2, y^{c_2} \rangle \in \omega(\Omega)$ with $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, there is an axiom in E stating that the predicate $r_2 : c_1 \times c_2$ holds for $x : c_1, y : c_2 \in TF$. Notice that the opposite direction holds for the equivalence of the predicates stated during the signature translation.
- E has some additional axioms:
  - Distinguishability: $\{x_i \neq x_j \mid i \neq j.\ x_i, x_j : c \in TF\}$ for all $c \in S$
  - Completeness of elements: for all $x : c$ we have that $x = o_i$ for some $o_i : c \in TF$.

When c is a non-abstract class having sub-classes, completeness must be defined for $o_i : c' \in TF$ for all $c' \leq c$. These are sort generation constraints.

– Completeness of relations: for all $x : c_1, y : c_2$ we have that $r(x, y)$ only if $x = o_1$ and $y = o_2$ for some $o_1 : c_1, o_2 : c_2 \in TF$ for which $r(c_1, c_2)$ hold.

The "distinguishability" and "completeness of elements" axioms correspond to the so-called "no junk, no confusion" principle: there are no other values than those denoted by the functions $x : c \in TF$, and all distinct functions denote different values.

Example 9.2.2

The variables within the class SW-model in Figure 2.7 is translated into the following set of total functions

$$\begin{aligned} TF \quad = \quad & \{\mathrm{p} : \mathrm{Package}, \mathrm{c} : \mathrm{Class}, \mathrm{a} : \mathrm{Attribute}, \mathrm{pdt} : \mathrm{PrimitiveDataType} \\ & \mathrm{Package} : \mathrm{String}, \mathrm{ID} : \mathrm{String}, \mathrm{Persistent} : \mathrm{String}, ...\} \end{aligned}$$

Moreover, for every link there is an axiom stating that the corresponding predicate holds for the functions corresponding to the translated elements within the link. This axiom can be stated in conjunction with the "completeness of relations" constraint as follows:
$\forall\, x : \mathrm{Package}, y : \mathrm{Classifier}.\ \mathrm{elements}(x, y) \Leftrightarrow (x = \mathrm{p} \wedge y = \mathrm{c}) \vee (x = \mathrm{p} \wedge y = \mathrm{pdt})$

The "completeness of elements" constraint is a sort generation constraint such that for a given sort representing a type, elements of that sort are one of the total functions representing objects of that type. For example, in the case of the non-abstract class Classifier which has sub-classes, we have the following axiom: $\forall\, x : \mathrm{Classifier}.\ x = \mathrm{c} \vee x = \mathrm{pdt}$.

Finally, the "distinguishability" constraint must be stated between elements of sorts related by the subsorting relation (there is no confusion between elements of non-related sorts). For example, in the case of the elements within the UMLModelElement hierarchy, we have the following constraint:
$\neg\,(\mathrm{a} = \mathrm{c}) \wedge\ \neg\,(\mathrm{a} = \mathrm{p}) \wedge\ \neg\,(\mathrm{a} = \mathrm{pdt}) \wedge\ \neg\,(\mathrm{c} = \mathrm{p}) \wedge\ \neg\,(\mathrm{c} = \mathrm{pdt}) \wedge\ \neg\,(\mathrm{p} = \mathrm{pdt})$

□

For the translation of a multiplicity constraint formula we define the following predicates for constraining the size of the set of elements in a relation with some other:

- $min(n, R : D \times C)$ holds if for all $y : D$ exists $x_1, ..., x_n : C$ such that $R(y, x_i)$ for all $i = \{1..n\}$, and $x_i \neq x_j$ for all $i = \{1..n-1\}$, $j = i + 1$.
- $max(n, R : D \times C)$ holds if for all $y : D$ and $x_1, ..., x_{n+1} : C$, $Rel(y, x_i)$ for all $i = \{1..n+1\}$ implies there is some $x_i = x_j$ $i = \{1..n\}$, $j = i + 1$

The first predicate states that there are at least n different elements related to every element y by the relation R, which represents a minimal cardinality for the relation. The other predicate states that there are no more than n elements related to any element y by the relation R, which

represents a maximal cardinality for the relation. Using these predicates, we can translate any multiplicity constraint formula as follows:

- $n \leq \#D \bullet R$ is translated into $min(n, R : D \times C)$
- $\#D \bullet R \leq n$ is translated into $max(n, R : D \times C)$
- $\#D \bullet R = n$ is translated into $min(n, R : D \times C) \wedge max(n, R : D \times C)$

such that $Q : C \times D, R : D \times C \in \Pi$ are the predicates generated by the functor $\Phi(\langle R : C, Q : D\rangle)$. In the case of $C \bullet Q$ the predicate $Q : C \times D$ is used instead of $R : D \times C$.

Example 9.2.3

The formula $\#(\text{UMLModelElement} \bullet \text{name}) = 1$ in Example 6.2.5 is translated into the conjunction of

$$min(1, \text{name} : \text{UMLModelElement} \times \text{String}) = \\ \forall\, x_1 : \text{UMLModelElement}.\ \exists\, y_1 : \text{String}.\ \text{name}(x_1, y_1)$$

$$max(1, \text{name} : \text{UMLModelElement} \times \text{String}) = \\ \forall\, x_1 : \text{UMLModelElement}, y_2, y_1 : \text{String}. \\ (\text{name}(x_1, y_1) \wedge\ \text{name}(x_1, y_2)) \Rightarrow y_1 = y_2$$

Moreover, the formula $\#(\text{Class} \bullet \text{attribute}) \leq 2$ is translated into

$$max(2, \text{attribute} : \text{Class} \times \text{Attribute}) = \\ \forall\, x_1 : \text{Class}, y_3, y_2, y_1 : \text{Attribute}. \\ (\text{attribute}(x_1, y_1) \wedge\ \text{attribute}(x_1, y_2) \wedge\ \text{attribute}(x_1, y_3)) \Rightarrow \\ (y_1 = y_2 \vee\ y_1 = y_3 \vee\ y_2 = y_3)$$

□

Now we can define the translation of a theory morphism in $\mathcal{I}^{\mathrm{M}^+}$, i.e. how signature morphisms are translated. Given $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1, 2)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$ and $\boldsymbol{P}_i = (R_i, P_i)$, and a signature morphism $\sigma : \Sigma_1 \to \Sigma_2 = \langle \sigma_T, \sigma_R \rangle$, its translation $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ signature morphism $\langle \sigma_S, \sigma_F, \sigma_P \rangle$ defined as follows:

- $\sigma_S(\Phi(c)) = \Phi(\sigma_T(c))$ for every $c \in C_i$
- the subsort preservation condition as well as the disjoint embedding constraint of $\Phi(\sigma)$ follow from the similar conditions in σ.
- $\sigma_P(r_1(c_2, c_1)) = \sigma_R(r_1)(\sigma_T(c_2), \sigma_T(c_1))$, and $\sigma_P(r_2(c_1, c_2)) = \sigma_R(r_2)(\sigma_T(c_1), \sigma_T(c_2))$, for every $\langle r_1 : c_1, r_2 : c_2\rangle \in P_1$ and $r_1(c_2, c_1), r_2(c_1, c_2)$ the predicates generated from $\Phi(\langle r_1 : c_1, r_2 : c_2\rangle)$.

For each morphism between formulas in $\mathcal{I}^{\mathrm{M}^+}$, i.e. the canonical application of a signature morphism to every element in the formula, there is a morphism between formulas in $\mathcal{I}^{\mathrm{C}}$ which is the canonical application of the translated signature morphism to every element in the translated formula. We have that the extension of $\mathcal{I}^{\mathrm{M}^+}$ signature morphisms to theories $\sigma : \langle \Sigma, \Psi \rangle \to \langle \Sigma_2, \Psi_2 \rangle$ is a theory morphism since $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$. We can prove that $\Phi(\sigma)$ is also a theory morphism. Complete proof is given in Appendix E.

Lemma 9.2.4. *Given $\mathcal{I}^{\mathrm{M}^+}$ signatures Σ_i, and a theory morphism $\sigma : \Sigma_1 \to \Sigma_2$, its translation $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.*

Proof sketch. We know that given a set of Σ_1-formulas Ψ, and a set of Σ_2-formulas Ψ_2 they comply with the theory morphism $\sigma : \langle \Sigma, \Psi \rangle \to \langle \Sigma_2, \Psi_2 \rangle$ if $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$. In such case we have an interpretation $\mathcal{I}$ satisfying Ψ_2, i.e. determining a concrete SW-model which satisfies the multiplicity constraints, which also satisfies $\sigma(\Psi)$. These formulas are derivable multiplicity constraints or same structured SW-models with potentially less types (remember that there must be an isomorphism between the SW-model and the explicit scope defined by the reduced interpretation with respect to the types in the SW-model). The translation Φ not only maps types and roles in a consistent way, but also adds axioms constraining the $\mathcal{I}^{\mathrm{C}}$ models that must be used for checking the satisfaction relation (e.g. the "distinguishability" and "completeness of elements" axioms). In this sense, the interpretation satisfying $\Phi(\Psi)$ and $\Phi(\Psi_2)$ must have the same structure that the original $\mathcal{I}$. Thus, we have that $\Phi(\Psi_2) \models_{\Phi(\Sigma_2)} \Phi(\sigma)(\Phi(\Psi))$ also hold, and in conclusion $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.

□

We can prove that Φ is indeed a functor. Complete proof is given in Appendix E.

Lemma 9.2.5. *The function $\Phi : Th^{\mathcal{I}^{\mathrm{M}^+}} \to Th^{\mathcal{I}^{\mathrm{C}}}$ is a functor from the category of $\mathcal{I}^{\mathrm{M}^+}$ theories and theory morphisms to the category of theories in $SubPCFOL^{=}$.*

Proof sketch. By definition we have that domain and codomain of the image of a $\mathcal{I}^{\mathrm{M}^+}$ signature morphism are the images of domain and codomain, respectively, of the signature morphism. We also have that the translation of a signature morphism changes sorts and predicates consistently with respect to the change of types and roles within the signature morphism. In this sense, the application of the translation of composed signature morphisms to a translated $SubPCFOL^{=}$ formula gives the same result that the independent application of the translation of each signature morphism to such formula. Thus, composition is preserved. Moreover, the identity signature morphism id_σ in $\mathbf{Sign}^{\mathrm{M}}$ is a tuple of identity functions for types and roles, and its translation $\Phi(id_\sigma)$ gives an identity signature morphism in $SubPCFOL^{=}$. Thus, identities are preserved. Finally, Φ is a functor.

□

The semantic part of the comorphism corresponds to the definition of the natural transformation β, which involves the translation of $\mathcal{I}^{\mathrm{C}}$ interpretations and homomorphisms into $\mathcal{I}^{\mathrm{M}^+}$ interpretations and homomorphisms.

Given a $\mathcal{I}^{\mathrm{M}^+}$ theory $T = \langle \Sigma, \Psi \rangle \in \mathrm{Th}^{\mathcal{I}^{\mathrm{M}^+}}$, a $\mathcal{I}^{\mathrm{C}}$ model M of its translated theory (Σ', E) is translated into a Σ-interpretation denoted $I = (\boldsymbol{V_C^T(O)}, \boldsymbol{A})$ such that:

- Each non-empty carrier set $|M|_s$ with $s \in S$, is translated into the set V_c in the object domain $\boldsymbol{V_C^T(O)}$ such that s is the translation of type $c \in T(C)$. Since the functor Φ translates every $c \in \alpha$ into an axiom in E stating that c is the disjoint embedding of it subsorts, a valid $|M|_c = \bigcup_{c_1 \leq_S c} |M|_{c_1}$. This property ensures that $c_2 \in \alpha$ implies $O_{c_2} = \bigcup_{c_1 \leq_C c_2} O_{c_1}$.
- Each relation p_M of a predicate symbol $r_2(c_1, c_2) \in P$ derived from the translation of a predicate $\langle r_1 : c_1, r_2 : c_2 \rangle$, is translated into the relation $p^{\mathcal{I}} \subseteq V_{c_1} \times V_{c_2} \ \in \boldsymbol{A}$.

Given a $\mathcal{I}^{\mathrm{M}^+}$ theory $T = \langle \Sigma, \Psi \rangle \in \mathrm{Th}^{\mathcal{I}^{\mathrm{M}^+}}$, $\mathcal{M}$ and $\mathcal{M}'$ be two $\mathcal{I}^{\mathrm{C}}$ models of its translated theory (Σ', E), and let $h' : \mathcal{M} \to \mathcal{M}'$ be a Σ'-homomorphism. Let us denote $\mathcal{I} = \beta(\mathcal{M})$ and $\mathcal{I}' = \beta(\mathcal{M}')$ and let us define $h : \mathcal{I} \to \mathcal{I}'$ as follows: for any $c \in T(C)$, $h_c = h'_{\Phi(c)}$. This is a homomorphism:

- Since h' gives different carrier sets for each sort, we have that $h_c(v) \in O'_c$ forall $v \in O_c$, and $h_c(v) \in V'_c \backslash O'_c$ forall $v \in V_c \backslash O_c$
- Since h' preserves the values of relations, we have that $(v_1, v_2) \in p^{\mathcal{I}}$ iff $(h_{c_1}(v_1), h_{c_2}(v_2)) \in p^{\mathcal{I}'}$

We can prove that β is indeed a natural transformation (complete proof in Appendix E).

Lemma 9.2.6. *The function* $\beta : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}} \to \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}}$*, with* $\boldsymbol{Mod} : Th \to \boldsymbol{Cat}$ *the functor giving the category of models of a theory, is a natural transformation, i.e. a family of arrows* $\beta_A : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}(A) \Rightarrow \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}}(A)$*, one for each theory* A *of* $\boldsymbol{Sign}^{\mathrm{M}}$*, such that, for every theory morphism* $\sigma : A \to B$ *it holds:* $\boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}} \circ \beta_B = \beta_A \circ (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}$

Proof sketch. Given any model in $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(B))$, the translation β_B gives an interpretation such that: each non-empty carrier set $|M|_s$ is translated into the set V_c such that s is the translation of type c, and each relation p_M of a predicate symbol derived from the translation of a predicate is translated into a relation. By definition of reduct, the application of $\mathbf{Mod}^{\mathcal{I}^{\mathrm{M}^+}}$ to this interpretation gives the same interpretation. In the other side, the reduct $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(\sigma))$ gives an interpretation of symbols of the translated signature A, and it composition with the translation β_A produces the same interpretation as before, since the carries sets $|M|_s$ and the relations p_M are those derived from the translation of elements in the translates signature, which was reduced to the elements in the signature A. In the case of homomorphisms, its translation is defined in conformance with the original homomorphism, and the reduct gives the same homomorphism. Finally, β is a natural transformation. □

There is a final result which states that the comorphism admits model expansion, and thus admits borrowing of entailment and refinement for theories. The comorphism admits model expansion since β is pointwise surjective on objects, i.e. each model of a $\mathcal{I}^{\mathrm{M}^+}$-theory has a corresponding model in the translated theory within the $\mathcal{I}^{\mathrm{C}}$ institution. Just consider any model $(\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$, we can construct a $\mathcal{I}^{\mathrm{C}}$ model such that $|M|_s$ corresponds to V_c with s the translation of type $c \in T(C)$, and for each relation $\langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}} \in \boldsymbol{A}$ a pair of relations p_M (or just one if the relation is unidirectional). Since they are equivalent, if the model satisfies the $\mathcal{I}^{\mathrm{M}^+}$-theory, it also satisfies the translated theory.

9.3 Encoding QVTR into CASL

We define a generalized theoroidal institution comorphism between the extended QVTR institution $\mathcal{I}^{\mathrm{Q}^+}$ defined in Chapter 8 and $\mathcal{I}^{\mathrm{C}}$for $SubPCFOL^{=}$, i.e.

- a functor $\Phi : \mathrm{Th}^{\mathcal{I}^{\mathrm{Q}^+}} \rightarrow \mathrm{Th}^{\mathcal{I}^{\mathrm{C}}}$, with Th the category of theories and theory morphisms
- a natural transformation $\beta : (\Phi)^{op}; \mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}} \rightarrow \mathbf{Mod}^{\mathcal{I}^{\mathrm{Q}^+}}$, with $\mathbf{Mod} : \mathrm{Th} \rightarrow \mathbf{Cat}$ the functor giving the category of models of a theory.

As with the encoding of extended CSMOF, we use generalized theoroidal institution comorphism since we introduce in the CASL signature some elements in order to construct a more optimal and readable CASL representation. In this case we will define predicates (within the CASL signature) corresponding to transformation rules (within a $\mathcal{I}^{\mathrm{Q}^+}$ formula) such that it is possible to call a rule just referencing the predicate.

Given a theory $T = \langle \Sigma, \Psi \rangle \in \mathrm{Th}^{\mathcal{I}^{\mathrm{Q}^+}}$, the functor Φ is defined in three steps: first, Σ is translated into a CASL theory, then a key formula in Ψ is translated into a CASL formula and a predicate of the same theory, and finally transformation rule formulas in Ψ are translated into CASL formulas and some predicates of the same theory. The translation of $\mathcal{I}^{\mathrm{M}^+}$ formulas which can also be within Ψ is the same as in the last chapter.

Every $\mathcal{I}^{\mathrm{Q}^+}$ signature $\langle \Sigma_1^{\mathrm{M}}, \Sigma_2^{\mathrm{M}} \rangle$ is translated by the functor Φ into a theory such that each signature Σ_i^{M} is translated as defined in the encoding of extended CSMOF into CASL. Notice that since there are no name clashes between the signatures, the result is the disjoint union of both translations. As said before with respect to the expressions language, we assume that the institution $\mathcal{I}^{\mathrm{E}}$ has a correspondence (via a comorphism) with the built-in extension of the institution $\mathcal{I}^{\mathrm{C}}$ introduced in the last section.

Example 9.3.1

The functor Φ translates the signature in Example 7.2.2 into a theory $((S, TF, PF, P, \leq_S), E)$ such that the signatures Σ_i^{M} are translated as in Example 9.2.1.

□

$\mathcal{I}^{\mathrm{Q}^+}$ formulas representing keys and transformation rules are translated into named first-order formulas. Formulas will be of the form $P \Leftrightarrow F$ such that P is the predicate naming the formula, and F the conditions which must hold in order to satisfy a key constraint φ^{K} or transformation φ^{R}.

In the case of a formula φ^{K}, the formula F defines that there are not two different instances of that class with the same combination of properties conforming the key of such class. Formally, every formula $\varphi^{\mathrm{K}} = \langle C, \{r_1, ..., r_n\}\rangle$ is translated into

- a predicate key_C naming a key constraint definition
- a formula of the form $key\_C \Leftrightarrow \forall x, y \ \in C, v_j : T_j.\ x \neq y \rightarrow \bigwedge_{i,j} r_i(x, v_j) \ \rightarrow \bigvee_{i,j} \neg r_i(y, v_j)$, with $r_i(\_,\_)$ one of the two predicates obtained from the translation of the property $\langle r_1 : C_1, r_2 : C_2\rangle \in P_i$ such that one of the roles is of type C and the other of type T_j. In the case that r_i is the role of C in the property (because the opposite role is not navigable), we use $r_i(v_j, x)$ instead of $r_i(x, v_j)$.

Example 9.3.2

Key formulas in Example 7.2.7 are translated as follows. First, we generate 0-arity predicates for each key of name key_C, e.g. key_Table and then the following expressions are added.

- Key $\langle \mathrm{Table}, \{\mathrm{name}, \mathrm{schema}\}\rangle$ is translated into

$$\begin{aligned}&\mathrm{key\_Table} \Leftrightarrow\\ &\quad \forall x, y \ \in \mathrm{Table}, v_1 : \mathrm{String},\ v_2 : \mathrm{Schema}.\\ &\qquad x \neq y \rightarrow \mathrm{name}(x, v_1) \wedge \mathrm{schema}(x, v_2)\\ &\qquad\quad \rightarrow \neg\mathrm{name}(y, v_1) \vee \neg\mathrm{schema}(y, v_2)\end{aligned}$$

- Key $\langle \mathrm{Column}, \{\mathrm{name}, \mathrm{owner}\}\rangle$ is translated into

$$\begin{aligned}&\mathrm{key\_Column} \Leftrightarrow\\ &\quad \forall x, y \ \in \mathrm{Column}, v_1 : \mathrm{String},\ v_2 : \mathrm{Table}.\\ &\qquad x \neq y \rightarrow \mathrm{name}(x, v_1) \wedge \mathrm{owner}(x, v_2)\\ &\qquad\quad \rightarrow \neg\mathrm{name}(y, v_1) \vee \neg\mathrm{owner}(y, v_2)\end{aligned}$$

- Key $\langle \mathrm{Key}, \{\mathrm{name}, \mathrm{owner}\}\rangle$ is translated into

$$\begin{aligned}&\mathrm{key\_Key} \Leftrightarrow\\ &\quad \forall x, y \ \in \mathrm{Key}, v_1 : \mathrm{String},\ v_2 : \mathrm{Table}.\\ &\qquad x \neq y \rightarrow \mathrm{name}(x, v_1) \wedge \mathrm{owner}(x, v_2)\\ &\qquad\quad \rightarrow \neg\mathrm{name}(y, v_1) \vee \neg\mathrm{owner}(y, v_2)\end{aligned}$$

□

In the case of a formula φ^{R}, the formula F declares that top-level relations must hold, and each individual rule is translated into the set of conditions stated by the checking semantics of QVT-Relations, which was explained in Chapter 2, i.e. a relation holds if for each valid binding of variables of the `when` clause and variables of domains other than the target domain, that satisfy the `when` condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of the target domain that satisfies the target domain pattern and `where` condition.

Formally, every rule $\mathrm{Rule} = \langle \mathrm{top}, \mathrm{VarSet}, \mathrm{ParSet}, \mathrm{Pattern}_i\ (i = 1, 2), \mathrm{when}, \mathrm{where} \rangle \in \varphi^{\mathrm{R}}$ is translated into:

- a predicate $Rule : T_1 \times ... \times T_n \in P$ with $\mathrm{ParSet} = \{T_1, .., T_n\}$, and a predicate Top_Rule without parameters (only if $\mathrm{top} = true$), naming the formula
- a formula $\forall v_1 : T_1, ..., v_n : T_n.\ Rule(v_1, ..., v_n) \Leftrightarrow F$ such that $Rule(v_1, ..., v_n)$ is the predicate defined before. In the case of a top rule, there is also a formula $Rule \Leftrightarrow F$. For the formula F there are two cases corresponding to the checking semantics of QVT-Relations:

  1. If $\mathrm{WhenVarSet} = \emptyset$

$$\begin{aligned} &\forall\, x_1, ..., x_n \in\ (\mathrm{VarSet} \backslash \mathrm{2\_VarSet}) \backslash \mathrm{ParSet}.\ (\Phi(\mathrm{Pattern}_1) \rightarrow \\ &\quad \exists\, y_1, ..., y_m \in \mathrm{2\_VarSet} \backslash \mathrm{ParSet}.\ (\Phi(\mathrm{Pattern}_2)\ \wedge\ \Phi(\mathrm{where}))) \end{aligned}$$

  2. If $\mathrm{WhenVarSet} \neq \emptyset$

$$\begin{aligned} &\forall\, z_1, ..., z_o \in\ \mathrm{WhenVarSet} \backslash \mathrm{ParSet}.\ (\Phi(\mathrm{when}) \rightarrow \\ &\quad \forall\, x_1, ..., x_n \in\ (\mathrm{VarSet} \backslash (\mathrm{WhenVarSet} \cup \mathrm{2\_VarSet})) \backslash \mathrm{ParSet}. \\ &\qquad (\Phi(\mathrm{Pattern}_1) \rightarrow \\ &\qquad\quad \exists\, y_1, ..., y_m \in \mathrm{2\_VarSet} \backslash \mathrm{ParSet}. \\ &\qquad\qquad (\Phi(\mathrm{Pattern}_2)\ \wedge\ \Phi(\mathrm{where})))) \end{aligned}$$

The translation $\Phi(\mathrm{Pattern}_i)\ (i = 1, 2)$ of $\mathrm{Pattern}_i = \langle E_i, A_i, Pr_i \rangle$ is the formula

$$\bigwedge r_2(x, y) \wedge \Phi(Pr_i)$$

such that $r_2(x, y)$ is the translation of predicate $p = \langle r_1 : C, r_2 : D \rangle$ for every $rel(p, x, y) \in A_i$ with $x : C, y : D$; and $\Phi(Pr_i)$ is the translation of the $\mathcal{I}^{\mathrm{E}}$-formula into CASL. Moreover, the translation $\Phi(\mathrm{when})$ of $\mathrm{when} = \langle \mathrm{when}_c, \mathrm{when}_r \rangle$ is the formula

$$\bigwedge Rule(v) \wedge \Phi(\mathrm{when}_c)$$

such that $Rule(v)$ is the parametric invocation of the rule $(\mathrm{Rule}, v) \in \mathrm{when}_r$, and $\Phi(\mathrm{when}_c)$ is the translation of the $\mathcal{I}^{\mathrm{E}}$-formula into CASL. The translation $\Phi(\mathrm{where})$ is similar.

Example 9.3.3

For each rule in Example 7.2.7 there is a predicate defining the rule, as follows:

- Top_PackageToSchema
- PackageToSchema : Package × Schema
- Top_ClassToTable
- ClassToTable : Class × Table
- AttributeToColumn : Class × Table × String

The top and non-top versions of `PackageToSchema` are translated into the formulas stating that for each package there must be a schema with the same name.

$$\begin{array}{l}\mathrm{Top\_PackageToSchema} \Leftrightarrow \\ \quad \forall\, p : \mathrm{Package}, pn : \mathrm{String}. \\ \quad\quad \mathrm{name}(p, pn) \rightarrow \exists\, s : \mathrm{Schema}.\ \mathrm{name}(s, pn)\end{array}$$

$$\begin{array}{l}\forall p : \mathrm{Package}, s : \mathrm{Schema}.\ \mathrm{PackageToSchema}(p, s) \Leftrightarrow \\ \quad \forall\, pn : \mathrm{String}. \\ \quad\quad \mathrm{name}(p, pn) \rightarrow \mathrm{name}(s, pn)\end{array}$$

Notice the difference between these two versions: in the second case the variables p and s are in the set of parameters ParSet. Moreover, the when and where clauses are empty, thus they trivially hold (i.e. true formulas).

