Assignment of master's thesis

Title:	Implementation of the transformation of an OntoUML model in OpenPonk into its realization in a relational database
Student:	Bc. Jakub Jabůrek
Supervisor:	Ing. Zdeněk Rybola, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The goal of the thesis is the implementation of the transformation of a conceptual model in the OntoUML notation created in the OpenPonk application into its corresponding realization in a relational database while preserving as many integrity constraints derived from the original OntoUML model as possible.

During the realization, familiarize yourself with the latest version of the OntoUML notation and the underlying UFO ontology, and the transformation suggested by the thesis supervisor in his dissertation thesis [1]. Together with the supervisor, revise the suggested transformation rules in accordance with the latest version of OntoUML.
Design and implement the transformation of an OntoUML model created in OpenPonk into SQL scripts for creating the relational database structures realizing the original conceptual model together with its integrity constraints in an Oracle database. The transformation should be available to run in the OpenPonk application. It should provide both the SQL scripts for the target database and the intermediate conceptual UML model and the relational database model with appropriate constraints. These intermediate constraints do not have to be integrated into OpenPonk, for now.
Document and test the implementation according to the best practices of software engineering. For the tests, use the model example from [1], eventually other available models.

Electronically approved by Ing. Michal Valenta, Ph.D. on 22 December 2023 in Prague.

Master's thesis

Implementation of the Transformation of an OntoUML Model in OpenPonk into Its Realization in a Relational Database

Jakub Jabůrek

Department of Software Engineering
Supervisor: Ing. Zdeněk Rybola, Ph.D.

May 9, 2024

Acknowledgements

First, I would like to thank the supervisor of my thesis, Ing. Zdeněk Rybola, Ph.D., for his valuable time and guidance, and for his research, which my thesis builds upon. I also thank Ing. Jan Blizničenko for his advice about Pharo programming and OpenPonk development.

Last but not least, I'd like to express my gratitude to my entire family and friends, who have supported and encouraged me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 9, 2024.

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Jakub Jabůrek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

JABŮREK Jakub. *Implementation of the Transformation of an OntoUML Model in OpenPonk into Its Realization in a Relational Database.* Master's thesis. Prague: Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

This thesis deals with the implementation of an automated transformation of an OntoUML conceptual model to SQL. The implementation is developed as an extension of the OpenPonk modeling platform and is written in the Pharo programming language. An approach to the transformation that preserves constraints implied by the OntoUML model is studied, then a framework for the implementation of model transformations is developed, and the transformation is implemented and integrated with OpenPonk.

Keywords OntoUML, UML, relational database, SQL, transformation, constraints, OpenPonk, Pharo

Abstrakt

Práce se zabývá implementací automatické transformace konceptuálního OntoUML modelu do SQL. Implementace je vytvořena jako rozšíření modelovacího nástroje OpenPonk, a programována v jazyce Pharo. Je popsán princip transformace, který zachovává omezení implikovaná OntoUML modelem. Následně je vytvořeno řešení pro implementaci transformací modelů, a navržená transformace je implementována a integrována do OpenPonku.

Klíčová slova OntoUML, UML, relační databáze, SQL, transformace, omezení, OpenPonk, Pharo

Contents

1 Introduction **1**
1.1 Goals of This Thesis . . . 2
1.2 Structure of the Thesis . . . 2

2 State of the Art **3**
2.1 Conceptual Modeling . . . 3
2.1.1 Unified Modeling Language . . . 4
2.1.2 Unified Foundational Ontology . . . 6
2.1.3 OntoUML . . . 6
2.2 Relational Database Management Systems . . . 9
2.2.1 Relational Model . . . 10
2.2.2 Structured Query Language . . . 11
2.2.3 Oracle Database . . . 11
2.3 Data Modeling . . . 12
2.3.1 Entity–Relationship Model . . . 12
2.3.2 Data Modeling Profile of the Unified Modeling Language . . . 13
2.4 Object Constraint Language . . . 13
2.5 Realization of an OntoUML Model in SQL . . . 14
2.5.1 Transformation of an OntoUML Model to UML . . . 14
2.5.2 Transformation of a UML Model to a Relational Model . . . 18
2.5.3 Realization of a Relational Model in Oracle Database . . . 22

3 Analysis **29**
3.1 OpenPonk . . . 29
3.1.1 Data Model . . . 30
3.1.2 Transformation of an OntoUML Model to UML . . . 31
3.2 Pharo . . . 33
3.2.1 Code Organization . . . 33
3.2.2 Source Code Versioning . . . 34
3.2.3 Package Management . . . 34
3.2.4 Graphical User Interface . . . 35
3.3 Transformation Revision for the Latest OntoUML Version . . . 35
3.3.1 Changes in OntoUML . . . 35
3.3.2 Enumerations . . . 36
3.4 Software Requirements . . . 36
3.4.1 Functional Requirements . . . 36

3.4.2 Non-Functional Requirements . . . 37

4 Design **39**
4.1 Framework for Transformations . . . 39
4.2 Transformation Engine . . . 40
4.3 Moco Library . . . 42
4.4 Data Model . . . 42
4.4.1 UML Data Model . . . 42
4.4.2 OntoUML Data Model . . . 43
4.4.3 Relational Data Model . . . 44
4.4.4 Oracle SQL Data Model . . . 46
4.4.5 OCL Data Model . . . 46
4.5 Transformation Rounds . . . 47
4.6 User Interface . . . 49
4.6.1 Toolbar Menu . . . 49
4.6.2 Question Dialog . . . 49
4.6.3 Results Window . . . 50
4.6.4 Settings . . . 50