The non-top version of `ClassToTable` is translated into a formula stating that the relation holds if for every package and schema satisfying the relation `PackageToSchema`, there is a persistent class within that package and a table in the corresponding schema with the same class name, a column within such table which is defined as a default key, and the attributes and columns of both must be in the relation `AttributeToColumn`.

$$\forall c : \text{Class}, t : \text{Table}.\ \text{ClassToTable}(c,t) \Leftrightarrow$$
$$\forall\, p : \text{Package}, s : \text{Schema}.\ \text{PackageToSchema}(p,s) \rightarrow$$
$$\forall\, cn : \text{String}, prefix : \text{String}.$$
$$\text{namespace}(c,p) \wedge \text{name}(c,cn) \wedge \text{kind}(c, Persistent) \rightarrow$$
$$\exists\, cl : \text{Column}, k : \text{Key}.$$
$$\text{schema}(t,s) \wedge \text{name}(t,cn) \wedge \text{column}(t,cl) \wedge$$
$$\text{key}(t,k) \wedge \text{name}(cl, TID) \wedge \text{type}(cl, NUMBER) \wedge$$
$$\text{name}(k, PK) \wedge \text{column}(k,cl) \wedge$$
$$\text{AttributeToColumn}(c,t,prefix) \wedge prefix = EMPTY$$

$$\begin{array}{rcl} \text{Pattern}_1 & = & \langle E_1, A_1, Pr_1 \rangle \text{ in } \texttt{ClassToTable} \text{ with} \\ E_1 & = & \{\text{c}, \text{p}\} \\ A_1 & = & \{rel(\langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier} \rangle, \text{p}, \text{c})\} \\ Pr_1 & = & \texttt{name(c,cn) AND kind(c,Persistent)} \end{array}$$

was translated into the formula $\text{namespace}(c,p) \wedge \rho^{Sen}(Pr_1)$ since $\text{namespace}(c,p)$ is the translation of the predicate $pr = \langle \text{namespace} : \text{Package}, \text{elements} : \text{Classifier} \rangle$ for the relation $rel(pr,p,c) \in A_i$, and $\rho^{Sen}(Pr_1)$ was trivially translated since we are using $FOL^{=}$ as the expressions language. Moreover, `AttributeToColumn(c,t,prefix)` is the parametric invocation of the rule $(\texttt{AttributeToColumn}, \{c,t,prefix\})$ in the where clause.

□

For each signature morphism $\langle \sigma_1^{\text{M}}, \sigma_2^{\text{M}} \rangle$, there is a signature morphism which is the disjoint union of the translations of σ_i^{M}. The result is a signature morphism since the translations of σ_i^{M} preserve the subsort relation, as well as the overloading relations and the symbols used for embedding, projection, and membership. For each morphism between formulas in $\mathcal{I}^{\text{Q}^+}$, i.e. the canonical application of a signature morphism to every element in the formula, there is a morphism between formulas in $\mathcal{I}^{\text{C}^{\text{Th}}}$ which is the canonical application of the translated signature morphism to every element in the translated formula. Moreover, both signature morphisms are theory morphisms.

Lemma 9.3.4. *Given $\mathcal{I}^{\text{Q}^+}$ signatures Σ_i, and a theory morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$, its translation $\Phi(\sigma)$ is a $\mathcal{I}^{\text{C}}$ theory morphism.*

Proof sketch. Reasoning in the same way as in Lemma 9.2.4, we have that given a set of Σ_1-formulas Ψ, and a set of Σ_2-formulas Ψ_2 they comply with the theory morphism $\sigma : \langle \Sigma, \Psi \rangle \rightarrow \langle \Sigma_2, \Psi_2 \rangle$ if $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$. In such case we have a pair $\mathcal{M}$ of interpretations satisfying Ψ_2, i.e. determining concrete SW-models which satisfy the multiplicity constraints and key constraints; which also satisfies $\sigma(\Psi)$, i.e. these formulas are derivable multiplicity constraints, same structured SW-models with potentially less types, or the same key constraints. The translation Φ maps types and roles in a consistent way, and also adds

axioms constraining the $\mathcal{I}^{\mathrm{C}}$ models. In this sense, the pair of models satisfying $\Phi(\Psi_2)$ and $\Phi(\sigma)(\Phi(\Psi))$ must have the same structure than in the original interpretation $\mathcal{M}$. Thus, we have that $\Phi(\Psi_2) \models_{\Phi(\Sigma_2)} \Phi(\sigma)(\Phi(\Psi))$ also hold. If we consider now transformation rules, we have that such rules does not constraint each individual interpretation but the relation between them. Reasoning as before, rules in Ψ_2 can be more specific than those in Ψ, such that any pair of interpretations satisfying the relation defined by those rules in Ψ_2 also satisfies the relation defined by $\sigma(\Psi)$. Since the translation Φ also constraints the pair of models with respect to the transformation rules, we have that $\Phi(\Psi_2) \models_{\Phi(\Sigma_2)} \Phi(\sigma)(\Phi(\Psi))$ also hold with the inclusion of transformation rules. Thus, $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.

□

We can prove that Φ is indeed a functor. Complete proof is given in Appendix E.

Lemma 9.3.5. *The function* $\Phi : Th^{\mathcal{I}^{\mathrm{Q}^+}} \rightarrow Th^{\mathcal{I}^{\mathrm{C}}}$ *is a functor from the category of* $\mathcal{I}^{\mathrm{Q}^+}$ *theories and theory morphisms to the category of theories in* $SubPCFOL^{=}$.

Proof sketch. Using the fact that signature morphisms are defined as the disjoint union of extended CSMOF signature morphisms, in which composition is preserved, and that the translation is defined componentwise, we can conclude that the functor preserves the composition of signature morphisms. Reasoning in the same way, we have that the translated signature morphism preserves the identities in SubPCFOL=. Thus, Φ is a functor.

□

We need now to define how the natural transformation β is defined, i.e. how $\mathcal{I}^{\mathrm{C}}$ models and homomorphisms are translated into $\mathcal{I}^{\mathrm{Q}^+}$ interpretations and homomorphisms.

Given a $\mathcal{I}^{\mathrm{Q}^+}$ theory $T = \langle \Sigma, \Psi \rangle \in \mathrm{Th}^{\mathcal{I}^{\mathrm{M}^+}}$, a model M of its translated theory (Σ', E) is translated into a Σ-model $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}}, \mathcal{M}_2^{\mathrm{M}} \rangle$ by constructing disjoint models with an interpretation of elements for each corresponding $\mathcal{I}^{\mathrm{M}^+}$ theory. Each $\mathcal{M}_i^{\mathrm{M}}$ $(i = 1, 2)$ is defined as in Section 9.2.

Moreover, given two models of its translated theory (Σ', E), and let $h' : \mathcal{M} \rightarrow \mathcal{M}'$ be a Σ'-homomorphism. Let us denote $\mathcal{N} = \rho^{Mod}(\mathcal{M})$ and $\mathcal{N}' = \rho^{Mod}(\mathcal{M}')$ and let us define $h : \mathcal{N} \rightarrow \mathcal{N}'$ as the disjoint translation of h', as in Section 9.2, with respect to the elements in the corresponding signatures. The disjoint union is a homomorphism since each translated function is a homomorphism.

We can prove that β is indeed a natural transformation (complete proof in Appendix E).

Lemma 9.3.6. *The function* $\beta : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}} \to \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{Q}^+}}$*, with* $\boldsymbol{Mod} : Th \to \boldsymbol{Cat}$ *the functor giving the category of models of a theory, is a natural transformation, i.e. a family of arrows* $\beta_A : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}(A) \Rightarrow \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{Q}^+}}(A)$*, one for each theory* A *of* $\boldsymbol{Sign}^{\mathrm{Q}}$*, such that, for every theory morphism* $\sigma : A \to B$ *it holds:* $\boldsymbol{Mod}^{\mathcal{I}^{\mathrm{Q}^+}} \circ \beta_B = \beta_A \circ (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}$

Proof sketch. Given any model in $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(B))$, the translation β_B gives a pair of disjoint models, and the application of $\mathbf{Mod}^{\mathcal{I}^{\mathrm{Q}^+}}$ gives the same pair of models. In the other side, the reduct $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(\sigma))$ gives an interpretation of symbols of the translated signature A, and the translation β_A just divide the model into two disjoint ones with respect to the two parts of the signature A. Thus, the property holds. In the case of homomorphisms the property also holds, since the translation of a homomorphism is defined as a disjoint translation with respect to the elements in the corresponding signatures, and the reduct of a homomorphism is defined componentwise. Finally, β is a natural transformation.

□

Since each model of a $\mathcal{I}^{\mathrm{M}^+}$ theory has a correspondent model in the translated $\mathcal{I}^{\mathrm{C}}$ theory, as proved in Section 9.2, the property holds for the disjoint union of models. Thus, the comorphism also admits model expansion, and thus borrowing of entailment for theories.

9.4 Related Work

There are works representing the semantics of UML class diagrams with first-order logic, as in [SZ09]. Since there are no so many alternatives for this representation, these works have similarities with our representation of extended CSMOF into CASL. In particular, the work in [SZ09] is the nearest to ours from which we take many aspects, e.g. the "distinguishability" and "completeness of elements" axioms. In [TBHW99] the authors explain how class diagrams with OCL constraints can be translated into CASL. However, their definition is informally presented, and not in terms of a comorphism. In [JKMR12] the authors define a comorphism from UML class diagrams with rigidity constraints to ModalCASL (an extension of CASL). Since our $\mathcal{I}^{\mathrm{M}^+}$ institution is an adaptation of the institution for UML class diagrams, the comorphisms have some aspects in common, as the translation of formulas, but without the modal logic particularities. In [Fav09] the authors present a formalization of metamodels in the NEREUS language and they explain how this formalization is translated into CASL. The final representation differs a lot from our, since for example each class and association is represented as a CASL specification. Although ATL is another transformation language, it is somehow related to QVT-Relations since it was originally developed to answer the QVT Request For Proposal. In this sense, the mappings from ATL to constructive type theory [CLST10b] and first-order logic [BEC12] have some similarities with our mapping since they are also expressed as $\forall\exists$-formulas following the standard checking semantics.

10 Tool Support with HETS

In this chapter we detail the implementation of a prototype of our environment using the Heterogeneous Tools Set (HETS, [MML07, Mos05]), i.e. an implementation of the extended CSMOF and QVTR institutions introduced in Chapter 8, and of the generalized theoroidal comorphisms defined in Chapter 9. All these together give us the possibility of performing the heterogeneous verification of a model transformation, as described in Chapter 5. The prototype is already included in the HETS distribution that can be found in [Mos13b].

In Section 10.1 we present HETS and how institutions and comorphisms are handled within the Tool, and in Section 10.2 we give details on the implementation of our environment. Then, in Section 10.3 we present how the environment works by illustrating the main concepts with the example introduced in Chapter 2. Finally, in Section 10.4 we explore the capabilities of the environment with respect to verification aspects introduced in Chapter 3.

10.1 Heterogeneous Tools Set

As introduced in [MML07, Mos05], HETS is an open source software providing a general framework for formal methods integration and proof management. HETS acts like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations.

Based on the Theory of Institutions, HETS supports a variety of different logics (defined as institutions), as shown in Figure 10.1. Some of them are: CASL, an extension of many sorted first-order logic with partial functions and subsorting; CoCASL [MSRR06], a coalgebraic extension of CASL, suited for the specification of process types and reactive systems; ModalCASL [Mos13c], an extension of CASL with modalities; Haskell [Jon02], a pure and strongly typed functional programming language; ISABELLE [NPW02], an interactive theorem prover for higher-order logic; and Maude [CDE+02], a rewriting system for first-order

logic. Various of these logics have tool support, e.g. the automated theorem proving system SPASS [WBH+02] for first-order logic with equality.

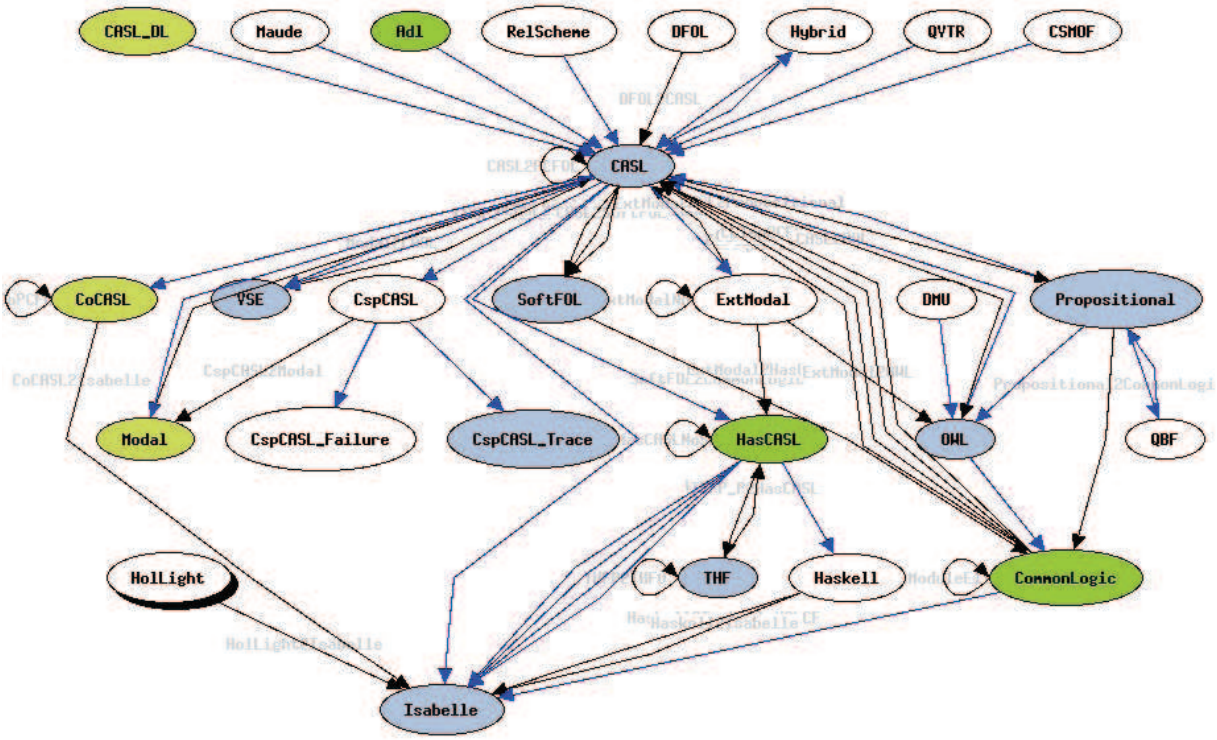


Figure 10.1: Almost complete logic graph of HETS

The architecture of HETS is depicted in Figure 10.2. HETS consists of logic-specific tools for the parsing and static analysis of basic theories written in the different involved logics. The internal representation of a Logic is provided by a Haskell multiparameter type class `Logic` with functional dependencies [JJM97]. This class allows representing signatures, signature morphisms, sentences, abstract syntax of basic specifications etc., and functions for parsing, printing, static analysis, and proving. Proof support for the other logics can be obtained by using logic translations by means of implementing comorphisms within the logic graph.

For heterogeneous specifications there is the heterogeneous CASL (HetCASL, [Mos13a]) language, which allows to combine and rename specifications (written in any logic of HETS), hide parts thereof, and also translate them to other logics. The logic graph is flattened into a Grothendieck logic (in Haskell) which is formally based on a Grothendieck institution [Dia02], with a proof calculus for heterogeneous specifications providing HetCASL with a sound basis. There are also generic heterogeneous tools for parsing and static analysis of heterogeneous specifications that call the logic-specific tools whenever is required.

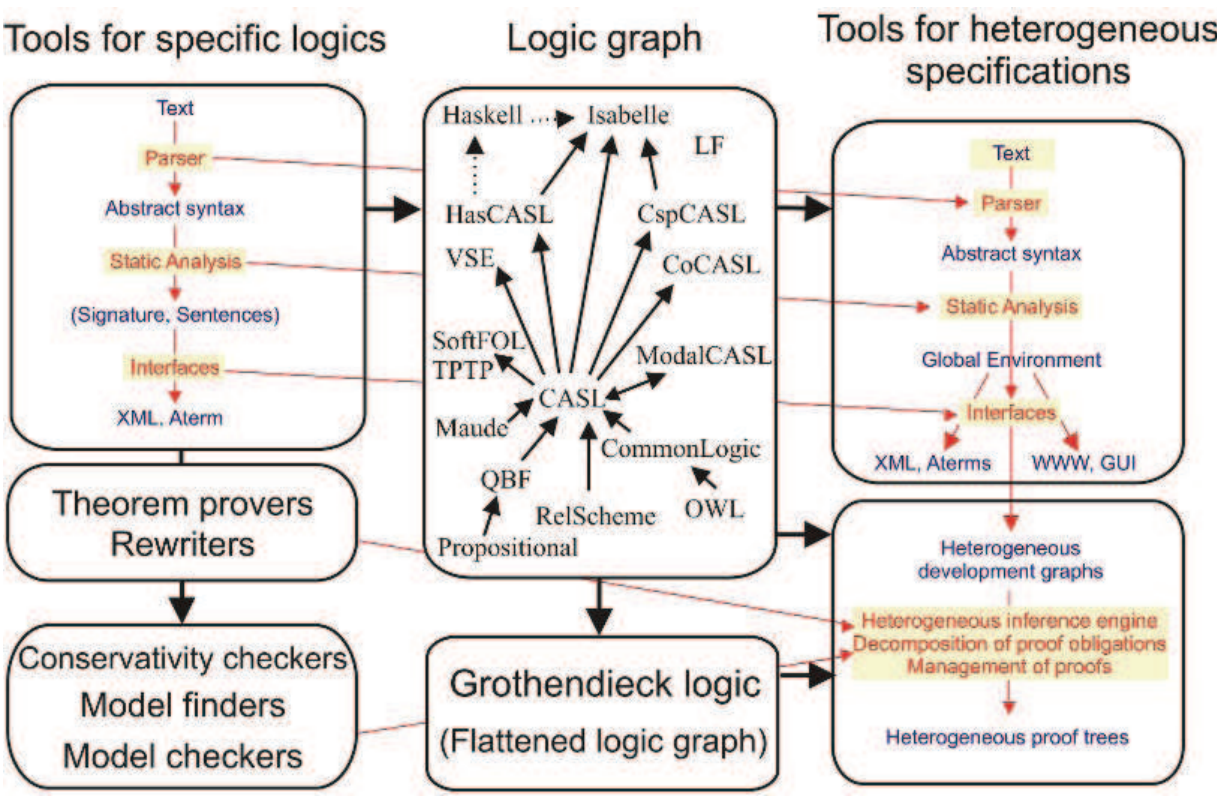


Figure 10.2: Architecture of HETS

HETS provides proof management capabilities for monitoring the overall correctness of a heterogeneous specification. A heterogeneous specification is represented in a development graph [MAH06], as the one in Figure 10.4. A development graph consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called definition links, indicating the dependency of each involved structured specification on its subparts. The proof calculus for development graphs allows for decomposing global theorem links into simpler ones, until eventually local implications are reached. The latter can be discharged using a logic-specific calculus as given by an entailment system. The use of HETS will be explained in Section 10.3 for the running example.

10.2 Implementation of the Environment

The extended CSMOF and QVTR institutions are implemented within HETS.

In the case of CSMOF, the abstract syntax of the language is a direct Haskell representation of classes and relations in Figure 10.3. The simplified version of MOF already depicted in Figure 2.3, is supplemented with the notion of a metamodel containing named elements, and associated to many SW-models. A SW-model is a collection of objects referencing types in the metamodel, and of links between objects corresponding to properties in the metamodel.

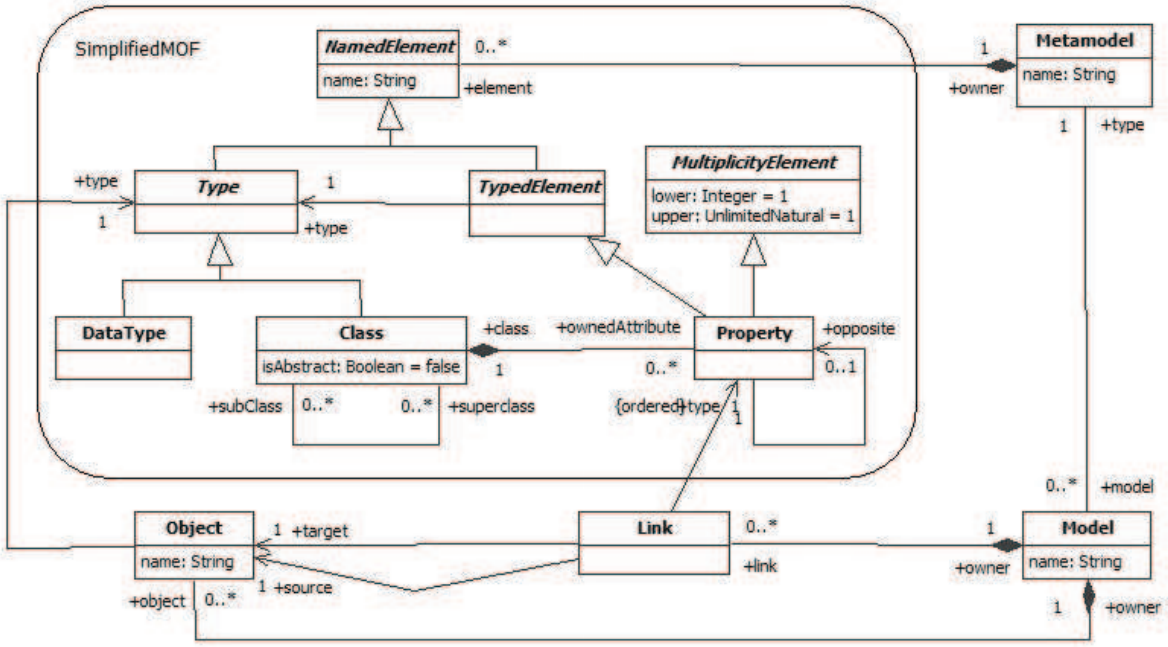


Figure 10.3: Class-based representation of CSMOF

The following types correspond to the representation of metamodels and SW-models. We note that the abstract syntax of any logic which must be included in HETS can be automatically derived from a XMI file specifying its metametamodel (e.g. as the one in Figure 10.3), as we will discuss in Chapter 11.

```
data Metamodel = Metamodel
                 { metamodelName :: String
                 , element :: [NamedElement]
                 , model :: [Model]
                 } deriving (Eq, Ord)

data Model = Model
             { modelName :: String
             , object :: [Object]
             , link :: [Link]
             , modelType :: Metamodel
             } deriving (Eq, Ord)
```

The abstract syntax definition is used by a Haskell parser, based on the Text.XML.Light library, which takes a XMI instance of the metametamodel in Figure 10.3 and constructs an instance of these types. XMI files conform to a schema defined using Ecore (Eclipse Modeling Framework [FSM+03]) as the metametamodel language, instead of MOF. This decision was taken since in the context of this thesis the MediniQVT [IKV13] tool was used for executing QVT-Relational transformations, and it includes an Ecore editor. The Ecore definition of CSMOF can be found in Section F.1.

We also have a static analyzer which takes the Haskell type instances, performs some static checking and if it is right, returns a concrete CSMOF theory. In the following code there is an excerpt of the representation of a CSMOF theory following the definitions of signatures (`Sign`) in Section 6.2 and extended formulas (`Sen`) in Section 8.2.

```
data Sign = Sign { types :: Set.Set TypeClass
                 , typeRel :: Rel.Rel TypeClass
                 , abstractClasses :: Set.Set TypeClass
                 , roles :: Set.Set Role
                 , properties :: Set.Set PropertyT }

data Sen = MultConstrSen { multConstr :: MultConstr }
         | SWModelSen { sModel :: SWModel }

data MultConstr = MultConstr { getType :: TypeClass
                             , getRole :: Role
                             , cardinality :: Integer
                             , constraintType :: ConstraintType }

data ConstraintType = EQUAL | LEQ | GEQ

data SWModel = SWModel { instances :: Map.Map String TypeClass
                       , links :: Set.Set LinkT }
```

Finally, we provide an instance of the type class `Logic` which is defined as follows. Note that this class includes the definition of the abstract syntax (`Metamodel`), the signature (`Sign`), the formulas (`Sen`) and signature morphisms (`Morphism`). After some configuration of HETS, the logic graph includes a node with our CSMOF logic, as shown in Figure 10.1.

```
instance Logic CSMOF
  ()                -- Sublogics
  Metamodel         -- basic_spec
  Sen               -- sentence
  ()                -- symb_items
  ()                -- symb_map_items
  Sign              -- sign
  Morphism          -- morphism
  ()                -- symbol
  ()                -- raw_symbol
  ()                -- proof_tree
  where
    stability CSMOF = Experimental
    empty_proof_tree _ = ()
```

We do the same in the case of QVTR, but in this case we have as the abstract syntax a simplified version of the metamodel in Figure 2.5. In the following code, there is an excerpt of the definition of a transformation and of a relation. Note that the transformation includes a source and a target metamodel, using the same Haskell type as for CSMOF.

```
data Transformation = Transformation
            { transfName :: String
            , sourceMetamodel :: (String,String,CSMOF.Metamodel)
            , targetMetamodel :: (String,String,CSMOF.Metamodel)
            , keys :: [Key]
            , relations :: [Relation]
            } deriving (Eq, Ord)

data Relation = Relation
            { top :: Bool
            , relName :: String
            , varSet :: [RelVar]
            , primDomains :: [PrimitiveDomain]
            , sourceDomain :: Domain
            , targetDomain :: Domain
            , whenCond :: Maybe WhenWhere
            , whereCond :: Maybe WhenWhere
            } deriving (Eq, Ord)
```

The abstract syntax definition is used by a Haskell parser, based on the Parsec library, which takes a `.qvt` file and constructs an instance of these types. As before, we also have a static analyzer which returns a QVTR theory. In the following code there is an excerpt of the representation of the QVTR theory following the definitions of signatures (`Sign`) in Section 7.2 and extended formulas (`Sen`) in Section 8.2. Also note that the signature and formulas contain a reference to CSMOF signatures and formulas, respectively.

```
data Sign = Sign { sourceSign :: CSMOF.Sign
                 , targetSign :: CSMOF.Sign
                 } deriving (Show, Eq, Ord)

data Sen = KeyConstr { keyConst :: Key }
         | QVTSen { rule :: RelationSen }
         | CSMOFSen { mofSen :: CSMOF.Sen }
         deriving (Show, Eq, Ord)
```

Finally, we provide an instance of the type class `Logic` which is defined as follows. As before, this class includes the definition of the abstract syntax (`Transformation`), the signature (`Sign`), the formulas (`Sen`) and signature morphisms (`Morphism`). After some configuration of HETS, the logic graph includes a node with our QVTR logic, as shown in Figure 10.1.

```
instance Logic QVTR
  ()                -- Sublogics
  Transformation    -- basic_spec
  Sen               -- sentence
  ()                -- symb_items
  ()                -- symb_map_items
  Sign              -- sign
  Morphism          -- morphism
  ()                -- symbol
  ()                -- raw_symbol
  ()                -- proof_tree
  where
    stability QVTR = Experimental
    empty_proof_tree _ = ()
```

The final step is the implementation of the generalized theoroidal comorphisms from extended CSMOF and QVTR to CASL. As shown in the following code (just an excerpt), we instantiate the type class `Comorphism`. This instance has the elements defined in the instance of `Logic` for both, CSMOF and CASL. The most important part is the implementation of the translation functions, in particular `mapTheory` which takes an CSMOF theory and constructs a CASL theory. This function is called each time the comorphism is applied to a CSMOF specification. The definition of the instance for the comorphism from QVTR to CASL is straightforward.

```
instance Comorphism CSMOF2CASL
    CSMOF.CSMOF
    ()
    CSMOFAs.Metamodel
    CSMOF.Sen
    ()
    ()
    CSMOF.Sign
    CSMOF.Morphism
    ()
    ()
    ()
    CASL
    CASL_Sublogics
    CASLBasicSpec
    CASLFORMULA
    SYMB_ITEMS
    SYMB_MAP_ITEMS
    CASLSign
    CASLMor
    C.Symbol
    C.RawSymbol
    ProofTree
    where
      sourceLogic CSMOF2CASL = CSMOF
      sourceSublogic CSMOF2CASL = ()
      targetLogic CSMOF2CASL = CASL
      map_theory CSMOF2CASL = mapTheory
      has_model_expansion CSMOF2CASL = True
```

Once HETS is executed, the logic graph includes arrows from the nodes representing our CSMOF and QVTR logics to the node representing the CASL logic, as shown in Figure 10.1.

10.3 How the Environment Works

Our problem is stated as a heterogeneous specification (in a `.het` file) using CASL structuring constructs [CoF04]. Within such specifications it is possible to use references to other logics by using the keyword `logic`. We have at least three logics: `CASL`, `CSMOF` and `QVTR`. When a logic is called, HETS uses logic-specific tools for parsing and static analysis. We also perform logic translations through the implemented generalized theoroidal comorphisms which are `CSMOF2CASL` and `QVTR2CASL`. Next, there is an excerpt of the heterogeneous specification of the example.

```
logic CSMOF                                                          (1)

from QVTR/UML get UML |-> UMLMetamodel                               (2)
from QVTR/UML_WMult get UML |-> UMLConstraints

spec UMLProof = UMLMetamodel                                         (3)
then %implies
  UMLConstraints
end

logic QVTR                                                           (4)

from QVTR/uml2rdbms get uml2rdbms |-> QVTTransformation              (5)

logic CASL                                                           (6)

spec ModelTransformation =
     QVTTransformation with logic QVTR2CASL                          (7)
then %implies
  . key_Table
  . key_Column
  . key_Key
  . Top_PackageToSchema
  . Top_ClassToTable
end

spec MoreProofs = UMLMetamodel with logic CSMOF2CASL                 (8)
then %implies
  forall x,y : Classifier; str : String
  . name(x,str) /\ name(y,str) => x = y
end
```

Within the CSMOF logic (1) we create two CSMOF-theories (specifications) from XMI files (2) with the information of the class metamodel in Figure 2.6 and the class SW-model in Figure 2.7. This implies the creation of a Haskell representation of signatures and formulas according to the extended institution defined in Chapter 8. Both theories only differ in the formulas, i.e. the first theory UMLMetamodel does not have any multiplicity constraint. The same is done with the rdbms information, which is not shown in this example. For convenience, we have defined an integrated XMI schema containing both metamodel and SW-model information. However, it is possible to automatically generate this file from different formats used in UML modeling environments. Another specification is created (3) by extending UMLMetamodel and stating that UMLConstraints is implied. This means that every formula (multiplicity constraint) in the second specification can be derived, thus there must be a proof of it. This is how the satisfaction relation of the CSMOF institution is checked. Notice that for developing the proof, the comorphism CSMOF2CASL (or any other if defined) must be called since CSMOF does not have any specific proof system.

We also use the QVTR logic (4) to create a specification (5) from a standard `.qvt` file according to the extended institution defined in Chapter 8. The model transformation is specified using the same language defined within the QVT standard (e.g. as the one in the running example). The only difference is that instead of using OCL as the expressions language, we use for now a very simple FOL-based language containing boolean connectives, constants true and false, term equality, strings and variables. This step also loads the XMI files containing the information of source and target metamodels for constructing the signature of the QVTR institution, as done in (2). We use the name of the source and target metamodel in the transformation specification for finding the corresponding files. Finally, we move into CASL, through the comorphism QVTR2CASL, (6) for creating another specification (7) in which the translation of key and rule formulas (represented as propositions) defined in Chapter 7 are implied by the transformation specification. As in (3), for each implied formula, a proof must be given.

We can also translate our specifications and complement them with other constraints (8) which cannot be stated as formulas of the former institutions. As an example we can state that there cannot be two Classifiers with the same name in the UMLMetamodel specification. For this purpose we are using the CSMOF2CASL comorphism. Again, as in (3), a proof of it must be given. Notice that that we can use any other logic within the logics graph of HETS through existing comorphisms. This improves the proof capabilities of our environment.

When the comorphism CSMOF2CASL is called, the CSMOF theory is translated according to the comorphism specified in Section 9.2. The translation of the running example gives the following CASL theory (in CASL syntax), as defined in Example 9.2.1 and in Example 9.2.2. The complete code is in Appendix F.

```
sorts Class, PrimitiveDataType < Classifier;
      Attribute, Package, String

generated type UMLModelElement ::=
  sort Attribute | sort Classifier | sort Package

op Package : String
op Persistent : String
...

op a : Attribute
op c : Class
op p : Package
op pdt : PrimitiveDataType

pred attribute : Class * Attribute
pred name : UMLModelElement * String
pred elements : Package * Classifier
pred namespace : Classifier * Package
```

```
...

forall x : Package; y : Classifier
  . elements(x, y) <=> namespace(y, x)

%{ completeness of relations }%
forall x : Package; y : Classifier
  . elements(x, y) <=> (x = p /\ y = c) \/ (x = p /\ y = pdt)

%{ completeness of elements }%
forall x : Classifier . x = c \/ x = pdt

%{ distinguishability }%
. not a = c /\ not a = p /\ not a = pdt /\
  not c = p /\ not c = pdt /\ not p = pdt
```

Notice that there is a sort for each type and a subsorting relation according to the hierarchy relation between types. The abstract class UMLModelElement is represented as a generated type, i.e. a sort defined together with a sort generation constraint which states that it is the disjoint embedding of it subsorts. Moreover, there is a total function for each element in the SW-model formula (not every string is shown in the example) and a predicate or a pair of predicates for each unidirectional or bidirectional property, correspondingly.

There are also many axioms stating:

- the equivalence of predicates as the case of elements and namespace
- the "completeness of relations" constraint together with the satisfaction of a predicate by the corresponding links in the SW-model formula, as in the case of elements.
- the "completeness of elements" constraint as in the case of the non-abstract class Classifier which has sub-classes, we have an axiom involving the functions with codomains Class and PrimitiveDataType
- the "distinguishability" constraint as in the case of elements within the UMLModelElement hierarchy

The formula $\#(\text{UMLModelElement} \bullet \text{name}) = 1$ is translated into the conjunction of $min(1, \text{name} : \text{UMLModelElement} \times \text{String})$ and $max(1, \text{name} : \text{UMLModelElement} \times \text{String})$, as defined in Example 9.2.3, which is represented in CASL syntax as follows.

```
. (forall x_1 : UMLModelElement
   . exists y_1 : String . name(x_1, y_1))              %Min%
  /\ forall x_1 : UMLModelElement; y_2, y_1 : String
     . (name(x_1, y_2) /\ name(x_1, y_1)) => y_2 = y_1  %Max%
```

When the comorphism `QVTR2CASL` is called, the QVTR signature and formulas are also translated into CASL according to the comorphism specified in Section 9.3. The translation of the running example gives a CASL signature which is the disjoint union of the translations of each CSMOF theory (one of them is the signature presented before in CASL syntax), together with predicates declaring the following transformation rules and keys, as defined in Example 9.3.3 and in Example 9.3.2, respectively.

```
pred Top_PackageToSchema : ()
pred PackageToSchema : Package * Schema
pred Top_ClassToTable : ()
pred ClassToTable : Class * Table
pred AttributeToColumn : Class * Table * String

pred key_Table : ()
pred key_Column : ()
pred key_Key : ()
```

Now is time to translate key and rule formulas. As defined in Example 9.3.2, the key $\langle \text{Table}, \{\text{name}, \text{schema}\}\rangle$ is translated into

```
. key_Table
  <=> forall x_2, x_1 : Table; y_2 : String; y_1 : Schema
      . not x_2 = x_1
        => name(x_1, y_2) /\ schema(x_1, y_1)
           => not name(x_2, y_2) \/ not schema(x_2, y_1)   %
```

In the following code there are the top and non-top versions of `PackageToSchema` and non-top version of `ClassToTable` formulas are translated, as defined in Example 9.3.3, into the following formulas.

```
Top_PackageToSchema <=>
  forall p : Package; pn : String
  . name(p, pn) => exists s : Schema . name(s, pn)

forall p : Package; s : Schema . PackageToSchema(p,s) <=>
  forall pn : String
  . name(p,pn) => name(s,pn)
```

```
forall c : Class; t : Table . ClassToTable(c,t) <=>
  forall p : Package; s : Schema
  . PackageToSchema(p,s) =>
    forall cn : String
    . namespace(c,p) /\ kind(c, Persistent) /\ name(c, cn) =>
      exists cl : Column; k : Key; prefix : String
      . schema(t,s) /\ name(t,cn) /\ column(t,cl) /\ key(t,k) /\
        name(cl,TID) /\ type(cl,NUMBER) /\ name(k,PK) /\ column(k,cl)
        /\ AttributeToColumn(c,t,prefix) /\ prefix = EMPTY
```

The last formula says that the top-level relation holds whether for every package and schema, if they satisfy the relation `PackageToSchema`, it implies that, if there is a persistent class within that package, there must exist a table in the corresponding schema with the same class name, and the attributes and columns of both must be in the relation `AttributeToColumn`. When the proposition `Top_ClassToTable` is called from the CASL specification, a proof of the implication must be given.

Once our heterogeneous specification is processed, HETS constructs a heterogeneous development graph, as the one in Figure 10.4 for the example.

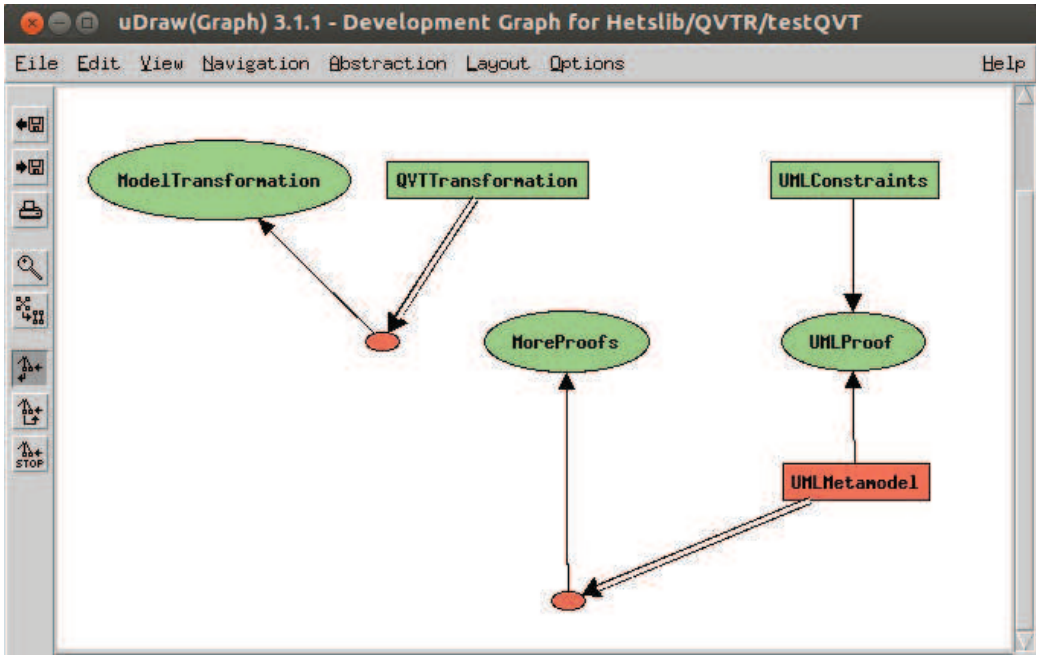


Figure 10.4: Development graph of the example

In such graph, nodes correspond to specifications and red ones correspond to specifications with open proof obligations. In the example we have three proof obligations which corre-

spond to those formulas marked as `%implies` within the specifications. The double arrows are heterogeneous theorem links, meaning that the logic changes along the arrow. In the example this corresponds to the construction of the specifications by extending an existing one which is translated through the comorphism `CSMOF2CASL` and `QVTR2CASL`. In the case of the multiplicity constraints, which are still within the CSMOF institution, some comorphism must be selected in order to perform the proof, e.g. the `CSMOF2CASL` comorphism.

Proof goals can be discharged using a logic-specific calculus. In the example we used the automated theorem proving systems SPASS [WBH+02], as shown in Figure 10.5.

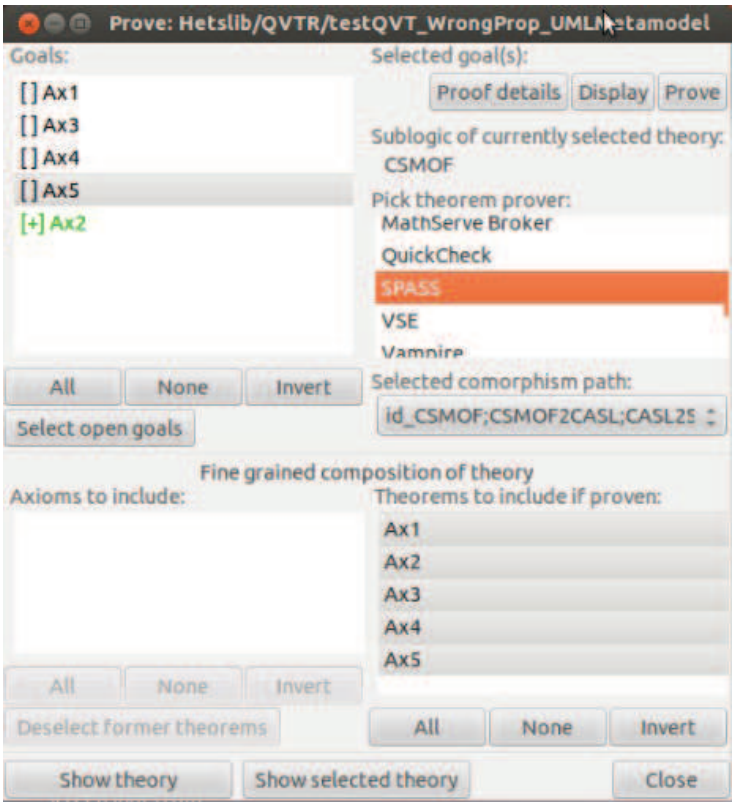


Figure 10.5: HETS selection of a logic-specific tool

The proof window shows all goal names prefixed with the proof status in square brackets. An empty bracket indicates an open proof goal, and a '+' (also painted in green) indicates a proved goal. The bottom right list shows the axioms used for proving a goal, as shown in Figure 10.6.

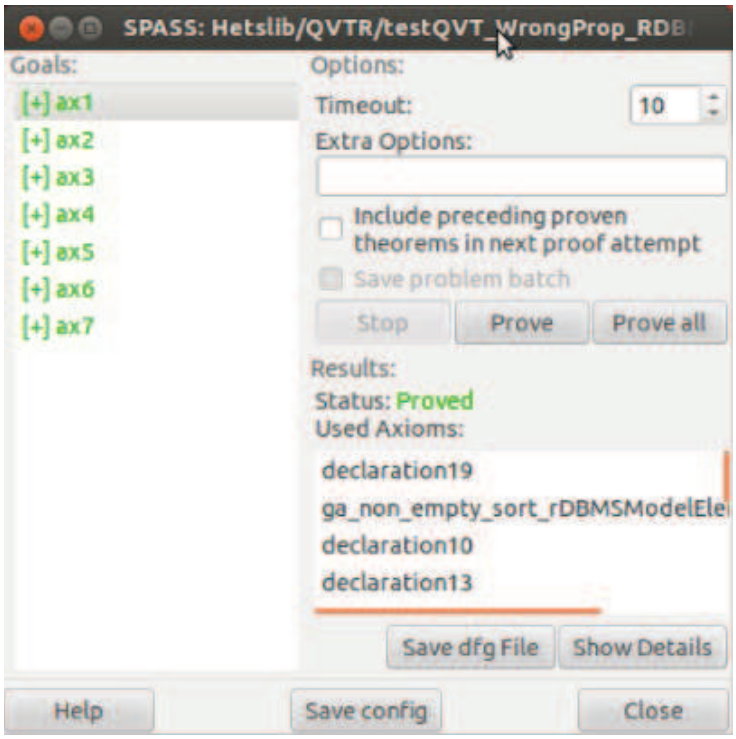


Figure 10.6: Proved goal in HETS

10.4 Verification Properties

There are several properties that can be verified, some of them related to the computational nature of transformations and target properties of transformation languages, and other to the modeling nature of transformations [CS13a], as discussed in Chapter 3.

The minimal requirement is conformance, i.e. that the source and target models (resp. the transformation specification) are syntactically well-formed instances of the source and target metamodels (resp. the transformation language). Our framework provides this verification, first, in the construction of the CSMOF and QVTR theories. Parsing and static analysis checks whether signatures and formulas are well-formed, and as we introduced before, a SW-model within a signature is a structurally well-formed instance of the metamodel in the same signature, as well as a transformation specification given in a formula is well-formed with respect to the signature containing both source and target metamodels. Second, by proving CSMOF formulas which contains the conformance with respect to multiplicity constraints. Finally, when verifying non-structural constraint by extending both CSMOF and QVTR specifications using other logics, as CASL in the example.

One interesting point is that HETS also allows for disproving goals using consistency checkers, provide an additional point of view in the verification process. In this sense, a goal can also be marked as disproved or inconsistent. In particular, an inconsistent set of transformation rule means that there are contradictory conditions which could inhibit the execution of the transformation. A very simple example of this is the following specification in which we state that every package contains only one classifier.

```
spec InvalidProperty = UMLMetamodel with logic CSMOF2CASL
then %implies
  forall x : Package; y,z : Classifier
  . elements(x,y) /\ elements(x,z) => y = z
end
```

This goal can be disproved since in our example, package `p` has two classifiers: class `c` and primitive datatype `pdt`.

In most cases a general-purpose logic, as provided by CASL, is enough to cover most of the verification approaches presented in Chapter 3. The future inclusion of OCL as an institution will provide additional support in this sense, as discussed in Chapter 11. However, the verification process may depend on the problem to verify, since it is well-known that there is a "state explosion" problem when using automated checkers. Thus, automatic proofs are not always possible. As shown in Figure 10.5, in HETS it is possible to choose the tool we want to use. In this sense, we can choose not to use an automated theorem proving system as SPASS, but for example an interactive theorem prover as ISABELLE.

Recall the notion of a transformation model, i.e. a model composed by the source and target metamodel, the transformation specification and the well-formedness rules. A QVTR theory (`QVTTransformation` in the example) is a transformation model which allows to add additional properties by combining elements from the source and target metamodels and SW-models. With this we can state model syntax relations, trying to ensure that certain elements or structures of any input model will be transformed into other elements or structures of the output model (transformation rules are indeed model syntax relations). We can also state model semantics relations, e.g. temporal properties and refinement. Although further work is needed to evaluate the alternatives, there are languages and tools already in HETS, as ModalCASL and VSE [CLMM09] (based on dynamic logics) commonly used for verifying these kind of things.

We could also be interested in working at another abstraction level, i.e. not considering specific SW-models but only metamodels and the transformation specification. This can be useful, for example, for proving that a transformation guarantees some model syntax relations when transforming any valid source SW-model. This is also needed for proving termination and determinism of a transformation: the existence of a target SW-model for any execution and the uniqueness of such SW-model, respectively. In particular, termination and determinism properties can be basically stated as in the following CASL code: for any class model satisfying the source constraints (`Pre`), there must be a relational model (only one in the

case of determinism) satisfying the target constraints (`Post`) and the transformation rules (`Rules`). These kind of problems are hard to be verified automatically since the space of solutions for `ma : ClaM` is almost always infinite.

```
%{ Termination property }%
forall ma : ClaM . Pre ma =>
                     (exists mb : RelM . Rules ma mb /\ Post mb)

%{ Determinism property }%
forall ma : ClaM . Pre ma =>
                     (exists! mb : RelM . Rules ma mb /\ Post mb)
```

The problem here is that we need another institutional representation, somehow related to [CK08] in which our CSMOF institution is based, in which models of the institution are not constrained by the SW-model within the signature. Moreover we need to generate as part of the comorphism an abstract representation of a SW-model (e.g. `ClaM` in the example). This is subject of future work, as discussed in Chapter 11.

11 Conclusions and Further Research

In this thesis we introduced a unified environment that allows formal verification of a model transformation using heterogeneous verification approaches. This chapter presents our conclusions and perspectives of further research. The main results and contributions of our work are summarized in Section 11.1. Section 11.2 closes with several discussions and open problems that promote further research.

11.1 Summary and Contributions

The idea of developing a unified environment for verification was motivated by the comprehensive literature review on the verification of model transformations we conducted. In this review, summarized in Chapter 3, we found that there are many properties of interest addressed by the verification of a model transformation. These properties can be classified in language-related and transformation-related properties, the first ones referring to the computational nature of transformations and target properties of transformation languages, and the second ones referring to the modeling nature of transformations. With respect to verification approaches, we found that almost any traditional verification technique can be applied in MDE. These techniques can also be classified in different orthogonal categories referring to: the kind of technique used for verification, the abstraction level with respect to the elements involved in the transformation, the abstraction level with respect to the implementation of the transformation, and the dependency/independence with respect to the transformation specification. A contribution that differentiates our review from the others is that we illustrated by means of examples how the verification of each kind of property can be addressed by many approaches.

This last result led us to the definition of a unified environment for the heterogeneous verification of model transformations, introduced in Chapter 5. In this environment the MDE elements involved in a model transformation (SW-models, metamodels and the transformation itself) are defined without depending on any specific logical domain, and there are semantic-preserving translations from these elements to several other logical domains where the verification of desired properties can be addressed. This idea is innovative since in most of the cases the MDE elements are directly represented within some specific logical domain which makes their translation into another domain expensive. In our case we do not need to maintain multiple formal representations of the same MDE elements, but to define those translations which can be then reused. These translations are also useful to integrate MDE elements with the specification and verification of other software artifacts in a traditional software development. Although we build on the MOF and QVT-Relations standards for the specification of the MDE elements, we follow an idea general enough to be extended to other transformation approaches and languages.

The definition of such environment was feasible due to the Theory of Institutions which provided solid foundations to achieve our goals. We have provided institutions for the structural conformance relation between SW-models and metamodels specified with a simplified version of MOF (which we called CSMOF), introduced in Chapter 6, and for QVT-Relations check-only unidirectional transformations (which we called QVTR), introduced in Chapter 7, both extended in Chapter 8. These institutions provide a generic representation of the MDE elements with formal semantics covering almost every important element. These definitions not only serve as a basis for our environment but also improve existent knowledge on the semantics of MDE and on the use of institutions for the formalization of specification languages. Indeed, those representations could be connected with other UML languages to conform the heterogeneous institution environment in the sense of [CKTW08]. We have not considered some aspects that are not commonly used, e.g. uniqueness and ordering properties within a property end. Beyond that, the most important aspect that we left apart is the inclusion of OCL. Instead we considered a generic institution used as an expressions language within transformations, which can be instantiated with, for example, an institution for first-order logic. With this decision we are not losing expressive power.

We also provided generalized theoroidal comorphisms from our institutions to CASL (an extension of many sorted first-order logic with partial functions and subsorting), defined in Chapter 9. This connection allowed us to develop a functional prototype of the environment supported by HETS, introduced in Chapter 10, providing an implementation for institutions and comorphisms. Within this prototype, MDE experts can specify model transformations in their technological space and such specifications can be complemented by verification experts with other properties to be verified, e.g. non-structural constraints. All this information is taken by HETS, which performs automatic translations of proof obligations into other logics and allows selecting the corresponding prover to be used, whilst a graphical user interface is provided for visualizing the whole proof. In other words, we provided to MDE practitioners the "glue" they need for connecting their technological space with the logical domains needed for verification. Another contribution is that there was no evidence about the using of the

Theory of Institutions in practice for the verification of model transformations. Moreover, the existent connections between CASL and other logics, broadens the spectrum of logical domains in which the verification of model transformations can be addressed.

11.2 Discussions and Open Problems

Throughout the development of this thesis we have identified several aspects that deserve discussion. These aspects cover the different alternatives for the definition of the institutions and comorphisms, the definition of an institution for OCL, the relation between model transformations and comorphisms, and the improvement of the environment. A discussion of these aspects introduces possible directions of future research.

11.2.1 On the Formal Definition of the Environment

In Chapter 6 and Chapter 7 we analyzed some related works to point out that there is no only one alternative for the definition of an institution.

In general, when trying to define an institution, two main discussions are addressed. The first one refers to what the semantics of the logic we are trying to represent is. This discussion is mostly presented when the institution is not for a logic in the classical sense but for a specification language, as CSMOF and QVTR. The other important discussion refers to which elements must be in the signature and which one in the formulas. Institutions neither restrict nor determine this, and the decision depends on what we want to express and on the purpose of the institution.

From a model-theoretic point of view the original CSMOF and QVTR institutions are enough to represent the whole problem. However, as we already discussed in Section 5.4, we need a syntactic representation for SW-models in order to prove (in a proof-theoretic environment) that multiplicity constraints can be derived from SW-models. An entailment system allows proving a formula if it can be derived from other valid formulas. In the CSMOF institution, we only have a syntactic representation of multiplicity constraints as formulas. In this case, any possible entailment system will only derive multiplicity constraints from other multiplicity constraints, which is pointless for our purposes. The problem is that the specification language used for formulas is not expressive enough as for example a general logic like FOL. In this sense, several languages, as QVT-Relations transformations and OCL constraints, are not completely expressed by their own, i.e. they depend on the existence of a context, given by SW-models, in which they must be defined.

However, we have different alternatives: (1) to represent SW-models in the formulas, as in the extended institutions defined in Chapter 8, (2) to represent SW-models in the signature in such a way that the institution model is reduced to mimic the structure of the SW-model in the signature to be used by the satisfaction relation to avoid some contradictions (as the one

in Chapter 8), or (3) to represent SW-models in the signature in such a way that an institution model is determined by the SW-model. We took the first representation which is the most natural decision.

It is possible to define only one institution for representing MDE elements. The CSMOF institution together with an extended definition of formulas adding an expressions language on metamodels, e.g. the one defined in Chapter 7, can be enough. Indeed, this representation is equivalent to the idea of a transformation contract, as in [BMC+11], in which both source and target metamodels are defined in a unified metamodel, and constraints are used for expressing invariants in metamodels as well as describing the pre and post-conditions of the transformation. The problem with a representation of this kind is that its translation into a logical domain could be useless, since there is no difference between transformation rules and any other constraint, or between each one of the metamodels, which affects the understanding of the problem in the logical domain and perhaps the feedback which can be returned.

As discussed in Section 6.6, typing requirements of the structural conformance relation are not addressed by the satisfaction relation of the (extended) CSMOF institution since interpretations (SW-models) are always well-typed with respect to the signature (metamodel). We also argued in Section 3.1 that typing (conformance in general) is nowadays well understood and automatically checked within modeling frameworks. Nevertheless, for a complete formal basis based on institutions, it is interesting to consider the ideas presented in Section 6.6 for the definition of an institution addressing the type-checking problem, which is also connected with the theory of graph transformations.

As explained in Section 10.4, to address the verification of properties at another abstraction level, i.e. not considering specific SW-models but only metamodels and the transformation specification, we need another institutional representation or another comorphism in which concrete SW-models are forgotten but an abstract representation for them is generated. In terms of CASL, we can represent a SW-model as a type generated from a set for each concrete class. This SW-model can be instantiated if necessary with specific elements. These aspects are shown in the following code.

```
type Class_Model ::= mkClass_Model (Set[Class];...)
%{ A specific model instance }%
setClass : Set[Class] = {} + Class_1 + Class_2 + Class_3 + Class_4;

model : Class_Model = mkClass_Model(setClass,...)
```

For processing this model we need to generate some minimum infrastructure. In the following example we have an operation accessing `Classifier` instances in the SW-model and a predicate allowing identifying if some `Classifier` instance exists within the SW-model.

```
Classifier_allInstances __ : Class_Model -> Set[Classifier];
Classifier_allInstances model =
  PrimitiveDataType_allInstances model
    union Class_allInstances model

inModel __ __ (x : Classifier; model : Class_Model) <=>
    x eps Classifier_allInstances model;
```

Constraints must also be changed since every element referred in the constraint must be defined within the SW-model, as in the following example.

```
constraint __ (model : Class_Model) <=>
forall x : Class . inModel x model /\
  exists y : Attribute . inModel y model /\ attribute x y;
```

Finally, the use of generalized theoroidal comorphisms [Cod08] was a must for the translation of QVTR but not for the translation of CSMOF. In the case of CSMOF we introduced total functions (constants) representing typed elements contained in a SW-model formula. In the case of using a theoroidal or plain comorphism we cannot generate these functions, thus existential and equality axioms of typed elements must be added such that, in conjunction with the other generated axioms, need to enclose any other formula which must be proved. In this sense, we end up having big formulas, which is not necessarily a problem but it is a less optimal and not so readable CASL representation. In the case of QVTR we define predicates (within the CASL signature) corresponding to transformation rules (within a QVTR formula) such that it is possible to call a rule just referencing to its corresponding predicate. In the case of using a theoroidal or plain comorphism we cannot generate these predicates. However, in this case we can have a problem where two rules are mutually dependent, since the direct translation would imply an infinite embedding of formulas.

11.2.2 An Institution for OCL

As mentioned in Chapter 2, the `when` and `where` clauses, as well as the predicate of a pattern, may contain arbitrary boolean OCL expressions. From a formal perspective we would rather have an institution for OCL which would allow us to use the language not only for constraining the transformation rules, but also for expressing general constraints on metamodels.

Since OCL constraints need to be expressed in the context of a concrete SW-model in order to be checked, we can use our extended CSMOF institution and add the OCL constraints as part of the formulas (the multiplicity constraints can be expressed in OCL). The most problematic aspect for defining such institution is the fact that OCL is a three-valued logic, i.e. beyond the notion of truth or falsity of a constraint there exists the notion of undefinedness. In this sense, the satisfaction relation needs to be handled in a different way. Fortunately, there are some works, as [Dia13] in which the author develops institutional basis to combine many-valued logics with other logical systems. Further studies should be conducted in this sense.

Instead of defining a new institution we can evaluate the use of some current representation of OCL in another institution, e.g. in rewriting logics as in [BM09]. We already consider the inclusion of another institution for representing OCL by means of the expressions language introduced in Section 7.1.2, but we need to study the relation between our abstract definition and these other semantics.

The future inclusion of OCL as an institution will provide additional support in automated proofs. For example, we will provide a comorphism from our institutions to the one for OCL to support other verification approaches, as the one in [CCGdL10] for generating a transformation contract in OCL.

11.2.3 Model Transformations & Comorphisms

It is important to understand the relation between model transformations and comorphisms. Comorphisms are semantic-preserving transformations but not every model transformation is semantic-preserving. The identification of a model transformation as a comorphism can be reasonable for some kinds of model transformations, like a refactoring (a change in the representation without changing its externally observable behavior) or the Class to Relational transformation in this thesis. The key point here is that both source and target languages are considered semantically equivalent, even though there are represented in different technological spaces. In our example the semantics of Class diagrams and Relational models are defined in terms of the set of SW-models which conform to a specific Class (resp. Relational) metamodel. The transformation preserves these instances, e.g. any valid class diagram is transformed into a valid relational model. However, if source and target metamodels contain fundamentally different assumptions, it might be difficult to completely preserve the semantics.

On the contrary, a comorphism can be represented as a model transformation. In fact, the representation of comorphisms in HETS can be seen as functional text-to-text transformations. We can actually represent the institutions as metamodels, and state the comorphisms as model transformations. This idea promotes the use of MOF and QVT-Relations as a metalanguage within HETS which may improve the adoption of HETS by MDE practitioners. We can specify a metamodel for any logic we want to add, e.g. our CSMOF logic is specified by the metamodel in Figure 10.3, and we can define any logical description conforming to such model, e.g. any CSMOF instance is represented as a XMI file conforming to such metamodel. From this point we have two alternatives:

1. We can automatically derive a Haskell representation of the metamodel, following the idea of the CSMOF representation in which hierarchical classes are represented as interrelated types, and state the comorphism in Haskell as usual. We also need to define a Haskell representation of the institution, as well as a parser and a static analyzer. This is basically the path we followed.

2. If the target logic is already specified by a metamodel, we can define a QVT-Relations transformation between both logics such that any logical description using the source logic will be transformed into a logical description of the target logic. This avoids the implementation of Haskell code, but the environment has formal basis that cannot be ignored, i.e. the transformation between both logics must be semantic-preserving. Indeed, we must prove on paper that there is an institution for the source logic and a comorphism from it to the target logic such that the comorphism is model expansive.

11.2.4 Bridging Technological Spaces

The environment can be extended to support other transformation approaches, and to provide specific tools for MDE practitioners.

We can remove the restrictions imposed for the definition of an institution for QVT-Relations, i.e. we considered only a source and a target metamodel, and the transformation was executed in the direction of the second domain. Now that we have a first version of the environment, we can study how to extend its support to QVT-Relations by considering multiple metamodels and several domains, as defined in the standard. In the same sense, we need a deeper study for supporting other transformation approaches as the graph-transformation-based and the operational approaches. This introduces an interesting question: Is it possible to add another language into the environment without defining a new institution?

If it is possible to define a model transformation from the desired language to QVT-Relations, we can add it to the environment following the same idea introduced in the last subsection in which metamodels and model transformations are devised as a meta-language within HETS. However, the use of an intermediate language (QVT-Relations in this case) before reaching a logical domain may generate a formal representation which is not the one we expected. An example of this is the following, related to the comorphism between Maude and CASL [CMRM10], and the proposal or representing the semantics of MOF with rewriting logics [BM09]. We can specify a model transformation between our CSMOF institution and the representation of SW-models and metamodels in [BM09]. In this case, there will be a generic Maude sort `Class` representing every possible class in the metamodel, which can be instantiated. Then, if we follow the comorphism in [CMRM10] we will generate the corresponding sort in CASL. However, if we directly use our comorphism we will have one CASL sort for each class in the metamodel, thus, the representations are different and could affect the kind of reasoning we want to address.

Another alternative for the representation of transformation approaches and languages is the definition of an institution for a generic transformation language, as in [LR12], or a transformation contract, as in [BMC+11]. In both cases the institution must express the intended effect of a transformation by a metamodel plus constraints (e.g. in OCL) describing the pre- and post-conditions of the transformation. We must conduct a deeper study in this sense. For now we have identified some potential problems as stated in Section 11.2.1.

Although we state a separation in different technological domains, we can bridge the gap between MDE and formal verification in terms of tool development. First, we can connect the definition of the MDE elements in any popular tool with an automatic generation of the heterogeneous specification, as explained in Section 10.3, and the execution of HETS using this specification. Moreover, we could perform an automated verification of some properties (if possible) by running HETS in the background and providing a better user interface to show the problems found by HETS. For this to be possible, we need to improve feedback from existing formal tools. As studied in [ZCP13], this needs better traceability between the problem definition and the results given by a verification tool. We can define some traceability links from comorphisms, interpret the output of the verification tool and return something that the MDE practitioner can interpret. This interpretation is like defining a transformation between the domain of outputs of the verification tool and the domain of messages in MDE.

Finally, as described in Chapter 10, the environment deals with many verification properties, but a deeper understanding of this is a must. In this sense, we can use the knowledge in Chapter 3 to provide a guide for the selection of the "right" verification approach for the problem which is of interest to verify.

11.2.5 Evolution of the Prototype

Although the prototype is fully functional, there are some aspects that can be improved, e.g. a better documentation for final users, the inclusion of OCL (as described before), and a better support for primitive types. This last aspect is related to the fact that the comorphisms need to generate CASL theories with a representation of primitive types and type constructors (those within the expressions language introduced in Chapter 7). Nowadays the `QVTR2CASL` comorphism generates types as any other class, and includes the generation of the type constructor $+$ (append) for strings. However, there is an extensive CASL library which would be good to integrate. Since theories within these libraries are needed to be added to other theories when a comorphism is executed, we must define a new mechanism in HETS for this purpose which is currently not supported.

CASL libraries are not tuned for automated proofs. An evidence of this is the constraint on strings in the where clause of the `AttributeToColumn` relation (see Section F.4). The constraint states that if `prefix` is not empty, the name of the column must be equal to the prefix plus the name of the attribute, i.e. `=(cn)(prefix+an)`. The problem here is that the whole expression generates an unbounded domain in which there are potentially infinite strings to consider. If we change this expression for `=(cn)(an)` the proof is found immediately. Another limitation for automated proofs is the representation of hierarchical structures between classes by the subsorting relation. In the example we have three levels, i.e. `Class` $\leq$ `Classifier` $\leq$ `UMLModelElement`, but when trying to verify the complete Class to Relational example in the standard and another level is included, i.e. `Class` $\leq$ `Classifier` $\leq$ `PackageElement` $\leq$ `UMLModelElement`, the complexity of proving properties in upper levels (e.g. that each `Classifiers` has only one name) increases exponentially.

Appendices

A Publication List

The research work undertaken during the development of this thesis has been presented and published in the following instances. These papers cover most of the work: the comprehensive literature review on the verification of model transformations [CS13d], with an extended version [CS12] and preliminary literature reviews on the specification [LVV+09a] and verification [LVV+09b] of models transformations published as technical reports; the definition of the heterogeneous environment [CS13a]; and the definition of the institutions [CS13b], with an extended version [CS13c].

1. [CS13d] Daniel Calegari, Nora Szasz: Verification of Model Transformations: A Survey of the State-of-the-Art. Proc. Conf. Latinoamericana de Informática, Medellín, Colombia, 2012. ENTCS 292: 5-25. Elsevier (2013) **Best Paper Award**

    - [CS12] Daniel Calegari, Nora Szasz: Verification of Model Transformations: A Survey of the State-of-the-Art (Extended Version). Tech. Rep. 12-05, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2012)
    - [LVV+09a] Horacio López, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, Carlos Luna. Estado del Arte de Lenguajes y Herramientas de Transformación de Modelos. Tech. Rep. 09-19, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2009)
    - [LVV+09b] Horacio López, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, Carlos Luna. Estado del Arte de Verificación de Transformación de Modelos. Tech. Rep. 10-07, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2010)

2. [CS13a] Daniel Calegari, Nora Szasz: Bridging Techological Spaces for the Verification of Model Transformations. Proc. Conf. Iberoamericana de Software Engineering, Montevideo, Uruguay (2013)

3. [CS13b] Daniel Calegari, Nora Szasz: Institution-Based Semantics for MOF and QVT-Relations. Proc. 16th Brazilian Symp. Formal Methods, Brasilia, Brasil. LNCS 8195: 34-50. Springer. (2013) **2nd Best Paper Award**

    - [CS13c] Daniel Calegari, Nora Szasz: Institution-Based Semantics for MOF and QVT-Relations (extended version). Tech. Rep. 13-06, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2013)

There are other papers not completely related to the content of this thesis but produced under the same context. In early stages of this thesis we studied how to apply a type-theoretic framework, composed out by the Calculus of Inductive Constructions (CIC) and its associated tool the Coq proof assistant [BC04], to the formal treatment of model transformations [CLST09, CLST10b, CLST10a]. The approach is based on a semi-automatic translation process from metamodels, models and transformations of the MDE technical space into types, propositions and functions of the CIC technical space.

- [CLST09] Daniel Calegari, Carlos Luna, Nora Szasz, Álvaro Tasistro: Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. Proc. II Workshop Chileno de Métodos Formales, Santiago de Chile, Chile (2009)

- [CLST10b] Daniel Calegari, Carlos Luna, Nora Szasz, Alvaro Tasistro: A Type-Theoretic Framework for Certified Model Transformations. Proc. 13th Brazilian Symposium Formal Methods, Natal, Brasil. LNCS 6527: 112-127. Springer. (2010)

- [CLST10a] Daniel Calegari, Carlos Luna, Nora Szasz, Alvaro Tasistro Representation of Metamodels using Inductive Types in a Type-Theoretic Framework for MDE. Tech. Rep. 10-01, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2010)

Moreover, as a way of exploring the theory of Institutions and the characteristics of a heterogeneous environment, as in [CKTW08], we defined an institution for UML 2.0 State Machines [CS11b], with an extended version [CS11a] published as a technical report (Series ISSN: 0797–6410). The building blocks of this institution are based on a previous semantics dealing with processing simple input events within a transition step. We also extend these semantic definitions for handling sequences of events, and then for considering runs through the state machine.

- [CS11b] Daniel Calegari, Nora Szasz: Institutionalising UML 2.0 State Machines. ISSE 7(4): 315-323 (2011)

- [CS11a] Daniel Calegari, Nora Szasz: An Institution for UML 2.0 State Machines. Tech. Rep. 11-02, Series ISSN: 0797–6410, Instituto de Computación, Universidad de la República, Uruguay (2011)

B

Proofs :: Institution for CSMOF

In this appendix we present proofs for the CSMOF institution defined in Chapter 6.

Lemma 6.2.7. *Signatures and signature morphisms define a category* ***Sign****. The points of the category are the signatures and the arrows are the signature morphisms.*

Proof. Let $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1..4)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$, and $\boldsymbol{P}_i = (R_i, P_i)$ be signatures, and let $\sigma_i : \Sigma_i \rightarrow \Sigma_{i+1}$ $(i{=}1..3)$ be signature morphisms, then:

- Signature morphisms can be composed. We define the composition $\sigma_2 \circ \sigma_1$ as the tuple $\langle \sigma_T, \sigma_R \rangle$ such that $\sigma_T(c) = \sigma_{T_2}(\sigma_{T_1}(c))$, and $\sigma_R(c) = \sigma_{R_2}(\sigma_{R_1}(c))$. We have to show that $\sigma_2 \circ \sigma_1$ is a signature morphism:
  - For all $a \in C_1$ we have that $\sigma_T(a) = \sigma_{T_2}(\sigma_{T_1}(c))$ by definition of σ_T, and that $\sigma_{T_1}(c) \in C_2$ by definition of σ_{T_1}. Moreover, $\sigma_{T_2}(\sigma_{T_1}(c)) \in C_3$ by definition of σ_{T_2}. In consequence, $a \in C_1$ implies $\sigma_T(a) \in C_3$.
  - For all $a, b \in C_1$ with $a \leq_{C_1} b$ we have that $\sigma_{T_1}(a) \leq_{C_2} \sigma_{T_1}(b)$ by definition of σ_{T_1}. Moreover, we have that $\sigma_{T_1}(a), \sigma_{T_1}(b) \in C_2$ and thus $\sigma_{T_2}(\sigma_{T_1}(a)) \leq_{C_3} \sigma_{T_2}(\sigma_{T_1}(b))$ by definition of σ_{T_2}. Finally, we conclude that $a, b \in C_1$ with $a \leq_{C_1} b$ implies $\sigma_T(a) \leq_{C_3} \sigma_T(b)$ by definition of σ_T.
  - For all $a \in \alpha_1$ we have that $\sigma_{T_1}(a) \in \alpha_2$ by definition of σ_{T_1} and also that $\sigma_{T_2}(\sigma_{T_1}(a)) \in \alpha_3$ by definition of σ_{T_2}. Finally, $a \in \alpha_1$ implies $\sigma_T(a) \in \alpha_3$ by definition of σ_T.
  - For all $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ we have that $\langle \sigma_{R_1}(r_1) : \sigma_{T_1}(c_1), \sigma_{R_1}(r_2) : \sigma_{T_1}(c_2) \rangle \in P_2$ by definition of σ_{T_1} and σ_{R_1}, and also that $\langle \sigma_{R_2}(\sigma_{R_1}(r_1)) : \sigma_{T_2}(\sigma_{T_1}(c_1)), \sigma_{R_2}(\sigma_{R_1}(r_2)) : \sigma_{T_2}(\sigma_{T_1}(c_2)) \rangle \in P_3$ by definition of σ_{T_2} and σ_{R_2}. Finally, $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ implies

$\langle \sigma_R(r_1) : \sigma_T(c_1), \sigma_R(r_2) : \sigma_T(c_2) \rangle \in P_3$ by definition of σ_T and σ_R.

- Composition of signature morphisms is associative, i.e. $(\sigma_3 \circ \sigma_2) \circ \sigma_1 = \sigma_3 \circ (\sigma_2 \circ \sigma_1)$:
  - For each $a \in C_1$ we have that $\sigma_{T_2} \circ \sigma_{T_1}(a) = \sigma_{T_2}(\sigma_{T_1}(a))$ and thus $\sigma_{T_3} \circ (\sigma_{T_2} \circ \sigma_{T_1})(a) = \sigma_{T_3}(\sigma_{T_2}(\sigma_{T_1}(a)))$ by the definition of composition. Finally, this last result is equals to $\sigma_{T_3} \circ \sigma_{T_2}(\sigma_{T_1}(a))$ which is equals to $(\sigma_{T_3} \circ \sigma_{T_2}) \circ \sigma_{T_1}(a)$.
  - The proof is the same in the case of σ_R.
- There exists an identity signature morphism $id_{\Sigma_1} : \Sigma_1 \to \Sigma_1$ defined as a tuple $\langle id_T, id_R \rangle$ such that $id_T(c) = c$, and $id_R(c) = c$. This morphism satisfies the signature morphism conditions:
  - $a \in C_1$ implies $id_T(a) \in C_1$,
  - $a, b \in C_1$ with $a \leq_{C_1} b$ implies $id_T(a) \leq_{C_1} id_T(b)$,
  - $a \in \alpha_1$ implies $id_T(a) \in \alpha_1$,
  - $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ implies $\langle id_R(r_1) : id_T(c_1), id_R(r_2) : id_T(c_2) \rangle \in P_1$,

Finally, signatures and signature morphisms define a category.

□

Lemma 6.2.8. *There is a functor Sen giving a set of formulas ψ (object in the category **Set**) for each signature Σ (object in the category **Sign**), and a function $\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$ (arrow in the category **Set**) translating formulas for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**).*

Proof. We have to prove that *Sen* is indeed a functor, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.

(a) By the definition of formulas, the image of a signature Σ_i in the category **Sign** is an object $Sen(\Sigma_i)$ in the category **Set**. Moreover, by the definition of the extension of the signature morphism to formulas, the translation of the object in $Sen(\Sigma_1)$ coincides with an object in $Sen(\Sigma_2)$ with the types and roles translated with respect to the signature morphism $\sigma : \Sigma_1 \to \Sigma_2$. Thus, the image of an arrow in **Sign** is an arrow $\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$ in the category **Set**. Finally, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow.

(b) We have to prove that $Sen(\sigma_2 \circ \sigma_1) = Sen(\sigma_2) \circ Sen(\sigma_1)$.
Let Σ_i $(i{=}1..4)$ be signatures, and let $\sigma_i{:}\Sigma_i \rightarrow \Sigma_{i+1}$ $(i{=}1,2)$ be signature morphisms. $Sen(\sigma_2) \circ Sen(\sigma_1)$ is the canonical application of the signature morphism σ_1 to the elements in ψ_1, composed with the canonical application of the signature morphism σ_2 (by definition of signature morphism extend to formulas). Since signature morphisms can be composed (as defined in Lemma 6.2.7), this is the same as the canonical application of the composition of the signature morphism to ψ_1, i.e. $Sen(\sigma_2 \circ \sigma_1)$.

(c) Let $id_{\Sigma_1} : \Sigma_1 \rightarrow \Sigma_1$ be an identity signature morphism (defined in Lemma 6.2.7). We can see that identities are preserved since, by definition, for any Σ_1-formula ψ_1, $id_{\Sigma_1}(\psi_1)$ is a Σ_1-formula such that $id(r \bullet p) = id_R(r) \bullet id_P(p) = r \bullet p$.

Finally, the functor *Sen* is defined.

□

Lemma 6.3.7. *For any signatures, the Σ-interpretations and Σ-homomorphisms define a category **Mod**(Σ). The points of the category are the Σ-interpretations, its arrows are the Σ-homomorphisms.*

Proof. Let $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$, and $\boldsymbol{P} = (R, P)$ be a signature, let $\mathcal{I}_i = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}_i), \boldsymbol{A}_i)$ $(i{=}1..4)$ be Σ-interpretations, and let $h_i : \mathcal{I}_i \rightarrow \mathcal{I}_{i+1}$ $(i{=}1..3)$ be Σ-homomorphisms, then:

- Σ-homomorphisms can be composed. We define the composition $h_2 \circ h_1$ as a family of maps $(h_c)_{c \in T(C)}$ with $h_c : V_{c_1} \rightarrow V_{c_3}$ such that: $h_c(v) = h_{c_2}(h_{c_1}(v))$, $v \in O_{c_1}$. We have to prove that $h_2 \circ h_1$ is a Σ-homomorphism, i.e.
  - $h_c(v) \in O_{c_3}$ forall $v \in O_{c_1}$. By definition of homomorphism, we have that for each $v \in O_{c_1}$, $h_{c_1}(v) \in O_{c_2}$ and that for each $w \in O_{c_2}$, $h_{c_2}(w) \in O_{c_3}$, thus $h_{c_2}(h_{c_1}(v)) = h_c(v) \in O_{c_3}$.
  - $h_c(v) \in V_{c_3} \backslash O_{c_3}$ forall $v \in V_{c_1} \backslash O_{c_1}$. Proceeding in the same way as before, we have that for each $v \in V_{c_1} \backslash O_{c_1}$, $h_{c_2}(h_{c_1}(v)) = h_c(v) \in V_{c_3} \backslash O_{c_3}$.
  - $(v_1, v_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_1}$ iff $(h_c(v_1), h_c(v_2)) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_3}$ for any $v_i \in V_{c_1}$ $(i = 1, 2)$ and $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$. By definition of homomorphism, we have that for any $v_i \in V_{c_1}$ $(i = 1, 2)$ and $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, $(v_1, v_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}}_1$ iff $(h_{c_1}(v_1), h_{c_2}(v_2)) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_2}$, and also for any $w_i \in V_{c_2}$ $(i = 1, 2)$ and $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, $(w_1, w_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_2}$ iff $(h_{c_2}(w_1), h_{c_2}(w_2)) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_2}$. Thus, for any $v_i \in V_{c_1}$ $(i = 1, 2)$ and $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$, $(v_1, v_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_1}$ iff $(h_{c_2}(h_{c_1}(v_1)), h_{c_2}(h_{c_1}(v_2))) = (h_c(v_1), h_c(v_2)) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_3}$.
- Composition of Σ-homomorphisms is associative. By definition of composition of homomorphisms, for each $v \in O_{c_1}$ we have that $(h_3 \circ h_2) \circ h_1(v) = (h_3 \circ h_2)(h_{c_1}(v)) = h_{c_3}(h_{c_2}(h_{c_1}(v))) = h_3 \circ h_{c_2}(h_{c_1}(v)) = h_3 \circ (h_2 \circ h_1)$.

- There exist an identity Σ-homomorphism $id_{\mathcal{I}_1} : \mathcal{I}_1 \to \mathcal{I}_1$ consisting of a family of maps $(id_c)_{c \in T(C)}$ with $id_c : V_{c_1} \to V_{c_1}$ such that: $id_c(v) = v$, $v \in O_{c_1}$. It trivially holds that $id_{\mathcal{I}_1}$ is a Σ-homomorphism since:
  - $id_c(v) = v \in O_{c_1}$ forall $v \in O_{c_1}$.
  - $id_c(v) = v \in V_{c_1} \backslash O_{c_1}$ forall $v \in V_{c_1} \backslash O_{c_1}$.
  - $(v_1, v_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_1}$ iff $(id_c(v_1), id_c(v_2)) = (v_1, v_2) \in \langle r_1 : c_1, r_2 : c_2 \rangle^{\mathcal{I}_1}$ for any $v_i \in V_{c_1}$ $(i = 1, 2)$ and $\langle r_1 : c_1, r_2 : c_2 \rangle \in P$.

Finally, Σ-interpretations and Σ-homomorphisms define a category.

□

Lemma 6.4.4. *The reduct of Σ-interpretations and Σ-homomorphisms is a functor **Mod**(σ) from Σ_2-interpretations (Σ_2-homomorphisms) to Σ_1-interpretations (Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$.*

Proof. By definition, domain and codomain of the reduct of an homomorphism are the reduct of domain and codomain, respectively, of the homomorphism. We have now to prove that: (a) the reduct of a composition of two homomorphisms is the composition of the reducts of those homomorphisms, and (b) that the reduct of an identity homomorphism is likewise an identity.

Let $\Sigma = (\boldsymbol{C}, \alpha, \boldsymbol{P})$ with $\boldsymbol{C} = (C, \leq_C)$, and $\boldsymbol{P} = (R, P)$ be a signature, let $\mathcal{I}_i = (\boldsymbol{V}_{\boldsymbol{C}}^{\boldsymbol{T}}(\boldsymbol{O}_i), \boldsymbol{A}_i)$ $(i{=}1..3)$ be Σ-interpretations, and let $h_i : \mathcal{I}_i \to \mathcal{I}_{i+1}$ $(i{=}1, 2)$ be Σ-homomorphisms.

(a) $(h_2 \circ h_1)|_\sigma = h_2|_\sigma \circ h_1|_\sigma$. By definition of reduct of homomorphisms, we have that for any $c \in T(C)$, for any $v \in V_c$, $(h_2 \circ h_1)|_\sigma$ is defined by $(h_2 \circ h_1)_{\sigma(c)}(v)$. By definition of composition of homomorphisms, this is equals to $(h_2)_{\sigma(c)}((h_1)_{\sigma(c)}(v))$. Thus, by definition of composition of homomorphisms, this is equals to $((h_2)_{\sigma(c)} \circ (h_1)_{\sigma(c)})(v)$ which is the definition of $h_2|_\sigma \circ h_1|_\sigma$.

(b) Let $id_{\mathcal{I}_2}$ be an identity Σ_2-homomorphism, then $id_{\mathcal{I}_2}|_\sigma$ is an identity Σ_1-homomorphism, since by definition of reduct of a homomorphism $id_{\mathcal{I}_2}|_\sigma$ is the Σ_1-homomorphism h_1 defined by $h_{1_c}(v) = id_{\mathcal{I}_{2_{\sigma(c)}}}(v) = v$ for any $c \in T(C)$, for any $v \in V_c$.

Finally, the reduct of Σ-interpretations and Σ-homomorphisms is a functor. □

Lemma 6.4.5. *There is a functor **Mod** giving a category **Mod**(Σ) of Σ-interpretations (object in the category **Cat**) for each signature Σ (object in the category **Sign**), as shown in Lemma 6.3.7, and a functor **Mod**(σ) (arrow in the category **Cat**) from Σ_2-interpretations to Σ_1-interpretations (and Σ_2-homomorphisms to Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**), as shown in Lemma 6.4.4.*

Proof. We have to prove that **Mod** is indeed a functor, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.

(a) By Lemma 6.4.4, the image of an arrow $\sigma : \Sigma_2 \to \Sigma_1$ in the category $\mathbf{Sign}^{op}$ is an arrow $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ in the category **Cat**. Also, by Lemma 6.3.7, the image of a signature Σ in the category **Sign** is an object $\mathbf{Mod}(\Sigma)$ in the category **Cat**. Thus, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow.

(b) We have to prove that $\mathbf{Mod}(\sigma_2 \circ \sigma_1) = \mathbf{Mod}(\sigma_2) \circ \mathbf{Mod}(\sigma_1)$ for both, interpretations and homomorphisms. Let Σ_i $(i{=}1..3)$ be signatures, let $\sigma_i : \Sigma_i \to \Sigma_{i+1}$ $(i{=}1, 2)$ be signature morphisms, let $\mathcal{I} = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$ be a Σ_3-interpretation, and let h be a Σ_3-homomorphism. Then, we have to prove:

- $\mathcal{I}|_{\sigma_2 \circ\ \sigma_1} = (\mathcal{I}|_{\sigma_2})|_{\sigma_1}$.
  By definition of reduct, $\mathcal{I}|_{\sigma_2}$ is the Σ_2-interpretation $(\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}|_{\sigma_2}), \boldsymbol{A}|_{\sigma_2})$ such that:
  - $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}|_{\sigma_2}) = (V_{\sigma_2(c)})_{c \in T(C_2)}$
  - $\boldsymbol{A}|_{\sigma_2} = \{\langle \sigma_{2R}(r_1) : \sigma_{2T}(c_1), \sigma_{2R}(r_2) : \sigma_{2T}(c_2)\rangle^{\mathcal{I}} \mid \langle r_1 : c_1, r_2 : c_2\rangle \in P_2\}$

  Then $(\mathcal{I}|_{\sigma_2})|_{\sigma_1}$ is the Σ_1-interpretation $(\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}((\boldsymbol{O}|_{\sigma_2})|_{\sigma_1}), (\boldsymbol{A}|_{\sigma_2})|_{\sigma_1})$ such that:
  - $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}((\boldsymbol{O}|_{\sigma_2})|_{\sigma_1}) = (V_{\sigma_2(\sigma_1(c))})_{c \in T(C_1)}$
  - $(\boldsymbol{A}|_{\sigma_2})|_{\sigma_1} = \{\langle \sigma_{2R}(\sigma_{1R}(r_1)) : \sigma_{2T}(\sigma_{1T}(c_1)), \sigma_{2R}(\sigma_{1R}(r_2)) : \sigma_{2T}(\sigma_{1T}(c_2))\rangle^{\mathcal{I}} \mid \langle r_1 : c_1, r_2 : c_2\rangle \in P_1\}$

  and this is equal to $\mathcal{I}|_{\sigma_2 \circ\ \sigma_1}$.
- $h|_{\sigma_2 \circ\ \sigma_1} = (h|_{\sigma_2})|_{\sigma_1}$.
  By definition of reduct, $h|_{\sigma_2}$ is defined by $h|_{\sigma_2 c}\ (v) = h_{\sigma_2(c)}(v)$ for any $c \in T(C_2)$, for any $v \in V_c$, and thus $(h|_{\sigma_2})|_{\sigma_1}$ is defined by $(h|_{\sigma_2})|_{\sigma_1 c}\ (v) = h_{\sigma_2(\sigma_1(c))}(v) = h_{\sigma_2 \circ \sigma_1(c)}(v)$ for any $c \in T(C_1)$, for any $v \in V_c$, which is equals to $h|_{\sigma_2 \circ\ \sigma_1}$.

(c) Let $id_\sigma : \Sigma \to \Sigma$ be an identity signature morphism (defined in Lemma 6.2.7). We have to prove that $\mathbf{Mod}(id_\sigma)$ is an identity functor, i.e., it is composed by the identity reduct of Σ-interpretations and the identity reduct of Σ-homomorphisms.

- By definition of reduct, for any Σ-interpretation $\mathcal{I} = (\boldsymbol{V_C^T(O)}, \boldsymbol{A})$, $\mathcal{I}\,|_{id_\sigma}$ is the Σ-interpretation
  $(\boldsymbol{V_C^T(O}|_{id_\sigma}), \boldsymbol{A}|_{id_\sigma})$ such that:
  - $\boldsymbol{V_C^T(O}|_{id_\sigma}) = (V_{id_\sigma(c)})_{c \in T(C)}$
  - $\boldsymbol{A}|_{id_\sigma} = \{\langle id_{\sigma R}(r_1) : id_{\sigma T}(c_1), id_{\sigma R}(r_2) : id_{\sigma T}(c_2)\rangle^{\mathcal{I}} \mid \langle r_1 : c_1, r_2 : c_2\rangle \in P\}$

  Finally, by the definition of id_σ, $\mathcal{I}\,|_{id_\sigma} = \mathcal{I}$, thus $\_\,|_{id_\sigma}$ is the identity reduct of Σ-interpretations.
- By definition of reduct, given a Σ-interpretation $\mathcal{I}_1 = (\boldsymbol{V_C^T(O}_1), \boldsymbol{A}_1)$, for any Σ-homomorphism $h : \mathcal{I}_1 \to \mathcal{I}_2$, the reduct $h|_{id_\sigma}$ is defined by $h|_{id_\sigma\, c}\,(v) = h_{id_\sigma(c)}(v) = h_c(v)$ for any $c \in T(C)$, for any $v \in V_c$. Now, since $\mathcal{I}|_{id_\sigma} = \mathcal{I}$, we have that $\_|_{id_\sigma}$ is the identity reduct of Σ-homomorphisms.

Finally, the functor **Mod** is defined.

□

Theorem 6.4.6 (CSMOF satisfaction condition). *Given signatures $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1, 2)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$, and $\boldsymbol{P}_i = (R_i, P_i)$ a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I} = (\boldsymbol{V_C^T(O)}, \boldsymbol{A})$, and a Σ_1-formula ψ, the following satisfaction condition holds.*

$$\mathcal{I}|_\sigma \models_{\Sigma_1} \psi \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\psi)$$

Proof. We know that $\langle r_1 : c_1, r_2 : c_2\rangle^{\mathcal{I}|_\sigma} = \langle \sigma_R(r_1) : \sigma_T(c_1), \sigma_R(r_2) : \sigma_T(c_2)\rangle^{\mathcal{I}}$ $\forall\, \langle r_1 : c_1, r_2 : c_2\rangle \in P_1$ by definition of $\_|_\sigma$. With this result, we can deduce that $(c_i \bullet r_j)^{\mathcal{I}|_\sigma} = (\sigma_T(c_i) \bullet \sigma_R(r_j))^{\mathcal{I}}$. Moreover, for all φ of the form $\#(c \bullet r) = n$, we have that $\sigma(\varphi)$ is $\#(\sigma_T(c) \bullet \sigma_R(r)) = n$, by definition of σ. Finally, using both results we have that φ is $\#(c \bullet r) = n$ and $|S| = n$ $\forall\, S \in (c \bullet r)^{\mathcal{I}|_\sigma} \Leftrightarrow \sigma(\varphi)$ is $\#(\sigma_T(c) \bullet \sigma_R(r)) = n$ and $|S| = n$ $\forall\, S \in (\sigma_C(c) \bullet \sigma_R(r))^{\mathcal{I}}$. Thus, $\mathcal{I}|_\sigma \models_{\Sigma_1} \varphi \Leftrightarrow \mathcal{I} \models_{\Sigma_2} \sigma(\varphi)$.
We can proceed exactly in the same way for proving the other two cases of φ.

Finally, the satisfaction condition holds.

□

C

Proofs :: Institution for QVTR

In this appendix we present proofs for the QVTR institution defined in Chapter 7.

Lemma 7.2.5. *Signatures and signature morphisms define a category **Sign**. The points of the category are the signatures and the arrows are the signature morphisms.*

Proof. A signature morphisms is defined as a tuple of morphisms of the corresponding institutions $\langle \sigma_1^{\mathrm{C}}, \sigma_2^{\mathrm{C}} \rangle$. In those institutions, morphisms are composable, the composition is associative, and there exists an identity signature morphism. We define the composition $\sigma_2 \circ \sigma_1$ as the componentwise composition of morphisms, as well as the identity signature morphism as the tuple with the identity signature morphisms of the corresponding institutions. Using these facts, we conclude that: signature morphisms can be composed, composition of signature morphisms is associative, and there exists an identity signature morphism. Finally, signatures and signature morphisms define a category.

□

Lemma 7.2.6. *There is a functor Sen giving a set of formulas ψ (object in the category **Set**) for each signature Σ (object in the category **Sign**), and a function $\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$ (arrow in the category **Set**) translating formulas for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**).*

Proof. We have to prove that *Sen* is indeed a functor, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.

(a) By the extension of the signature morphism to formulas, the image of an arrow $\sigma : \Sigma_1 \rightarrow \Sigma_2$ in the category **Sign** is an arrow $\sigma : \mathit{Sen}(\Sigma_1) \rightarrow \mathit{Sen}(\Sigma_2)$ in the category **Set**. Also, by the definition of formulas, the image of a signature Σ in the category **Sign** is an object $\mathit{Sen}(\Sigma)$ in the category **Set**. Thus, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow.

(b) We have to prove that $\mathit{Sen}(\sigma_2 \circ \sigma_1) = \mathit{Sen}(\sigma_2) \circ \mathit{Sen}(\sigma_1)$.
Let Σ_i $(i{=}1..4)$ be signatures, and let $\sigma_i{:}\Sigma_i \rightarrow \Sigma_{i+1}$ $(i{=}1, 2)$ be signature morphisms. $\mathit{Sen}(\sigma_2) \circ \mathit{Sen}(\sigma_1)$ is the canonical application of the signature morphism σ_1 to the elements in ψ_1, composed with the canonical application of the signature morphism σ_2 (by definition of signature morphism extend to formulas). Since signature morphisms can be composed (as defined in Lemma 7.2.5), this is the same as the canonical application of the composition of the signature morphism to ψ_1, i.e. $\mathit{Sen}(\sigma_2 \circ \sigma_1)$.

(c) Let $id_{\Sigma_1} : \Sigma_1 \rightarrow \Sigma_1$ be an identity signature morphism (defined in Lemma 7.2.5). Identities are preserved since is the componentwise application of the identity signature morphisms to every element in the formula, which already preserve the identities.

Finally, the functor *Sen* is defined.

□

Lemma 7.3.6. *For any signatures, the Σ-interpretations and Σ-homomorphisms define a category **Mod**(Σ). The points of the category are the Σ-interpretations, and its arrows are the Σ-homomorphisms.*

Proof. A Σ-interpretation is a tuple of interpretations of the corresponding institutions, as well as homomorphisms are defined componentwise. In the corresponding institutions, homomorphisms are composable, the composition is associative, and there exists an identity homomorphism. We define the composition of homomorphisms as the componentwise composition of homomorphisms, as well as the identity homomorphism as the tuple with the identity homomorphisms of the corresponding institutions. Using these facts, it is straightforward to conclude that: homomorphisms can be composed, composition of homomorphisms is associative, and there exists an identity homomorphism. Finally, Σ-interpretations and Σ-homomorphisms define a category.

□

Lemma 7.3.7. *The reduct of Σ-interpretations and Σ-homomorphisms is a functor **Mod**(σ) from Σ_2-interpretations (Σ_2-homomorphisms) to Σ_1-interpretations (Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$.*

Proof. An interpretation is a tuple of interpretations of the corresponding institutions, as well as reducts are defined componentwise. In the corresponding institutions, the reduct of interpretations and homomorphisms is a functor. From this we can conclude straightforward that domain and codomain of the reduct of a homomorphism are the reduct of domain and codomain, respectively, the reduct of a composition of two homomorphisms is the composition of the reducts of those homomorphisms, and the reduct of an identity homomorphism is likewise an identity. Finally, the reduct of interpretations and homomorphisms is a functor.

□

Lemma 7.3.8. *There is a functor **Mod** giving a category **Mod**(Σ) of Σ-interpretations (object in the category **Cat**) for each signature Σ (object in the category **Sign**), as shown in Lemma 7.3.6, and a functor **Mod**(σ) (arrow in the category **Cat**) from Σ_2-interpretations to Σ_1-interpretations (and Σ_2-homomorphisms to Σ_1-homomorphisms) for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (arrow in the category **Sign**), as shown in Lemma 7.3.7.*

Proof. We have to prove that **Mod** is indeed a functor, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.

(a) By Lemma 7.3.7, the image of an arrow $\sigma : \Sigma_2 \to \Sigma_1$ in the category $\mathbf{Sign}^{op}$ is an arrow $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ in the category **Cat**. Also, by Lemma 7.3.6, the image of a signature Σ in the category **Sign** is an object $\mathbf{Mod}(\Sigma)$ in the category **Cat**. Thus, domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow.

(b) We have to prove that $\mathbf{Mod}(\sigma_2 \circ \sigma_1) = \mathbf{Mod}(\sigma_2) \circ \mathbf{Mod}(\sigma_1)$ for both, interpretations and homomorphisms. A reduct (homomorphism) is defined as a tuple of reducts (homomorphisms) of the corresponding institutions which are applied to each component in the interpretation. Since the property holds in isolation for each component, we can directly conclude that this also holds for the tuple.

(c) Let $id_\sigma : \Sigma \to \Sigma$ be an identity signature morphism (defined in Lemma 6.2.7). We have to prove that $\mathbf{Mod}(id_\sigma)$ is an identity functor, i.e., it is composed by the identity reduct of Σ-interpretations and the identity reduct of Σ-homomorphisms. Since reducts and homomorphisms are defined componentwise and the property holds in isolation for each component, we can directly conclude that this also holds for the tuple.

Finally, the functor **Mod** is defined.

□

Theorem 7.4.3 (QVTR satisfaction condition). *Given signatures* $\Sigma_i = \langle \Sigma_1^{\mathrm{C}_i}, \Sigma_2^{\mathrm{C}_i} \rangle (i = 1, 2)$*, a signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$*, a* Σ_2*-interpretation* $\mathcal{M} = \langle \mathcal{M}_1^{\mathrm{M}_2}, \mathcal{M}_2^{\mathrm{M}_2} \rangle$*, a set of variables* $X_2 = (X_2^s)_{s \in S_2}$*, and a* Σ_1*-formula* φ *with variables in* $X_2|_\sigma$*, the following satisfaction condition holds.*

$$\mathcal{M}|_\sigma \models_{\Sigma_1} \varphi \text{ iff } \mathcal{M} \models_{\Sigma_2} \sigma(\varphi)$$

Proof. We will first prove some preliminary results:

1. Given $\mathcal{I}^{\mathrm{E}}$ signatures Σ_i $(i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{M}$, a set of variables $X_2 = (X_2^s)_{s \in S_2}$, a $\mathcal{M}$-variable assignments μ, and a Σ_1-formula φ with variables in $X_2|_\sigma$, the following condition holds by definition of the $\mathcal{I}^{\mathrm{E}}$ institution: $\mathcal{M}|_\sigma, \mu|_\sigma \models_{\Sigma_1} \varphi$ iff $\mathcal{M}, \mu \models_{\Sigma_2} \sigma(\varphi)$

2. Given signatures $\Sigma_i (i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a derived Σ_2-model $\mathcal{M}^{\mathrm{E}}$ with a variable assignment μ, and a pattern $\mathrm{Pattern} = \langle E, A, Pr \rangle$, the following condition holds: $\mathcal{M}^{\mathrm{E}}, \mu \models \sigma(\mathrm{Pattern})$ iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models \mathrm{Pattern}$

   Proof.

   $\mathcal{M}^{\mathrm{E}}, \mu \models \sigma(\mathrm{Pattern})$
   iff $(\mu(\sigma(x)), \mu(\sigma(y))) \in \mathcal{M}^{\mathrm{E}}. \forall\, rel(\sigma(p), \sigma(x), \sigma(y)) \in \sigma(A)$
   and $\mathcal{M}^{\mathrm{E}}, \mu \models_{\mathrm{E}} \sigma(Pr)$
   (by def. of pattern satisf.)
   iff $(\mu|_\sigma\ (x), \mu|_\sigma\ (y)) \in \mathcal{M}^{\mathrm{E}}|_\sigma\ . \forall\, rel(p, x, y) \in A$
   (by def. of $\mathcal{M}^{\mathrm{E}}|_\sigma$ and σ)
   and $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models_{\mathrm{E}} Pr$
   (by result 1)
   iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models \mathrm{Pattern}$
   (by def. of pattern satisf.)

   □

3. Given signatures $\Sigma_i (i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a derived Σ_2-model $\mathcal{M}^{\mathrm{E}}$ with a variable assignments μ, and a clause $\mathrm{when} = \langle \mathrm{when}_c, \mathrm{when}_r \rangle$, the following condition holds: $\mathcal{M}^{\mathrm{E}}, \mu \models \sigma(\mathrm{when})$ iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models \mathrm{when}$

   Proof.

   $\mathcal{M}^{\mathrm{E}}, \mu \models \sigma(\mathrm{when})$
   iff $\mathcal{M}^{\mathrm{E}}, \mu \models_{\mathrm{E}} \sigma(\mathrm{when}_c)$
   and $\mathcal{M}^{\mathrm{E}}, \mu[\sigma(v)] \models \sigma(r). \forall(\sigma(r), \sigma(v)) \in \sigma(\mathrm{when}_r)$
   (by def. of satisfaction of a when clause)

   We also know that $\mathcal{M}^{\mathrm{E}}, \mu \models_{\mathrm{E}} \sigma(\mathrm{when}_c)$ iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models_{\mathrm{E}} \mathrm{when}_c$ by result 1. Thus, we have to prove that:

$$\mathcal{M}^{\mathrm{E}}, \mu[\sigma(v)] \models \sigma(r).\ \forall(\sigma(r), \sigma(v)) \in \sigma(\mathrm{when}_r) \text{ iff}$$
$$\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma\ [v] \models r.\ \forall(r, v) \in \mathrm{when}_r$$

This can be proved by induction on the length of the chain of dependencies of `when` and `where` clauses which is assumed to be finite as we discussed in Section 7.1. This means that the base case is $\mathrm{when}_r = \emptyset$ in which the condition trivially holds, since also $\sigma(\mathrm{when}_r) = \emptyset$. The inductive hypothesis is such that $\forall(\sigma(r), \sigma(v)) \in \sigma(\mathrm{when}_r)$ we have that $\mathcal{M}^{\mathrm{E}}, \mu[\sigma(v)] \models \sigma(r).$ iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma\ [v] \models r$ (and the same $\forall(r, v) \in \mathrm{when}_r$. Thus, the inductive thesis trivially holds from these hypothesis.

Finally, by definition of satisfaction of a `when` clause, we conclude that $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models \mathrm{when}$ also holds. This proof can be read backwards.

□

4. Given signatures $\Sigma_i (i = 1, 2)$, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a derived Σ_2-interpretation $\mathcal{M}^{\mathrm{E}}$ with a variable assignments μ, and a clause where, the following condition holds: $\mathcal{M}^{\mathrm{E}}, \mu \models \sigma(\mathrm{where})$ iff $\mathcal{M}^{\mathrm{E}}|_\sigma, \mu|_\sigma \models \mathrm{where}$

   Proof. The proof is similar to the case of a `when` clause.

□

Now, we can prove the satisfaction condition for the two kinds of formulas.

In the case of a formula $\varphi^{\mathrm{K}} = \langle c, \{r_1, ..., r_n\}\rangle\ (1 \leq n)$, we have that:

$\mathcal{M}|_\sigma \models_{\Sigma_1} \varphi^{\mathrm{K}}$
- iff $\forall x, y \in (V_{\sigma(c)})_{c \in C_i}, x \neq y$ implies
  $\bigcup_j \{\pi_2(t) |\ \pi_1(t) = x,\ t \in {p_j}^{\mathcal{M}|_\sigma}\} \neq \bigcup_j \{\pi_2(t) |\ \pi_1(t) = y,\ t \in {p_j}^{\mathcal{M}|_\sigma}\}$
  (by def. of satisfaction relation)
- iff $\forall x, y \in (V_c)_{c \in C_i}, x \neq y$ implies
  $\bigcup_j \{\pi_2(t) |\ \pi_1(t) = x,\ t \in \sigma(p_j)^{\mathcal{M}}\} \neq \bigcup_j \{\pi_2(t) |\ \pi_1(t) = y,\ t \in \sigma(p_j)^{\mathcal{M}}\}$
  (by def. of $\mathcal{M}|_\sigma$ and σ,
  since the morphism and the interpretation preserves the inequalities)
- iff $\mathcal{M} \models_{\Sigma_2} \sigma(\varphi^{\mathrm{K}})$
  (by def. of satisfaction relation)

In the case of a formula φ^{R}, for every top rule $\mathrm{Rule} \in \varphi^{\mathrm{R}}$ we need to prove that

$$\mathcal{M}^{\mathrm{E}}|_{\sigma}, \emptyset \models \mathrm{Rule} \text{ iff } \mathcal{M}^{\mathrm{E}}, \emptyset \models \sigma(\mathrm{Rule})$$

For any rule $\mathrm{Rule} = \langle \mathrm{Rule}, \mathrm{VarSet}, \mathrm{ParSet}, \mathrm{Pattern}_i \ (i = 1, 2), \mathrm{when}, \mathrm{where} \rangle$, we have two cases:

1. If $\mathrm{WhenVarSet} = \emptyset$, $\mathcal{M}^{\mathrm{E}_2}|_{\sigma}, \emptyset \models \mathrm{Rule}$ if

$$\begin{array}{l} \forall\, \mu^1|_{\sigma}\ [x_1, ..., x_n] \in\ |\mathrm{VarSet} \backslash \mathrm{2\_VarSet}|, \\ \quad (\ \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma}\ [x_1, ..., x_n] \models \mathrm{Pattern}_1 \rightarrow \\ \quad\quad \exists\, \mu^2|_{\sigma}\ [y_1, ..., y_m] \in |\mathrm{2\_VarSet}|, \\ \quad\quad\quad (\ \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma} \cup \mu^2|_{\sigma} \models \mathrm{Pattern}_2 \wedge \\ \quad\quad\quad\quad \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma} \cup \mu^2|_{\sigma} \models \mathrm{where})) \end{array}$$

2. If $\mathrm{WhenVarSet} \neq \emptyset$, $\mathcal{M}^{\mathrm{E}_2}|_{\sigma}, \emptyset \models \mathrm{Rule}$ if

$$\begin{array}{l} \forall\, \mu^w|_{\sigma}\ [z_1, ..., z_o] \in\ |\mathrm{WhenVarSet}|, \\ \quad (\ \mathcal{M}^{\mathrm{E}_2}|_{\sigma}, \mu^w|_{\sigma}\ [z_1, ..., z_o] \models \mathrm{when} \rightarrow \\ \quad\quad \forall\, \mu^1|_{\sigma}\ [x_1, ..., x_n] \in\ |\mathrm{VarSet} \backslash (\mathrm{WhenVarSet} \cup \mathrm{2\_VarSet})|, \\ \quad\quad\quad (\ \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma} \cup \mu^w|_{\sigma} \models \mathrm{Pattern}_1 \rightarrow \\ \quad\quad\quad\quad \exists\, \mu^2|_{\sigma}\ [y_1, ..., y_m] \in |\mathrm{2\_VarSet}|, \\ \quad\quad\quad\quad\quad (\ \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma} \cup \mu^2|_{\sigma} \cup \mu^w|_{\sigma} \models \mathrm{Pattern}_2 \wedge \\ \quad\quad\quad\quad\quad\quad \mathcal{M}^{\mathrm{E}_2}|_{\sigma},\ \mu^1|_{\sigma} \cup \mu^2|_{\sigma} \cup \mu^w|_{\sigma} \models \mathrm{where}))) \end{array}$$

In both cases we can directly use the preliminar results, plus the definition of $\mu|_{\sigma}$, to conclude that the following cases also hold:

1. If $\sigma(\mathrm{WhenVarSet}) = \emptyset$

$$\begin{array}{l} \forall\, \mu^1 [x_1, ..., x_n] \in\ |\mathrm{VarSet} \backslash \mathrm{2\_VarSet}|, \\ \quad (\ \mathcal{M}^{\mathrm{E}_2},\ \mu^1 [x_1, ..., x_n] \models \sigma(\mathrm{Pattern}_1) \rightarrow \\ \quad\quad \exists\, \mu^2 [y_1, ..., y_m] \in |\mathrm{2\_VarSet}|, \\ \quad\quad\quad (\ \mathcal{M}^{\mathrm{E}_2},\ \mu^1 \cup \mu^2 \models \sigma(\mathrm{Pattern}_2) \wedge \\ \quad\quad\quad\quad \mathcal{M}^{\mathrm{E}_2},\ \mu^1 \cup \mu^2 \models \sigma(\mathrm{where}))) \end{array}$$

2. If $\sigma(\text{WhenVarSet}) \neq \emptyset$

$$\begin{aligned}
&\forall\, \mu^w[z_1, ..., z_o] \in\ |\text{WhenVarSet}|,\\
&\quad (\ \mathcal{M}^{\text{E}_2}, \mu^w[z_1, ..., z_o] \models \sigma(\text{when}) \rightarrow\\
&\qquad \forall\, \mu^1[x_1, ..., x_n] \in\ |\text{VarSet}\backslash(\text{WhenVarSet} \cup \text{2_VarSet})|,\\
&\qquad\quad (\ \mathcal{M}^{\text{E}_2},\ \mu^1 \cup \mu^w \models \sigma(\text{Pattern}_1) \rightarrow\\
&\qquad\qquad \exists\, \mu^2[y_1, ..., y_m] \in |\text{2_VarSet}|,\\
&\qquad\qquad\quad (\ \mathcal{M}^{\text{E}_2},\ \mu^1 \cup \mu^2 \cup \mu^w \models \sigma(\text{Pattern}_2) \wedge\\
&\qquad\qquad\qquad \mathcal{M}^{\text{E}_2},\ \mu^1 \cup \mu^2 \cup \mu^w \models \sigma(\text{where}))))
\end{aligned}$$

Finally, we conclude that $\mathcal{M}^{\text{E}}, \emptyset \models \sigma(\text{Rule})$. Note that we can also read this proof downside up, thus the satisfaction condition holds.

□

D

Proofs :: CSMOF and QVTR Extensions

In this appendix we present proofs for the extensions of the CSMOF institution and the QVTR institution defined in Chapter 8.

Lemma 8.1.3. *There is a functor Sen giving a set of formulas ψ (object in the category **Set**) for each signature Σ (object in the category **Sign**), as shown in the definition of a formula, and a function $\sigma : Sen(\Sigma_1) \rightarrow Sen(\Sigma_2)$ (arrow in the category **Set**) translating formulas for each signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ (arrow in the category **Sign**), as shown in the extension of the signature morphism to formulas.*

Proof. We have to prove that *Sen* is indeed a functor, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.

(a) In the original definition we have that domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow. Since formulas are just extended with a new component (Ω), and the extension of the signature morphism to formulas is still the canonical application of the signature morphism, the proof also holds in the case of formulas Ω.

(b) We have proved that composition was preserved in the original definition of formulas and signature morphisms. The proof was based in the fact that the extension of signature morphisms to formulas is the canonical application of the signature morphism, and that signature morphisms can be composed, thus, the canonical application of the composition of the signature morphism is the same as the composition of the canonical application of both signature morphisms. In the new case, nothing changes, since the the extension of signature morphisms to these formulas is still the canonical application of the signature morphism. Thus, composition is still preserved.

(c) Let $id_{\Sigma_1} : \Sigma_1 \to \Sigma_1$ be an identity signature morphism (defined in Lemma 6.2.7). We had that identities are preserved in the case of a Σ_1-formula Φ since $id_{\Sigma_1}(\Phi)$ is a Σ_1-formula such that $id(r \bullet p) = id_R(r) \bullet id_P(p) = r \bullet p$. Now, identities are preserved also in the case of a Σ_1-formula Ω since $id_{\Sigma_1}(\Omega)$ is a Σ_1-formula such that $id_{\Sigma_1}(x^c) = x^{(c)} = x^c$, and $id_{\Sigma_1}(\langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2}\rangle) = \langle id(r_1), {x_1}^{id(c_1)}, id(r_2), {x_2}^{id(c_2)}\rangle = \langle r_1, {x_1}^{c_1}, r_2, {x_2}^{c_2}\rangle$.

Finally, the functor *Sen* is defined.

□

Theorem 8.1.10 ($\mathcal{I}^\Omega$ satisfaction condition). *Given signatures $\Sigma_i = (\boldsymbol{C}_i, \alpha_i, \boldsymbol{P}_i)$ $(i = 1, 2)$ with $\boldsymbol{C}_i = (C_i, \leq_{C_i})$, and $\boldsymbol{P}_i = (R_i, P_i)$ a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-interpretation $\mathcal{I} = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$, and a Σ_1-formula Ω, the following satisfaction condition holds.*

$$\mathcal{I}|_\sigma \models_{\Sigma_1} \Omega \text{ iff } \mathcal{I} \models_{\Sigma_2} \sigma(\Omega)$$

Proof. We know by definition that $\mathcal{I}|_\sigma \models_{\Sigma_1} \Omega$ if there exists a bijective function $K^{(\mathcal{I}|_\Omega)|_\sigma}(\Omega)$ mapping each $x^c \in \upsilon(\Omega)$ to an element of $(V_{\sigma(c)})_{c \in T(C)}$ for each $c \in T(C)$, such that syntactic links in Ω and semantic links in $(\mathcal{I}|_\Omega)|_\sigma$ coincide. Since $\sigma(\Omega)$ change types and roles not affecting variables or links, e.g. $\sigma(x^c) = x^{\sigma_T(c)}$, we can define that $K^{\mathcal{I}|_{\sigma(\Omega)}}(\sigma(\Omega))$ is the bijective function mapping each $x^{\sigma_T(c)} \in \upsilon(\sigma(\Omega))$ with $c \in T(C)$ to the same element of $(V_{\sigma(c)})_{c \in T(C)}$ as the other function. Since the interpretation $\mathcal{I}$ is restricted with respect to the explicit scope defined by $\sigma(\Omega)$, it will have the same elements than those in $(\mathcal{I}|_\Omega)|_\sigma$ and thus the sets of elements $(V_{\sigma(c)})_{c \in T(C)}$ considered in both functions are exactly the same. In the same way, since the interpretations coincide, we can define the correspondence between a link in Ω and its interpretation to be the same as for the translated link in $\sigma(\Omega)$.

Finally, we have that given a function $K^{(\mathcal{I}|_\Omega)|_\sigma}(\Omega)$ or $K^{\mathcal{I}|_{\sigma(\Omega)}}(\sigma(\Omega))$ we can directly define the other, and thus $\mathcal{I}|_\sigma \models_{\Sigma_1} \Omega$ iff $\mathcal{I} \models_{\Sigma_2} \sigma(\Omega)$, i.e. the satisfaction condition holds.

□

E

Proofs :: Comorphisms

In this appendix we present complete proofs for the properties stated in Chapter 9. In Section E.1 we present proofs for the generalized theoroidal comorphism from CSMOF to CASL, and in Section E.2 for the generalized theoriodal comorphism from QVTR to CASL.

E.1 Encoding CSMOF into CASL

Lemma 9.2.4. *Given $\mathcal{I}^{\mathrm{M}^+}$ signatures Σ_i, and a theory morphism $\sigma : \Sigma_1 \to \Sigma_2$, its translation $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.*

Proof. Given a set of Σ_1-formulas Ψ and a set of Σ_2-formulas Ψ_2 these sets have Φ formulas representing multiplicity constraints and Ω formulas representing SW-models. We know that Ψ and Ψ_2 comply with the theory morphism $\sigma : \langle \Sigma, \Psi \rangle \to \langle \Sigma_2, \Psi_2 \rangle$ if $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$. This means that if an interpretation $\mathcal{I}$ satisfies Ψ_2, it must satisfy $\sigma(\Psi)$. An interpretation satisfying Ψ_2 determines a concrete SW-model which satisfies the multiplicity constraints. In this sense, if there is a formula Ω in Ψ_2 representing a SW-model, any formula in $\sigma(\Psi)$ satisfied by the same interpretation $\mathcal{I}$ must be a derivable multiplicity constraint from the SW-model, or same structured SW-model with potentially less types (remember that there must be an isomorphism between the SW-model and the explicit scope defined by the reduced interpretation with respect to the types in the SW-model). Moreover, if there is a formula Φ in Ψ_2 representing a multiplicity constraint, any formula in $\sigma(\Psi)$ satisfied by the same interpretation $\mathcal{I}$ must be a derivable multiplicity constraint.

We also know that the translation Φ not only maps types and roles in a consistent way, but also adds axioms constraining the $\mathcal{I}^{\mathrm{C}}$ models that must be used for checking the satisfaction relation (e.g. the "distinguishability" and "completeness of elements" axioms). In this sense, any interpretation satisfying $\Phi(\Psi_2)$ and $\Phi(\sigma)(\Phi(\Psi))$ must have the same structure that the original $\mathcal{I}$. This means that if we translate the formula Ω in Ψ_2 and the corresponding deriv-

able multiplicity constraint or reduced SW-model in $\sigma(\Psi)$, we can find an equivalent $\mathcal{I}^{\mathrm{C}}$ model satisfying $\Phi(\Psi_2) \models_{\Phi(\Sigma_2)} \Phi(\sigma)(\Phi(\Psi))$. The same happens with the translation of Φ formulas representing multiplicity constraints.

In conclusion $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.

□

Lemma 9.2.5. *The function $\Phi : Th^{\mathcal{I}^{\mathrm{M}^+}} \rightarrow Th^{\mathcal{I}^{\mathrm{C}}}$ is a functor from the category of $\mathcal{I}^{\mathrm{M}^+}$ theories and theory morphisms to the category of theories in $SubPCFOL^{=}$, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.*

Proof. (a) By definition, the image of an arrow $\sigma : \Sigma_1 \rightarrow \Sigma_2$ in the category of $\mathcal{I}^{\mathrm{M}^+}$ theories is an arrow $\Phi(\sigma) : \Phi(\Sigma_1) \rightarrow \Phi(\Sigma_2)$ in the category of theories in $SubPCFOL^{=}$.

(b) We have to prove that $\Phi(\sigma_2 \circ \sigma_1) = \Phi(\sigma_2) \circ \Phi(\sigma_1)$. Let Σ_i $(i{=}1..4)$ be signatures, and let $\sigma_i{:}\Sigma_i \rightarrow \Sigma_{i+1}$ $(i{=}1,2)$ be signature morphisms. As defined in Lemma 6.2.7, the composition $\sigma_2 \circ \sigma_1$ is a tuple $\langle \sigma_T, \sigma_R \rangle$ such that $\sigma_T(c) = \sigma_{T_2}(\sigma_{T_1}(c))$, and $\sigma_R(c) = \sigma_{R_2}(\sigma_{R_1}(c))$. Its translation $\Phi(\sigma_2 \circ \sigma_1)$ is a $SubPCFOL^{=}$ signature morphism $\langle \sigma_S, \sigma_F, \sigma_P \rangle$ such that:

- $\sigma_S(\Phi(c)) = \Phi(\sigma_{T_2}(\sigma_{T_1}(c)))$ for every $c \in T(C_i)$
- $\sigma_P(r_1(c_2, c_1)) = \sigma_{R_2}(\sigma_{R_1}(r_1))(\sigma_{T_2}(\sigma_{T_1}(c_2)), \sigma_{T_2}(\sigma_{T_1}(c_1)))$, and
  $\sigma_P(r_2(c_1, c_2)) = \sigma_{R_2}(\sigma_{R_1}(r_2))(\sigma_{T_2}(\sigma_{T_1}(c_1)), \sigma_{T_2}(\sigma_{T_1}(c_2)))$,
  for every $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ and $r_1(c_2, c_1), r_2(c_1, c_2)$ the predicates generated from $\Phi(\langle r_1 : c_1, r_2 : c_2 \rangle)$.

Moreover, the translations $\Phi(\sigma_1)$ and $\Phi(\sigma_2)$ are $SubPCFOL^{=}$ signature morphisms $\langle \sigma_{S_i}, \sigma_{F_i}, \sigma_{P_i} \rangle$ $(i = 1, 2)$. Using this information and the properties of signature morphisms, we can conclude that $\Phi(\sigma_2 \circ \sigma_1) = \Phi(\sigma_2) \circ \Phi(\sigma_1)$, since

- $\sigma_{S_2} \circ \sigma_{S_1}(\Phi(c)) = \sigma_{S_2}(\Phi(\sigma_{T_1}(c))) = \Phi(\sigma_{T_2}(\sigma_{T_1}(c)))$
- $\sigma_{P_2} \circ \sigma_{P_1}(r_1(c_2, c_1)) = \sigma_{P_2}(\sigma_{R_1}(r_1)(\sigma_{T_1}(c_2), \sigma_{T_1}(c_1))) =$
  $\sigma_{R_2}(\sigma_{R_1}(r_1))(\sigma_{T_2}(\sigma_{T_1}(c_2)), \sigma_{T_2}(\sigma_{T_1}(c_1)))$. The other case is analogous.

(c) As defined in Lemma 6.2.7, the identity signature morphism id_σ in $\mathbf{Sign}^{\mathrm{M}}$ is a tuple of identity functions for types and roles. Its translation $\Phi(id_\sigma)$ is the identity signature morphism in $SubPCFOL^{=}$, since by definition of Φ we have that:

- $\sigma_S(\Phi(c)) = \Phi(id_\sigma(c)) = \Phi(c)$ for every $c \in T(C_i)$
- $\sigma_P(r_1(c_2, c_1)) = id_\sigma(r_1)(id_\sigma(c_2), id_\sigma(c_1)) = r_1(c_2, c_1)$, and
  $\sigma_P(r_2(c_1, c_2)) = id_\sigma(r_2)(id_\sigma(c_1), id_\sigma(c_2)) = r_2(c_1, c_2)$,
  for every $\langle r_1 : c_1, r_2 : c_2 \rangle \in P_1$ and $r_1(c_2, c_1), r_2(c_1, c_2)$ the predicates generated from $\Phi(\langle r_1 : c_1, r_2 : c_2 \rangle)$.

Finally, Φ is a functor. □

Lemma 9.2.6. *The function* $\beta : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}} \rightarrow \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}}$*, with* ***Mod*** $: Th \rightarrow$ ***Cat*** *the functor giving the category of models of a theory, is a natural transformation, i.e. a family of arrows* $\beta_A : (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}(A) \Rightarrow \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}}(A)$*, one for each theory* A *of* $\boldsymbol{Sign}^{\mathrm{M}}$*, such that, for every theory morphism* $\sigma : A \rightarrow B$ *it holds:* $\boldsymbol{Mod}^{\mathcal{I}^{\mathrm{M}^+}} \circ \beta_B = \beta_A \circ (\Phi)^{op}; \boldsymbol{Mod}^{\mathcal{I}^{\mathrm{C}}}$

Proof. Given any model in $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(B))$, the translation β_B gives an interpretation denoted $I = (\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O}), \boldsymbol{A})$ such that:

- Each non-empty carrier set $|M|_s$ with $s \in S$, is translated into the set V_c in the object domain $\boldsymbol{V}^{\boldsymbol{T}}_{\boldsymbol{C}}(\boldsymbol{O})$ such that s is the translation of type $c \in T(C)$.
- Each relation p_M of a predicate symbol $r_2(c_1, c_2) \in P$ derived from the translation of a predicate $\langle r_1 : c_1, r_2 : c_2 \rangle$, is translated into the relation $p^{\mathcal{I}} \subseteq V_{c_1} \times V_{c_2} \ \in \boldsymbol{A}$.

By definition of reduct, the application of $\mathbf{Mod}^{\mathcal{I}^{\mathrm{M}^+}}$ to this interpretation gives the same interpretation. In the other side, the reduct $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}(\Phi(\sigma))$ gives an interpretation of symbols of the translated signature A, and it composition with the translation β_A produces the same interpretation as before, since the carries sets $|M|_s$ with $s \in S$, and the relations p_M are those derived from the translation of elements in the translates signature, which was reduced to the elements in the signature A.

In the case of homomorphisms the property also holds. The translation of a homomorphism is defined in conformance with the original homomorphism, and the reduct gives the same homomorphism. For a homomorphism in $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}$ we have that $h'_{\sigma}(c) = h''_{\Phi(\sigma(c))}$ and the composition with the reduct gives a homomorphism $h_c = h'_{\sigma(c)}$. In the other side, the reduct of a homomorphism in $\mathbf{Mod}^{\mathcal{I}^{\mathrm{C}}}$ gives a homomorphism $h'_{\Phi(c)} = h''_{\Phi(\sigma(c))}$, and its translation gives a homomorphism $h_c = h'_{\Phi(c)}$ which is the same as before.

Finally, β is a natural transformation.

□

E.2 Encoding QVTR into CASL

Lemma 9.3.4. *Given $\mathcal{I}^{\mathrm{Q}^+}$ signatures Σ_i, and a theory morphism $\sigma : \Sigma_1 \to \Sigma_2$, its translation $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.*

Proof. We already proved in Lemma 9.2.4 that the translation of extended CSMOF formulas is a $\mathcal{I}^{\mathrm{C}}$ theory morphism. Since $\mathcal{I}^{\mathrm{Q}^+}$ signatures and signature morphisms are a pair of $\mathcal{I}^{\mathrm{M}^+}$ signatures and signature morphisms, we can conclude that $\Phi(\sigma)$ is a theory morphism when restricting the set of formulas to extended CSMOF formulas (each component of σ is a standalone theory morphism).

We can also extend this result by considering φ^{K} formulas representing key constraints. We have that given a set of Σ_1-formulas Ψ, and a set of Σ_2-formulas Ψ_2 they comply with the theory morphism $\sigma : \langle \Sigma, \Psi \rangle \to \langle \Sigma_2, \Psi_2 \rangle$ if $\Psi_2 \models_{\Sigma_2} \sigma(\Psi)$. In such case we have a pair $\mathcal{M}$ of interpretations satisfying Ψ_2, i.e. determining concrete SW-models which satisfy the multiplicity constraints and key constraints; which also satisfies $\sigma(\Psi)$. As before, the translation Φ maps types and roles in a consistent way, and also adds axioms constraining the $\mathcal{I}^{\mathrm{C}}$ models. In this sense, the pair of models satisfying $\Phi(\Psi_2)$ and $\Phi(\sigma)(\Phi(\Psi))$ must have the same structure than in the original interpretation $\mathcal{M}$. In this sense, if a formula φ^{K} in Ψ or Ψ_2 holds in $\mathcal{I}^{\mathrm{Q}^+}$, then its translation along Φ will also hold in $\mathcal{I}^{\mathrm{C}}$.

If we consider now transformation rules φ^{R}, we have that such rules does not constraint each individual interpretation but the relation between them. Reasoning as before, rules in Ψ_2 can be more specific than those in Ψ, such that any pair of interpretations satisfying the relation defined by those rules in Ψ_2 also satisfies the relation defined by $\sigma(\Psi)$. Since the translation Φ constraints the pair of models with respect to the transformation rules, any pair of models satisfying $\Phi(\Psi_2)$ will also satisfy $\Phi(\sigma)(\Phi(\Psi))$ given the fact that $\Phi(\sigma)$ maps types and roles in a consistent way without changing the required relation between models. Thus, we have that $\Phi(\Psi_2) \models_{\Phi(\Sigma_2)} \Phi(\sigma)(\Phi(\Psi))$ also hold with the inclusion of transformation rules.

Finally, $\Phi(\sigma)$ is a $\mathcal{I}^{\mathrm{C}}$ theory morphism.

□

Lemma 9.3.5. *The function* $\Phi : Th^{\mathcal{I}^{Q^+}} \rightarrow Th^{\mathcal{I}^{C}}$ *is a functor from the category of* $\mathcal{I}^{Q^+}$ *theories and theory morphisms to the category of theories in* $SubPCFOL^{=}$*, i.e.: (a) domain and codomain of the image of an arrow are the images of domain and codomain, respectively, of the arrow, (b) composition is preserved, and (c) identities are preserved.*

Proof. (a) By definition, the image of an arrow $\sigma : \Sigma_1 \rightarrow \Sigma_2$ in the category of $\mathcal{I}^{Q^+}$ theories is an arrow $\Phi(\sigma) : \Phi(\Sigma_1) \rightarrow \Phi(\Sigma_2)$ in the category of theories in $SubPCFOL^{=}$.

(b)We have to prove that $\Phi(\sigma_2 \circ \sigma_1) = \Phi(\sigma_2) \circ \Phi(\sigma_1)$. Since signature morphism are the disjoint union of extended CSMOF signature morphisms, we have that the composition is preserved by Lemma 9.2.5.

(c) As defined in Lemma 7.2.5, the identity signature morphism id_σ is the tuple with the identity signature morphisms of the corresponding institutions. Its translation $\Phi(id_\sigma)$ is the disjoint union of the translations of the identity signature morphisms. Thus, the translated signature morphism preserves the identity of sorts, functions and predicates, which is the identity signature morphism in the category of theories in $SubPCFOL^{=}$.

□

Lemma 9.3.6. *The function* $\beta : (\Phi)^{op}; \textbf{\textit{Mod}}^{\mathcal{I}^{C}} \rightarrow \textbf{\textit{Mod}}^{\mathcal{I}^{Q^+}}$*, with* ***Mod*** $: Th \rightarrow$ ***Cat*** *the functor giving the category of models of a theory, is a natural transformation, i.e. a family of arrows* $\beta_A : (\Phi)^{op}; \textbf{\textit{Mod}}^{\mathcal{I}^{C}}(A) \Rightarrow \textbf{\textit{Mod}}^{\mathcal{I}^{Q^+}}(A)$*, one for each theory* A *of* $\textbf{\textit{Sign}}^{Q}$*, such that, for every theory morphism* $\sigma : A \rightarrow B$ *it holds:* $\textbf{\textit{Mod}}^{\mathcal{I}^{Q^+}}(\sigma) \circ \beta_B = \beta_A \circ (\Phi)^{op}; \textbf{\textit{Mod}}^{\mathcal{I}^{C}}(\Phi(\sigma))$

Proof. Given any model in $\mathbf{Mod}^{\mathcal{I}^{C}}(\Phi(B))$, the translation β_B gives a pair of disjoint models. By definition of reduct, the application of $\mathbf{Mod}^{\mathcal{I}^{Q^+}}(\sigma)$ gives the same pair of models. In the other side, the reduct $\mathbf{Mod}^{\mathcal{I}^{C}}(\Phi(\sigma))$ gives an interpretation of symbols of the translated signature A, and the translation β_A just divide the model into two disjoint ones with respect to the two parts of the signature A. Thus, the property holds.

In the case of homomorphisms the property also holds, since the translation of a homomorphism is defined as a disjoint translation with respect to the elements in the corresponding signatures, and the reduct of a homomorphism is defined componentwise. This means that the translation gives the same homomorphism between elements as two disjoint functions, and it composition with the reduct gives a reduced homomorphism with respect to the elements in the source signature. This is equal to the reduct of the homomorphism with respect to the elements in the source signature and then its separation into two disjoint functions.

Finally, β is a natural transformation.

□

F Code Artifacts for the Running Example

In this appendix we present the code artifacts for the running example introduced in Section 2.2: (1) the XMI representation of UML class diagrams and Relation metamodels in Figure 2.6 and the corresponding SW-models in Figure 2.7 used for constructing the CSMOF institutions, (2) the QVT model transformation used for constructing the QVTR institution, and (3) the CASL theories generated by the comorphisms. CASL code has been commented.

In Section F.1 we first present the Ecore definition of CSMOF which allows representing CSMOF specifications as XMI files. Then, in Section F.2 we present the corresponding artifacts for UML class diagrams, in Section F.3 for Relational models, and in Section F.4 for the Class to Relational model transformation.

F.1 CSMOF Ecore

The Ecore model, strongly related to the EMOF standard, has a root object (EPackage) representing the whole model. This model has children representing classes (EClass), with zero or more attributes and zero or more references. Attributes (EAttribute) have a name and a type (represented by an EDataType). References (EReference) represent one end of an association between two classes. The following definition is almost self explanatory by looking at its graphical representation in Figure 10.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="CSMOF"
    nsURI="urn:CSMOF.ecore" nsPrefix="CSMOF">
```

```
<eClassifiers xsi:type="ecore:EClass"
    name="NamedElement" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="name" lowerBound="1"
    eType="ecore:EDataType
           http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="owner" ordered="false"
    eType="#//Metamodel" eOpposite="#//Metamodel/element"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Type"
  abstract="true" eSuperTypes="#//NamedElement"/>

<eClassifiers xsi:type="ecore:EClass" name="Class"
  eSuperTypes="#//Type">
  <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="isAbstract" lowerBound="1"
    eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EBoolean"
            defaultValueLiteral="false"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="ownedAttribute" ordered="false"
    upperBound="-1" eType="#//Property" containment="true"
    eOpposite="#//Property/class"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="superClass" ordered="false"
    upperBound="-1" eType="#//Class"
    eOpposite="#//Class/subClass"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="subClass" ordered="false"
    upperBound="-1" eType="#//Class"
    eOpposite="#//Class/superClass"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Property"
  eSuperTypes="#//MultiplicityElement #//TypedElement">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="opposite" ordered="false"
    eType="#//Property"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="class" lowerBound="1" eType="#//Class"
    eOpposite="#//Class/ownedAttribute"/>
</eClassifiers>
```

```
<eClassifiers xsi:type="ecore:EClass"
  name="MultiplicityElement" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="lower" lowerBound="1"
    eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore#//EInt"
                defaultValueLiteral="1"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="upper" lowerBound="1"
    eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore#//EInt"
                defaultValueLiteral="1"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass"
  name="DataType" eSuperTypes="#//Type"/>

<eClassifiers xsi:type="ecore:EClass" name="Metamodel">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="element" ordered="false"
    upperBound="-1" eType="#//NamedElement" containment="true"
    eOpposite="#//NamedElement/owner"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="name" lowerBound="1"
    eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="model" ordered="false"
    upperBound="-1" eType="#//Model" containment="true"
    eOpposite="#//Model/type"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="TypedElement"
  abstract="true" eSuperTypes="#//NamedElement">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="type" ordered="false" lowerBound="1"
    eType="#//Type"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Model">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="object" ordered="false"
    upperBound="-1" eType="#//Object" containment="true"
    eOpposite="#//Object/owner"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="link" ordered="false" upperBound="-1"
    eType="#//Link" containment="true"
    eOpposite="#//Link/owner"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name" lowerBound="1" eType="ecore:EDataType
              http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="type" lowerBound="1" eType="#//Metamodel"
      eOpposite="#//Metamodel/model"/>
  </eClassifiers>

  <eClassifiers xsi:type="ecore:EClass" name="Object">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="type" lowerBound="1" eType="#//Type"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name" lowerBound="1"
      eType="ecore:EDataType
              http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="owner" lowerBound="1" eType="#//Model"
      eOpposite="#//Model/object"/>
  </eClassifiers>

  <eClassifiers xsi:type="ecore:EClass" name="Link">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="type" lowerBound="1" eType="#//Property"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="source" lowerBound="1" eType="#//Object"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="target" lowerBound="1" eType="#//Object"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="owner" lowerBound="1" eType="#//Model"
      eOpposite="#//Model/link"/>
  </eClassifiers>
</ecore:EPackage>
```

F.2 UML Class Diagrams

In what follows we present the XMI file corresponding to the UML class diagrams metamodel and SW-model of the example with multiplicity constraints. As explained in Chapter 10, there is another XMI file without the multiplicity constraint. In this other file (not shown here) every property has a lower multiplicity of 0 (it may have no elements connected) and an upper multiplicity of -1 (it may have an unbounded number of elements connected). By default, the multiplicity is 1, so in these cases the lower and upper properties are not shown.

```
<?xml version="1.0" encoding="ASCII"?>
<CSMOF:Metamodel xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:CSMOF="urn:CSMOF.ecore"
  xsi:schemaLocation="urn:CSMOF.ecore ../metamodel/CSMOF.ecore"
  name="UML">

  <element xsi:type="CSMOF:Class" name="UMLModelElement"
      isAbstract="true"
      subClass="//@element.1 //@element.3 //@element.5">
    <ownedAttribute name="name" type="//@element.6"/>
    <ownedAttribute name="kind" type="//@element.6"/>
  </element>

  <element xsi:type="CSMOF:Class" name="Package"
      superClass="//@element.0">
    <ownedAttribute lower="0" upper="-1" name="elements"
      type="//@element.3"
      opposite="//@element.3/@ownedAttribute.0"/>
  </element>


  <element xsi:type="CSMOF:Class" name="PrimitiveDataType"
      superClass="//@element.3"/>

  <element xsi:type="CSMOF:Class" name="Classifier"
      superClass="//@element.0"
      subClass="//@element.2 //@element.4">
    <ownedAttribute name="namespace" type="//@element.1"
      opposite="//@element.1/@ownedAttribute.0"/>
  </element>

  <element xsi:type="CSMOF:Class" name="Class"
      superClass="//@element.3">
    <ownedAttribute upper="2" name="attribute"
      type="//@element.5"
      opposite="//@element.5/@ownedAttribute.0"/>
  </element>

  <element xsi:type="CSMOF:Class" name="Attribute"
      superClass="//@element.0">
    <ownedAttribute name="owner" type="//@element.4"
      opposite="//@element.4/@ownedAttribute.0"/>
    <ownedAttribute name="typeT" type="//@element.2"/>
  </element>
```

```
<element xsi:type="CSMOF:DataType" name="String"/>

<model name="UMLModel">
  <object type="//@element.1" name="p"/>
  <object type="//@element.2" name="pdt"/>
  <object type="//@element.5" name="a"/>
  <object type="//@element.4" name="c"/>
  <object type="//@element.6" name="Package"/>
  <object type="//@element.6" name="EMPTY"/>
  <object type="//@element.6" name="ID"/>
  <object type="//@element.6" name="Persistent"/>
  <object type="//@element.6" name="value"/>
  <object type="//@element.6" name="String"/>

  <link type="//@element.1/@ownedAttribute.0"
    source="//@model.0/@object.0"
    target="//@model.0/@object.3"/>
  <link type="//@element.1/@ownedAttribute.0"
    source="//@model.0/@object.0"
    target="//@model.0/@object.1"/>
  <link type="//@element.4/@ownedAttribute.0"
    source="//@model.0/@object.3"
    target="//@model.0/@object.2"/>
  <link type="//@element.5/@ownedAttribute.1"
    source="//@model.0/@object.2"
    target="//@model.0/@object.1"/>
  <link type="//@element.0/@ownedAttribute.0"
    source="//@model.0/@object.0"
    target="//@model.0/@object.4"/>

  <link type="//@element.0/@ownedAttribute.0"
    source="//@model.0/@object.3"
    target="//@model.0/@object.6"/>
  <link type="//@element.0/@ownedAttribute.0"
    source="//@model.0/@object.2"
    target="//@model.0/@object.8"/>
  <link type="//@element.0/@ownedAttribute.0"
    source="//@model.0/@object.1"
    target="//@model.0/@object.9"/>
  <link type="//@element.0/@ownedAttribute.1"
    source="//@model.0/@object.0"
    target="//@model.0/@object.5"/>
  <link type="//@element.0/@ownedAttribute.1"
    source="//@model.0/@object.3"
    target="//@model.0/@object.7"/>
  <link type="//@element.0/@ownedAttribute.1"
    source="//@model.0/@object.2"
    target="//@model.0/@object.5"/>
```

```
    <link type="//@element.0/@ownedAttribute.1"
      source="//@model.0/@object.1"
      target="//@model.0/@object.5"/>
  </model>
</CSMOF:Metamodel>
```

The CASL theory generated by the comorphism `CSMOF2CASL` is the following one.

```
%{Sorts and subsorting relations}%

sorts Attribute, Class, Classifier, Package, PrimitiveDataType,
      String, UMLModelElement

sorts Class, PrimitiveDataType < Classifier;
      Attribute, Classifier, Package < UMLModelElement

%{String elements}%
op EMPTY : String
op ID : String
op Package : String
op Persistent : String
op String : String
op value : String


%{Objects as total functions}%
op a : Attribute
op c : Class
op p : Package
op pdt : PrimitiveDataType

%{Properties: attributes and associations}%
pred attribute : Class * Attribute
pred elements : Package * Classifier
pred kind : UMLModelElement * String
pred name : UMLModelElement * String
pred namespace : Classifier * Package
pred owner : Attribute * Class
pred typeT : Attribute * PrimitiveDataType

%{Multiplicity constraints}%

. (forall x_1 : UMLModelElement
   . exists y_1 : String . name(x_1, y_1))
  /\ forall x_1 : UMLModelElement; y_2, y_1 : String
     . (name(x_1, y_2) /\ name(x_1, y_1)) => y_2 = y_1
```

```
. (forall x_1 : UMLModelElement
   . exists y_1 : String . kind(x_1, y_1))
  /\ forall x_1 : UMLModelElement; y_2, y_1 : String
     . (kind(x_1, y_2) /\ kind(x_1, y_1)) => y_2 = y_1


. (forall x_1 : Classifier
   . exists y_1 : Package . namespace(x_1, y_1))
  /\ forall x_1 : Classifier; y_2, y_1 : Package
     . (namespace(x_1, y_2) /\ namespace(x_1, y_1))
         => y_2 = y_1

. forall x_1 : Class; y_3, y_2, y_1 : Attribute
  . (attribute(x_1, y_3) /\ attribute(x_1, y_2)
      /\ attribute(x_1, y_1))
    => y_3 = y_2 \/ y_3 = y_1 \/ y_2 = y_1

. forall x_1 : Class
  . exists y_1 : Attribute . attribute(x_1, y_1)

. (forall x_1 : Attribute . exists y_1 : Class
  . owner(x_1, y_1)) /\
  forall x_1 : Attribute; y_2, y_1 : Class
    . (owner(x_1, y_2) /\ owner(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Attribute
   . exists y_1 : PrimitiveDataType . typeT(x_1, y_1))
  /\ forall x_1 : Attribute; y_2, y_1 : PrimitiveDataType
     . (typeT(x_1, y_2) /\ typeT(x_1, y_1)) => y_2 = y_1

%{Axioms}%

forall x : Class; y : Attribute . attribute(x, y) <=> owner(y, x)
                                        %(equiv_owner_attribute)%

forall x : Package; y : Classifier
. elements(x, y) <=> namespace(y, x)%(equiv_namespace_elements)%

forall x : Classifier; y : Package
. namespace(x, y) <=> elements(y, x)%(equiv_elements_namespace)%

forall x : Attribute; y : Class . owner(x, y) <=> attribute(y, x)
                                        %(equiv_attribute_owner)%

forall x : Attribute; y : PrimitiveDataType
. typeT(x, y) <=> (x = a /\ y = pdt)
                    %(compRel__AttributetypeTPrimitiveDataType)%
```

```
forall x : UMLModelElement; y : String
. kind(x, y)
  <=> (x = a /\ y = EMPTY) \/ (x = c /\ y = Persistent)
      \/ (x = p /\ y = EMPTY) \/ (x = pdt /\ y = EMPTY)
                          %(compRel__UMLModelElementkindString)%

forall x : UMLModelElement; y : String
. name(x, y)
  <=> (x = a /\ y = value) \/ (x = c /\ y = ID)
      \/ (x = p /\ y = Package) \/ (x = pdt /\ y = String)
                          %(compRel__UMLModelElementnameString)%

forall x : Package; y : Classifier
. elements(x, y) <=> (x = p /\ y = c) \/ (x = p /\ y = pdt)
                  %(compRel_namespacePackageelementsClassifier)%

forall x : Class; y : Attribute
. attribute(x, y) <=> (x = c /\ y = a)
                        %(compRel_ownerClassattributeAttribute)%

%{Sort generation constraints}%

%% free
generated type UMLModelElement ::=
                 sort Attribute | sort Classifier | sort Package
                                                  %(disjEmbedd)%

%% free
generated type Attribute ::= a          %(sortGenCon_Attribute)%

%% free
generated type Class ::= c                 %(sortGenCon_Class)%

%% free
generated type Package ::= p              %(sortGenCon_Package)%

%% free
generated type PrimitiveDataType ::= pdt
                               %(sortGenCon_PrimitiveDataType)%

%% free
generated type
String ::= EMPTY | ID | Package | Persistent | String | value
                                          %(sortGenCon_String)%

forall x : Classifier . x = c \/ x = pdt
                                      %(sortGenCon_Classifier)%
```

```
. not a = c /\ not a = p /\ not a = pdt
. not c = p /\ not c = pdt
. not p = pdt                    %(noConfusion_UMLModelElement)%


. not EMPTY = ID /\ not EMPTY = Package
  /\ not EMPTY = Persistent /\ not EMPTY = String
  /\ not EMPTY = value /\ not ID = Package
  /\ not ID = Persistent /\ not ID = String
  /\ not ID = value /\ not Package = String
  /\ not Package = Persistent /\ not Package = value
  /\ not Persistent = String /\ not Persistent = value
  /\ not String = value
                                          %(noConfusion_String)%
```

F.3 Relational Models

In what follows we present the XMI file corresponding to the Relational metamodel and SW-model of the example with multiplicity constraints. Same comments as in the last section apply.

```
<?xml version="1.0" encoding="ASCII"?>
<CSMOF:Metamodel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:CSMOF="urn:CSMOF.ecore"
    xsi:schemaLocation="urn:CSMOF.ecore ../metamodel/CSMOF.ecore"
    name="RDBMS">

  <element xsi:type="CSMOF:Class" name="RDBMSModelElement"
      isAbstract="true"
      subClass="//@element.1 //@element.2
                //@element.3 //@element.4">
    <ownedAttribute name="name" type="//@element.5"/>
  </element>

  <element xsi:type="CSMOF:Class" name="Schema"
      superClass="//@element.0">
    <ownedAttribute lower="0" upper="-1" name="tables"
      type="//@element.2"
      opposite="//@element.2/@ownedAttribute.0"/>
  </element>
```

```
<element xsi:type="CSMOF:Class" name="Table"
    superClass="//@element.0">
  <ownedAttribute name="schema" type="//@element.1"
    opposite="//@element.1/@ownedAttribute.0"/>
  <ownedAttribute lower="0" upper="-1" name="column"
    type="//@element.3"/>
  <ownedAttribute name="keyK" type="//@element.4"
    opposite="//@element.4/@ownedAttribute.1"/>
</element>

<element xsi:type="CSMOF:Class" name="Column"
  superClass="//@element.0">
  <ownedAttribute name="typeT" type="//@element.5"/>
  <ownedAttribute name="owner" type="//@element.2"
    opposite="//@element.2/@ownedAttribute.1"/>
  <ownedAttribute lower="0" upper="-1" name="keyK"
    type="//@element.4"
    opposite="//@element.4/@ownedAttribute.0"/>
</element>

<element xsi:type="CSMOF:Class" name="Key"
    superClass="//@element.0">
  <ownedAttribute name="column" type="//@element.3"
    opposite="//@element.3/@ownedAttribute.2"/>
  <ownedAttribute name="owner" type="//@element.2"
    opposite="//@element.2/@ownedAttribute.2"/>
</element>

<element xsi:type="CSMOF:DataType" name="String"/>
<model name="RDBMSModel">
  <object type="//@element.1" name="s"/>
  <object type="//@element.2" name="t"/>
  <object type="//@element.4" name="k"/>
  <object type="//@element.3" name="c1"/>
  <object type="//@element.3" name="c2"/>
  <object type="//@element.5" name="Package"/>
  <object type="//@element.5" name="ID"/>
  <object type="//@element.5" name="PK"/>
  <object type="//@element.5" name="TID"/>
  <object type="//@element.5" name="value"/>
  <object type="//@element.5" name="VARCHAR"/>
  <object type="//@element.5" name="NUMBER"/>

  <link type="//@element.1/@ownedAttribute.0"
    source="//@model.0/@object.0"
    target="//@model.0/@object.1"/>
```

```
    <link type="//@element.2/@ownedAttribute.2"
      source="//@model.0/@object.1"
      target="//@model.0/@object.2"/>
    <link type="//@element.2/@ownedAttribute.1"
      source="//@model.0/@object.1"
      target="//@model.0/@object.3"/>
    <link type="//@element.2/@ownedAttribute.1"
      source="//@model.0/@object.1"
      target="//@model.0/@object.4"/>
    <link type="//@element.3/@ownedAttribute.2"
      source="//@model.0/@object.3"
      target="//@model.0/@object.2"/>
    <link type="//@element.0/@ownedAttribute.0"
      source="//@model.0/@object.0"
      target="//@model.0/@object.5"/>
    <link type="//@element.0/@ownedAttribute.0"
      source="//@model.0/@object.1"
      target="//@model.0/@object.6"/>
    <link type="//@element.0/@ownedAttribute.0"
      source="//@model.0/@object.2"
      target="//@model.0/@object.7"/>
    <link type="//@element.0/@ownedAttribute.0"
      source="//@model.0/@object.3"
      target="//@model.0/@object.8"/>
    <link type="//@element.0/@ownedAttribute.0"
      source="//@model.0/@object.4"
      target="//@model.0/@object.9"/>
    <link type="//@element.3/@ownedAttribute.0"
      source="//@model.0/@object.3"
      target="//@model.0/@object.11"/>
    <link type="//@element.3/@ownedAttribute.0"
      source="//@model.0/@object.4"
      target="//@model.0/@object.10"/>
  </model>
</CSMOF:Metamodel>
```

The CASL theory generated by the comorphism `CSMOF2CASL` is the following one.

```
%{Sorts and subsorting relations}%
sorts Column, Key, RDBMSModelElement, Schema, String, Table
sorts Column, Key, Schema, Table < RDBMSModelElement

%{String elements}%
op ID : String
op NUMBER : String
op PK : String
op Package : String
```

```
op TID : String
op VARCHAR : String
op value : String

%{Objects as total functions}%
op c1 : Column
op c2 : Column
op k : Key
op s : Schema
op t : Table

%{Properties: attributes and associations}%
pred column : Key * Column
pred column : Table * Column
pred keyK : Column * Key
pred keyK : Table * Key
pred name : RDBMSModelElement * String
pred owner : Column * Table
pred owner : Key * Table
pred schema : Table * Schema
pred tables : Schema * Table
pred typeT : Column * String

%{Multiplicity constraints}%
. (forall x_1 : RDBMSModelElement
   . exists y_1 : String . name(x_1, y_1))
  /\ forall x_1 : RDBMSModelElement; y_2, y_1 : String
     . (name(x_1, y_2) /\ name(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Table . exists y_1 : Schema . schema(x_1, y_1))
  /\ forall x_1 : Table; y_2, y_1 : Schema
     . (schema(x_1, y_2) /\ schema(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Table . exists y_1 : Key . keyK(x_1, y_1))
  /\ forall x_1 : Table; y_2, y_1 : Key
     . (keyK(x_1, y_2) /\ keyK(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Column . exists y_1 : String . typeT(x_1, y_1))
  /\ forall x_1 : Column; y_2, y_1 : String
     . (typeT(x_1, y_2) /\ typeT(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Column . exists y_1 : Table . owner(x_1, y_1))
  /\ forall x_1 : Column; y_2, y_1 : Table
     . (owner(x_1, y_2) /\ owner(x_1, y_1)) => y_2 = y_1

. (forall x_1 : Key . exists y_1 : Column . column(x_1, y_1))
  /\ forall x_1 : Key; y_2, y_1 : Column
     . (column(x_1, y_2) /\ column(x_1, y_1)) => y_2 = y_1
```

```
. (forall x_1 : Key . exists y_1 : Table . owner(x_1, y_1))
  /\ forall x_1 : Key; y_2, y_1 : Table
     . (owner(x_1, y_2) /\ owner(x_1, y_1)) => y_2 = y_1

%{Axioms}%

forall x : Key; y : Column . column(x, y) <=> keyK(y, x)
                                          %(equiv_keyK_column)%

forall x : Table; y : Column . column(x, y) <=> owner(y, x)
                                         %(equiv_owner_column)%

forall x : Column; y : Key . keyK(x, y) <=> column(y, x)
                                          %(equiv_column_keyK)%

forall x : Table; y : Key . keyK(x, y) <=> owner(y, x)
                                           %(equiv_owner_keyK)%

forall x : Key; y : Table . owner(x, y) <=> keyK(y, x)
                                           %(equiv_keyK_owner)%

forall x : Table; y : Schema . schema(x, y) <=> tables(y, x)
                                         %(equiv_tables_schema)%

forall x : Schema; y : Table . tables(x, y) <=> schema(y, x)
                                         %(equiv_schema_tables)%

forall x : Column; y : String
. typeT(x, y) <=>
    (x = c1 /\ y = NUMBER) \/ (x = c2 /\ y = VARCHAR)
                                  %(compRel__ColumntypeTString)%

forall x : RDBMSModelElement; y : String
. name(x, y)
  <=> (x = c1 /\ y = TID) \/ (x = c2 /\ y = value)
      \/ (x = k /\ y = PK) \/ (x = s /\ y = Package)
      \/ (x = t /\ y = ID)
                        %(compRel__RDBMSModelElementnameString)%

forall x : Table; y : Column
. column(x, y) <=> (x = t /\ y = c1) \/ (x = t /\ y = c2)
                                  %(compRel__TablecolumnColumn)%

forall x : Column; y : Key . keyK(x, y) <=> (x = c1 /\ y = k)
                                 %(compRel_columnColumnkeyKKey)%
```

```
forall x : Table; y : Key . keyK(x, y) <=> (x = t /\ y = k)
                                    %(compRel_ownerTablekeyKKey)%

forall x : Schema; y : Table . tables(x, y) <=> (x = s /\ y = t)
                                %(compRel_schemaSchematablesTable)%

%{Sort generation constraints}%

%% free
generated type RDBMSModelElement ::=
               sort Column | sort Key | sort Schema | sort Table
                                                    %(disjEmbedd)%

%% free
generated type Column ::= c1 | c2           %(sortGenCon_Column)%

%% free
generated type Key ::= k                       %(sortGenCon_Key)%

%% free
generated type Schema ::= s                  %(sortGenCon_Schema)%

%% free
generated type String ::=
              ID | NUMBER | PK | Package | TID | VARCHAR | value
                                            %(sortGenCon_String)%

%% free
generated type Table ::= t                    %(sortGenCon_Table)%

. not c1 = c2 /\ not c1 = k /\ not c2 = k /\ not c1 = s
  /\ not c2 = s /\ not c1 = t /\ not c2 = t
                                          %(noConfusion_Column)%

. not k = s /\ not k = t                       %(noConfusion_Key)%

. not s = t                                 %(noConfusion_Schema)%

. not ID = NUMBER /\ not ID = PK /\ not ID = Package
  /\ not ID = TID /\ not ID = VARCHAR /\ not ID = value
  /\ not NUMBER = PK /\ not NUMBER = Package
  /\ not NUMBER = TID /\ not NUMBER = VARCHAR
  /\ not NUMBER = value /\ not PK = Package
  /\ not PK = TID /\ not PK = VARCHAR /\ not PK = value
  /\ not Package = TID /\ not Package = VARCHAR
  /\ not Package = value /\ not TID = VARCHAR
  /\ not TID = value /\ not VARCHAR = value
                                          %(noConfusion_String)%
```

F.4 Class to Relational Transformation

In what follows we present the QVT file corresponding to the model transformation. It is the same transformation introduced in Section 2.2 but with modified `where` clauses for using our simple expressions language, as explained in Section 10.3.

```
transformation uml2rdbms  ( uml : UML , rdbms : RDBMS ) {
  key RDBMS::Table {name, schema};
  key RDBMS::Column {name, owner};
  key RDBMS::Key {name, owner};

  top relation PackageToSchema {
    pn : String;

    checkonly domain uml p : UML::Package {
      name = pn
    };

    enforce domain rdbms s : RDBMS::Schema {
      name = pn
    };
  }

  top relation ClassToTable {
    cn, prefix : String;

    checkonly domain uml c : UML::Class {
      namespace = p : UML::Package { },
      kind = 'Persistent',
      name = cn
    };

    enforce domain rdbms t : RDBMS::Table {
      schema = s : RDBMS::Schema { },
      name = cn,
      column = cl : RDBMS::Column {
        name = 'TID',
        typeT = 'NUMBER'
      },
      keyK = k : RDBMS::Key {
        name = 'PK',
        column = cl
      }
    };
```

```
    when {
      PackageToSchema(p, s);
    }

    where {
      AttributeToColumn(c, t, prefix);

      = (prefix) ('EMPTY');
    }
  }

  relation AttributeToColumn {
    an, pn, cn, sqltype : String;

    primitive domain prefix : String;

    checkonly domain uml c : UML::Class {
      attribute = a : UML::Attribute {
        name = an,
        typeT = p : UML::PrimitiveDataType {
          name = pn
        }
      }
    };

    enforce domain rdbms t : RDBMS::Table {
      column = cl : RDBMS::Column {
        name = cn,
        typeT = sqltype
      }
    };

    where {
      or (and (= (prefix)('EMPTY')) (= (cn)(an)))
         (and (not (= (prefix)('EMPTY')))
              (= (cn)(prefix + an)));

      or (or (and (= (pn)('INTEGER')) (= (sqltype)('NUMBER')))
             (and (= (pn)('BOOLEAN')) (= (sqltype)('BOOLEAN'))))
             (and (and (not (= (pn)('INTEGER')))
                       (not (= (pn)('BOOLEAN'))))
                  (= (sqltype)('VARCHAR')));
    }
  }
}
```

The CASL theory generated by the comorphism QVTR2CASL includes the theories of the source and target metamodels (without multiplicity constraints) and SW-models generated by the comorphism CSMOF2CASL using the same XMI files of the last sections. These fragments are omitted in the following code.

```
%{String concatenation and new strings}%
op ++ : String * String -> String
op BOOLEAN : String
op INTEGER : String

%{Rules declarations}%
pred AttributeToColumn : Class * Table * String
pred ClassToTable : Class * Table
pred PackageToSchema : Package * Schema
pred Top_ClassToTable : ()
pred Top_PackageToSchema : ()

%{Keys declarations}%
pred key_Column : ()
pred key_Key : ()
pred key_Table : ()

%{Keys definitions}%

. key_Table
  <=> forall x_2, x_1 : Table; y_2 : String; y_1 : Schema
      . not x_2 = x_1
        => name(x_1, y_2) /\ schema(x_1, y_1)
           => not name(x_2, y_2) \/ not schema(x_2, y_1)

. key_Column
  <=> forall x_2, x_1 : Column; y_2 : String; y_1 : Table
      . not x_2 = x_1
        => name(x_1, y_2) /\ owner(x_1, y_1)
           => not name(x_2, y_2) \/ not owner(x_2, y_1)

. key_Key
  <=> forall x_2, x_1 : Key; y_2 : String; y_1 : Table
      . not x_2 = x_1
        => name(x_1, y_2) /\ owner(x_1, y_1)
           => not name(x_2, y_2) \/ not owner(x_2, y_1)
```

```
%{Rules definitions}%

. Top_PackageToSchema
  <=> forall p : Package; pn : String
      . name((var p : Package), pn)
        => exists s : Schema . name((var s : Schema), pn)

forall p : Package; s : Schema
. PackageToSchema((var p : Package), (var s : Schema))
  <=> forall pn : String
      . name((var p : Package), pn)
          => name((var s : Schema), pn)


. Top_ClassToTable
  <=> forall p : Package; s : Schema
      . PackageToSchema((var p : Package), (var s : Schema))
        => forall c : Class; cn : String
           . namespace((var c : Class), (var p : Package))
             /\ kind((var c : Class), Persistent)
             /\ name((var c : Class), cn)
             => exists cl : Column; k : Key;
                       prefix : String; t : Table
                . (schema((var t : Table), (var s : Schema))
                   /\ name((var t : Table), cn)
                   /\ column((var t : Table), cl)
                   /\ keyK((var t : Table), (var k : Key))
                   /\ name(cl, TID)
                   /\ typeT(cl, NUMBER)
                   /\ name((var k : Key), PK)
                   /\ column((var k : Key), cl))
                   /\ AttributeToColumn((var c : Class),
                                        (var t : Table),
                                        prefix)
                   /\ prefix = EMPTY

forall c : Class; t : Table
. ClassToTable((var c : Class), (var t : Table))
  <=> forall p : Package; s : Schema
      . PackageToSchema((var p : Package), (var s : Schema))
        => forall cn : String
           . namespace((var c : Class), (var p : Package))
             /\ kind((var c : Class), Persistent)
             /\ name((var c : Class), cn)
             => exists cl : Column; k : Key; prefix : String
                . (schema((var t : Table), (var s : Schema))
                   /\ name((var t : Table), cn)
                   /\ column((var t : Table), cl)
```

```
                    /\ keyK((var t : Table), (var k : Key))
                    /\ name(cl, TID)
                    /\ typeT(cl, NUMBER)
                    /\ name((var k : Key), PK)
                    /\ column((var k : Key), cl))
                    /\ AttributeToColumn((var c : Class),
                                         (var t : Table),
                                         prefix)
                    /\ prefix = EMPTY

forall c : Class; t : Table; prefix : String
. AttributeToColumn((var c : Class), (var t : Table), prefix)
  <=> forall a : Attribute; p : PrimitiveDataType; an : String;
      pn : String
      . attribute((var c : Class), (var a : Attribute))
        /\ name((var a : Attribute), an)
        /\ typeT((var a : Attribute), p)
        /\ name((var p : PrimitiveDataType), pn)
        => exists cl : Column; cn : String; sqltype : String
           . (column((var t : Table), cl) /\ name(cl, cn)
              /\ typeT(cl, sqltype))
             /\ (((prefix = EMPTY /\ cn = an)
                  \/ (not prefix = EMPTY /\ cn = ++(prefix, an)))
                 /\ (((pn = INTEGER /\ sqltype = NUMBER)
                      \/ (pn = BOOLEAN /\ sqltype = BOOLEAN))
                     \/ ((not pn = INTEGER /\ not pn = BOOLEAN)
                         /\ sqltype = VARCHAR)))

%{Since String is a primitive type, it is in both source and
  target domains. We redefine here the "no junk, no confusion"
  principle }%

. not BOOLEAN = EMPTY /\ not BOOLEAN = ID
  /\ not BOOLEAN = INTEGER /\ not BOOLEAN = NUMBER
  /\ not BOOLEAN = PK /\ not BOOLEAN = Package
  /\ not BOOLEAN = Persistent /\ not BOOLEAN = String
  /\ not BOOLEAN = TID /\ not BOOLEAN = VARCHAR
  /\ not BOOLEAN = value /\ not EMPTY = ID /\ not EMPTY = INTEGER
  /\ not EMPTY = NUMBER /\ not EMPTY = PK /\ not EMPTY = Package
  /\ not EMPTY = Persistent /\ not EMPTY = String
  /\ not EMPTY = TID /\ not EMPTY = VARCHAR /\ not EMPTY = value
  /\ not ID = INTEGER /\ not ID = NUMBER /\ not ID = PK
  /\ not ID = Package /\ not ID = Persistent /\ not ID = String
  /\ not ID = TID /\ not ID = VARCHAR /\ not ID = value
  /\ not INTEGER = NUMBER /\ not INTEGER = PK
  /\ not INTEGER = Package /\ not INTEGER = Persistent
  /\ not INTEGER = String /\ not INTEGER = TID
  /\ not INTEGER = VARCHAR /\ not INTEGER = value
```

```
  /\ not NUMBER = PK /\ not NUMBER = Package
  /\ not NUMBER = Persistent /\ not NUMBER = String
  /\ not NUMBER = TID /\ not NUMBER = VARCHAR
  /\ not NUMBER = value /\ not PK = Package
  /\ not PK = Persistent /\ not PK = String /\ not PK = TID
  /\ not PK = VARCHAR /\ not PK = value
  /\ not Package = Persistent /\ not Package = String
  /\ not Package = TID /\ not Package = VARCHAR
  /\ not Package = value /\ not Persistent = String
  /\ not Persistent = TID /\ not Persistent = VARCHAR
  /\ not Persistent = value /\ not String = TID
  /\ not String = VARCHAR /\ not String = value
  /\ not TID = VARCHAR /\ not TID = value
  /\ not VARCHAR = value          %(noConfusion_String_77)%

%% free
generated type
String
::= ++(String; String)
  | BOOLEAN
  | EMPTY
  | ID
  | INTEGER
  | NUMBER
  | PK
  | Package
  | Persistent
  | String
  | TID
  | VARCHAR
  | value                          %(sortGenCon_String_86)%
```

References

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of model transformations via Alloy. In Benoit Baudry, Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[AKP03] David Akehurst, Stuart Kent, and Octavian Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.

[ALL10] Márk Asztalos, László Lengyel, and Tihamer Levendovszky. Towards automated, formal verification of model transformations. In *ICST*, pages 15–24. IEEE Computer Society, 2010.

[ALS+12] Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *ICST*, pages 921–928. IEEE, 2012.

[ARW13] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, pages 1–26, 2013.

[ATL05] ATLAS. *KM3: Kernel MetaMetaModel*. LINA & INRIA, Manual v0.3 edition, 2005.

[BBG+06a] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 72–81. ACM, 2006.

[BBG+06b] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? transformation models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCG11] Fabian Büttner, Jordi Cabot, and Martin Gogolla. On validation of ATL transformation rules by transformation models. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, pages 9:1–9:8. ACM, 2011.

[BEC12] Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 432–448. Springer, 2012.

[BEH06] Luciano Baresi, Karsten Ehrig, and Reiko Heckel. Verification of model transformations: A case study with BPEL. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 183–199. Springer, 2006.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[BHM09] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[BKMW08] Artur Boronat, Alexander Knapp, José Meseguer, and Martin Wirsing. What is a multi-modeling language? In Corradini and Montanari [CM09], pages 71–87.

[BKS02] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the object constraint language into first-order predicate logic. In *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC)*, pages 113–123, 2002.

[BLA+10] Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A turing incomplete transformation language. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *SLE*, volume 6563 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 2010.

[BM09] Artur Boronat and José Meseguer. Algebraic semantics of OCL-constrained metamodel specifications. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 96–115. Springer, 2009.

[BMC+11] Christiano Braga, Roberto Menezes, Thiago Comicio, Cassio Santos, and Edson Landim. On the specification, verification and implementation of model transformations with transformation contracts. In Adenilso da Silva Simão and Carroll Morgan, editors, *SBMF*, volume 7021 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2011.

[Bru06] Jean-Michel Bruel, editor. *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*. Springer, 2006.

[Bru08] Harrie Jan Sander Bruggink. Towards a systematic method for proving termination of graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 213(1):23–38, 2008.

[CBBD09] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL contracts for the verification of model transformations. *ECEASST*, 24, 2009.

[CCGdL10] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[CCGT09] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on semantics definition in MDE - an instrumented approach for model verification. *JSW*, 4(9):943–958, 2009.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[CGR12] Andrea Corradini, Fabio Gadducci, and Leila Ribeiro. An institution for graph transformation. In Mossakowski and Kreowski, editors, WADT, volume 7137 of Lecture Notes in Computer Science, pages 160–174. Springer, 2012

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[Cha06] Kenneth Chan. Formal proofs for QoS-oriented transformations. In *EDOC Workshops*, page 41. IEEE Computer Society, 2006.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *ASE*, pages 267–270. IEEE Computer Society, 2002.

[CK08] María Victoria Cengarle and Alexander Knapp. An institution for UML 2.0 static structures. Technical Report TUM-I0807, Institut für Informatik, Technische Universität München, 2008.

[CKTW08] María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A heterogeneous approach to UML semantics. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 383–402. Springer, 2008.

[CLMM09] Mihai Codescu, Bruno Langenstein, Christian Maeder, and Till Mossakowski. The VSE refinement method in hets. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 660–678. Springer, 2009.

[CLST09] Daniel Calegari, Carlos Luna, Nora Szasz, and Alvaro Tasistro. Experiment with a type-theoretic approach to the verification of model transformations. In *Proceedings of the II Workshop Chileno de Métodos Formales (ChWFM'09), Chile*, 2009.

[CLST10a] Daniel Calegari, Carlos Luna, Nora Szasz, and Alvaro Tasistro. Representation of metamodels using inductive types in a type-theoretic framework for MDE. Technical Report RT10-01, InCo-PEDECIBA. ISSN 0797-6410, 2010.

[CLST10b] Daniel Calegari, Carlos Luna, Nora Szasz, and Alvaro Tasistro. A type-theoretic framework for certified model transformations. In Jim Davies, Leila Silva, and Adenilso da Silva Simão, editors, *SBMF*, volume 6527 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2010.

[CM09] Andrea Corradini and Ugo Montanari, editors. *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Revised Selected Papers*, volume 5486 of *Lecture Notes in Computer Science*. Springer, 2009.

[CMRM10] Mihai Codescu, Till Mossakowski, Adrián Riesco, and Christian Maeder. Integrating Maude into Hets. In Michael Johnson and Dusko Pavlovic, editors, *AMAST*, volume 6486 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2010.

[CNS12] Marsha Chechik, Shiva Nejati, and Mehrdad Sabetzadeh. A relationship-based approach to model integration. *ISSE*, 8(1):3–18, 2012.

[Cod08] Mihai Codescu. Generalized theoroidal institution comorphisms. In Corradini and Montanari [CM09], pages 88–101.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[CS11a] Daniel Calegari and Nora Szasz. An institution for UML 2.0 state machines. Technical Report RT11-02, InCo-PEDECIBA, 2011.

[CS11b] Daniel Calegari and Nora Szasz. Institutionalising UML 2.0 state machines. *ISSE*, 7(4):315–323, 2011.

[CS12] Daniel Calegari and Nora Szasz. Verification of model transformations: A survey of the state-of-the-art (extended version). Technical Report RT12-05, InCo-PEDECIBA, 2012.

[CS13a] Daniel Calegari and Nora Szasz. Bridging techological spaces for the verification of model transformations. In *Proc. Conf. Iberoamericana de Software Engineering (CIbSE)*, 2013.

[CS13b] Daniel Calegari and Nora Szasz. Institution-based semantics for MOF and QVT-relations. In Juliano Iyoda and Leonardo Mendonça de Moura, editors, *SBMF*, volume 8195 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2013.

[CS13c] Daniel Calegari and Nora Szasz. Institution-based semantics for MOF and QVT-Relations (extended version). Technical Report TR13-06, InCo-PEDECIBA. ISSN 0797-6410, May 2013.

[CS13d] Daniel Calegari and Nora Szasz. Verification of model transformations: A survey of the state-of-the-art. *Electr. Notes Theor. Comput. Sci.*, 292:5–25, 2013.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[Dia02] Razvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402, 2002.

[Dia13] Razvan Diaconescu. Institutional semantics for many-valued logics. *Fuzzy Sets and Systems*, 218:32–52, 2013.

[dLG09] Juan de Lara and Esther Guerra. Formal support for QVT-relations with coloured petri nets. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2009.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[EHRT08] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*. Springer, 2008.

[EKHL03] Gregor Engels, Jochen Malte Küster, Reiko Heckel, and Marc Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[Fav09] Liliana Favre. A formal foundation for metamodeling. In Fabrice Kordon and Yvon Kermarrec, editors, *Ada-Europe*, volume 5570 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2009.

[For13] N. Fornaro. Interpretación de KM3/ATL en Teoría de Tipos. Master Thesis, Universidad ORT, Uruguay, 2013.

[FSM+03] F.Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003.

[GB83] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In Edmund M. Clarke and Dexter Kozen, editors, *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer, 1983.

[GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GdL12] Esther Guerra and Juan de Lara. An algebraic semantics for QVT-Relations check-only transformations. *Fundam. Inform.*, 114(1):73–101, 2012.

[GdLW+13] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.

[GGL+06] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner. Towards verified model transformations. In David Hearnden, Jörn Guy Süß, Benoit Baudry, and Nicolas Rapin, editors, *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification*, pages 78–93. Le Commissariat a l'Energie Atomique - CEA, October 2006.

[GM07] Miguel García and Ralf Möller. Certification of transformation algorithms in model-driven software development. In Wolf-Gideon Bleek, Jörg Raasch, and Heinz Züllighoven, editors, *Software Engineering*, volume 105 of *LNI*, pages 107–118. GI, 2007.

[GMLF14] Manuel Giménez, Mariano Moscato, Carlos López and Marcelo Frias. HeteroGenius: A Framework for Hybrid Analysis of Heterogeneous Software Specifications. *1st Latin American Workshop on Formal Methods. LAFM, 2008. Proceedings*, volume 139 of *EPTCS*. pages 65–70, 2014.

[GR02] Joseph A. Goguen and Grigore Rosu. Institution morphisms. *Formal Asp. Comput.*, 13(3-5):274–307, 2002.

[HEOG10] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *ICGT*, volume 6372 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2010.

[HET08] Frank Hermann, Hartmut Ehrig and Gabriele Taentzer. A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. *Electronic Notes in Theoretical Computer Science*, 211(28):261–269, 2008.

[HHK10] Frank Hermann, Mathias Hülsbusch, and Barbara König. Specification and verification of model transformations. *ECEASST*, 30, 2010.

[HJC+08] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Commun. ACM*, 51(9):54–59, 2008.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002.

[HS02] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[HT05] Reiko Heckel and Sebastian Thöne. Behavioral refinement of graph transformation-based models. *Electr. Notes Theor. Comput. Sci.*, 127(3):101–111, 2005.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[IKV13] IKV++ Technologies. mediniQVT. http://projects.ikv.de/qvt, apr 2014.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Bruel [Bru06], pages 128–138.

[JKMR12] Phillip James, Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Designing domain specific languages - a craftsman's approach for the railway domain using Casl. In Narciso Martí-Oliet and Miguel Palomino, editors, *WADT*, volume 7841 of *Lecture Notes in Computer Science*, pages 178–194. Springer, 2012.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[KAER06] Jochen Malte Küster and Mohamed Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2006.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[Ken02] Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[KMS+09] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic transformation development. *ECEASST*, 21, 2009.

[Küs04] Jochen Malte Küster. Systematic validation of model transformations. In *Proceedings of WiSME'04 (associated to UML'04)*, 2004.

[Küs06] Jochen Malte Küster. Definition and validation of model transformations. *Software and System Modeling*, 5(3):233–259, 2006.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lan98] Saunders M. Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, 2nd edition, September 1998.

[LBA10] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010.

[LBEE+06] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange and Gabriele Taentzer. Attributed graph transformation with node type inheritance. *Fundamental Aspects of Software Engineering*, 376(3):139–163, 2007.

[LD10] Hung Ledang and Hubert Dubois. Proving model transformations. In Jing Liu, Doron Peled, Bow-Yaw Wang, and Farn Wang, editors, *TASE*, pages 35–44. IEEE Computer Society, 2010.

[LEO06] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient detection of conflicts in graph-based model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:97–109, 2006.

[LLM09] Tihamer Levendovszky, László Lengyel, and Tamás Mészáros. Supporting domain-specific model patterns with metamodeling. *Software and System Modeling*, 8(4):501–520, 2009.

[LMAL10] Laszlo Lengyel, Istvan Madari, Mark Asztalos, and Tihamer Levendovszky. Validating Query/View/Transformation relations. *Model-Driven Engineering, Verification, and Validation, Workshop on*, 0:7–12, 2010.

[LR10] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Specification and verification of model transformations using UML-RSDS. In Dominique Méry and Stephan Merz, editors, *IFM*, volume 6396 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2010.

[LR11] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Model-driven development of model transformations. In Jordi Cabot and Eelco Visser, editors, *ICMT*, volume 6707 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2011.

[LR12] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Model transformation specification and design. *Advances in Computers*, 85:123–163, 2012.

[LS05] Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In Bruel [Bru06], pages 139–150.

[LVV+09a] Horacio Lopez, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, and Carlos Luna. Estado del arte de lenguajes y herramientas de transformación de modelos. Technical Report RT09-19, InCo-PEDECIBA. ISSN 0797-6410, 2009.

[LVV+09b] Horacio Lopez, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, and Carlos Luna. Estado del arte de verificación de transformación de modelos. Technical Report RT09-19, InCo-PEDECIBA. ISSN 0797-6410, 2009.

[MAH06] Till Mossakowski, Serge Autexier, and Dieter Hutter. Development graphs - proof management for structured specifications. *J. Log. Algebr. Program.*, 67(1-2):114–145, 2006.

[MCG04] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion - a taxonomy of model transformations. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2004.

[Men10] Tom Mens. Model transformation: A survey of the state-of-the-art. In Sebastien Gerard, Jean-Philippe Babau, and Joel Champeau, editors, *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Wiley - ISTE, 2010.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[MGB05] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal refactoring for UML class diagrams. In *19th brazilian Symposium on Software Engineering (SBES)*, pages 152–167, 2005.

[MGDT05] Till Mossakowski, Joseph Goguen, Răzvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Béziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser, 2005.

[MHST03] Till Mossakowski, Anne Elisabeth Haxthausen, Donald Sannella, and Andrzej Tarlecki. Casl - the common algebraic specification language: Semantics and proof theory. *Computers and Artificial Intelligence*, 22(3-4):285–321, 2003.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, Hets. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007.

[Mos05] Till Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universitaet Bremen, 2005. Habilitation thesis.

[Mos13a] Till Mossakowski. HetCASL - heterogeneous specification. language summary. http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HetCASL/HetCASL-Summary.pdf, apr 2014.

[Mos13b] Till Mossakowski. Hets - the heterogeneous tool set website. http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm, apr 2014.

[Mos13c] Till Mossakowski. ModalCASL - specification with multi-modal logics language summary. http://www.informatik.uni-bremen.de/~till/papers/Modal-Summary.pdfv, apr 2014.

[MSJ+10] Tim Molderez, Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. A platform for experimenting with language constructs for modularizing crosscutting concerns. In *Proceedings of the 3rd International Workshop on Academic Software Development Tools and Techniques (WASDeTT)*, 2010.

[MSRR06] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006.

[NK08a] Anantha Narayanan and Gabor Karsai. Specifying the correctness properties of model transformations. In *Proceedings of the third international workshop on Graph and model transformations (GRaMoT'08)*, pages 45–52. ACM, 2008.

[NK08b] Anantha Narayanan and Gabor Karsai. Towards verifying model transformations. *Electr. Notes Theor. Comput. Sci.*, 211:191–200, 2008.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[OMG03a] OMG. MDA guide version 1.0.1. Technical report, Object Management Group, 2003.

[OMG03b] OMG. Meta Object Facility (MOF) 2.0 Core Specification. Specification Version 2.0, Object Management Group, 2003.

[OMG05] OMG. Unified Modeling Language: Superstructure. Specification Version 2.0, Object Management Group, August 2005.

[OMG09] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation. Final Adopted Specification Version 1.1, Object Management Group, 2009.

[OMG10] OMG. Object Constraint Language. Formal Specification Version 2.2, Object Management Group, 2010.

[OMG11] OMG. OMG MOF 2 XMI Mapping Specification. Specification Version 2.4.1, Object Management Group, 2011.

[OW09] Fernando Orejas and Martin Wirsing. On the specification and verification of model transformations. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *Lecture Notes in Computer Science*, pages 140–161. Springer, 2009.

[PGE97] Julia Padberg and Magdalena Gajewsky and Claudia Ermel. Refinement versus verification: Compatibility of net-invariants and stepwise development of high-level petri nets. Technical Report 97-22, Technical University Berlin, 1997.

[PCG11] Elena Planas, Jordi Cabot, and Cristina Gómez. Two basic correctness properties for ATL transformations: Executability and coverage. In *3rd International Workshop on Model Transformation with ATL*, July 2011.

[PG08] Claudia Pons and Diego García. A lightweight approach for the semantic validation of model refinements. *Electr. Notes Theor. Comput. Sci.*, 220(1):43–61, 2008.

[Poe08] Iman Poernomo. Proofs-as-model-transformations. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[PT10] Iman Poernomo and Jeffrey Terrell. Correct-by-construction model transformations from partially ordered specifications in Coq. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2010.

[RDV09] José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific models using Maude. *Simulation*, 85(11-12):778–792, 2009.

[RLK+08] Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan. Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In Ehrig et al. [EHRT08], pages 242–256.

[RWLN89] Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The nature of modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[Sch06] Douglas Schmidt. Guest editor's introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Sch10] Bernhard Schätz. Verification of model transformations. *ECEASST*, 29, 2010.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Ehrig et al. [EHRT08], pages 411–425.

[SMR11] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Formal verification of QVT transformations for code generation. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 533–547. Springer, 2011.

[ST12] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs on theoretical computer science. Springer, 2012.

[Ste13] Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Software and System Modeling*, 12(1):175–199, 2013.

[SZ09] Lijun Shan and Hong Zhu. Semantics of metamodels in UML. In Wei-Ngan Chin and Shengchao Qin, editors, *TASE*, pages 55–62. IEEE Computer Society, 2009.

[Tae03] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[TBHW99] Françoise Tort, Michel Bidoit, Rolf Hennicker, and Martin Wirsing. Correct realization of interface constraints with OCL. In Robert B. France and Bernhard Rumpe, editors, *UML*, volume 1723 of *Lecture Notes in Computer Science*, pages 399–415. Springer, 1999.

[VJBB13] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. Typing artifacts in megamodeling. *Software and System Modeling*, 12(1):105–119, 2013.

[VP03] Dániel Varró and András Pataricza. Automated formal verification of model transformations. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *Critical Systems Development in UML (CSDUML 2003), Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.

[VR11] Andreza Vieira and Franklin Ramalho. A static analyzer for model transformations. In *3rd International Workshop on Model Transformation with ATL*, Zurich, Switzerland, July 2011.

[VVGE+06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination analysis of model transformations by petri nets. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2006.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. Spass version 2.0. In Andrei Voronkov, editor, *Automated Deduction CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer Berlin Heidelberg, 2002.

[WKC06] Junhua Wang, Soon-Kyeong Kim, and David A. Carrington. Verifying metamodel coverage of model transformations. In *ASWEC*, pages 270–282. IEEE Computer Society, 2006.

[WKK+09] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Right or wrong? - verification of model transformations using colored petri nets. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[WMP13] Hao Wu, Rosemary Monahan and James Power. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *Proceedings of the 7th Intl. Symp. on Theoretical Aspects of Software Engineering (TASE)*, pages 175–182, 2013.

[WSTL10] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010.

[ZCP13] Faiez Zalila, Xavier Crégut, and Marc Pantel. A transformation-driven approach to automate feedback verification results. In Alfredo Cuzzocrea and Sofian Maabout, editors, *MEDI*, volume 8216 of *Lecture Notes in Computer Science*, pages 266–277. Springer, 2013.

[Zhu12] Hong Zhu. An institution theory of formal meta-modelling in graphically extended BNF. *Frontiers of Computer Science*, 6(1):40–56, 2012.