5 Implementation **51**
5.1 Transformation Engine . . . 51
5.1.1 Adding Elements and Linking Source with Output Elements . . . 51
5.1.2 Selecting Elements and Managing Metadata . . . 52
5.1.3 Constructing Models . . . 53
5.1.4 Transformation Options . . . 53
5.1.5 Transformation Execution . . . 53
5.1.6 Naming Utilities . . . 54
5.2 Transformation Rules . . . 55
5.2.1 OpenPonk OntoUML to Moco OntoUML . . . 55
5.2.2 OntoUML to UML . . . 57
5.2.3 Preprocessing UML for the Relational Model . . . 62
5.2.4 UML to a Relational Model . . . 63
5.2.5 Relational Model to Oracle SQL . . . 66
5.2.6 Moco UML to OpenPonk UML . . . 67
5.2.7 Moco RDB to OpenPonk UML . . . 68
5.3 User Interface . . . 68
5.3.1 Transformation Transcript . . . 68
5.3.2 Toolbar Item . . . 69
5.3.3 Transformation Command . . . 69
5.3.4 Moco Playground . . . 70
5.4 Code Repository . . . 70
5.5 OpenPonk Build . . . 71
5.5.1 Project Saving Workaround . . . 71

6 Evaluation **73**
6.1 Functional Testing . . . 73
6.2 Differences in Generated Names . . . 74
6.2.1 OntoUML to UML . . . 74
6.2.2 UML to a Relational Model . . . 75
6.3 Transformation of the Example Model . . . 75
6.3.1 OntoUML to UML . . . 76
6.3.2 UML to a Relational Model . . . 76

6.3.3 Realization of Constraints and Resulting SQL Model 77
6.3.4 Summary 78

7 Conclusion **79**
7.1 Summary 80
7.2 Contributions of this Thesis 80

Bibliography **81**

Attachments **85**

A Example Model **87**
A.1 OntoUML Model 88
A.2 UML Model 90
A.3 Relational Model 92
A.4 Realization in SQL 94

B Transformation Rules **95**
B.1 OpenPonk OntoUML to Moco OntoUML 95
B.2 OntoUML to UML 97
B.3 Preprocessing UML for the Relational Model 98
B.4 UML to a Relational Model 98
B.5 Relational Model to Oracle SQL 98
B.6 Moco UML to OpenPonk UML 99
B.7 Moco RDB to OpenPonk UML 99

List of Figures

2.1 Instances of Class, Node and Actor UML classifiers 4
2.2 Annotated example of a UML class element 5
2.3 Annotated example of a UML relationship 5
2.4 UML generalization 6
2.5 Entity–Relationship model 13
2.6 UML model using the Data Modeling Profile 13
2.7 Transformation of an OntoUML Kind and SubKind (left) to UML (right) . . 15
2.8 Optimization (right) of a generalization set (left) 15
2.9 Transformation of an OntoUML Role (top) to UML (bottom) 16
2.10 Transformation of an OntoUML Phase (left) to UML (right) using exclusive phase associations 17
2.11 OntoUML Phase transformed to UML via an abstract phase 17
2.12 Transformation of an OntoUML Mode (left) to UML (right) 18
2.13 Transformation of a UML class (left) to a table (right) 18
2.14 Transformation of a one–to–many association (top) to a foreign key (bottom) 19
2.15 Transformation of a UML generalization set (left) to referencing tables (right) 20

3.1 OpenPonk user interface 30
3.2 Inheritance hierarchy of an `OPUMLClass` class in OpenPonk 30
3.3 Object diagram of the composition of an element that represents an OntoUML Kind 31
3.4 Sequence diagram of the transformation of an OntoUML class to UML in OpenPonk 31
3.5 Pharo user interface 35

4.1 Class diagram of UML data model 43
4.2 Class diagram of an excerpt from the OntoUML data model 44
4.3 Class diagram of an excerpt from the relational data model 45
4.4 Class diagram of the Oracle SQL data model 46
4.5 Class diagram of an excerpt from the OCL data model 47
4.6 Wireframe of the OpenPonk window toolbar 49
4.7 Wireframe of the transformation option choice dialog 49
4.8 Wireframe of the transformation transcript window 50
4.9 Settings Browser in Pharo 50

6.1 OntoUML model with a Role (left), the proposed optimization (middle), and the result of the implemented optimization (right) 76

6.2 Manually applied generalization set optimization (left) and result of the implemented transformation (right) 77

A.1 Example OntoUML model 89
A.2 Transformed UML model 91
A.3 Transformed relational model 93

List of Listings

2.1 SQL SELECT statement . . . 11
2.2 OCL postcondition asserting the value attribute is incremented by one . . . 14
2.3 OCL constraint restricting the values of a discriminator attribute . . . 15
2.4 OCL invariant realizing exclusive phase associations . . . 17
2.5 OCL invariant realizing phase attribute . . . 17
2.6 OCL mandatory multiplicity constraint . . . 19
2.7 OCL special multiplicity constraint . . . 20
2.8 OCL constraint for a disjoint and incomplete generalization set realized by referencing tables . . . 21
2.9 OCL postcondition realizing the immutability of target table . . . 21
2.10 OCL postcondition preventing the deletion of the source side of an association 21
2.11 SQL CREATE TABLE statement . . . 22
2.12 SQL ALTER TABLE statements for primary key and unique constraint . . . 22
2.13 SQL ALTER TABLE statement realizing a foreign key . . . 23
2.14 SQL trigger for the superclass table realizing a generalization set constraint . 23
2.15 SQL subclass INSERT trigger realizing a generalization set constraint . . . 23
2.16 SQL trigger guarding the UPDATE operation for a generalization set constraint 24
2.17 SQL trigger guarding the DELETE operation for a generalization set constraint 24
2.18 SQL trigger definition realizing a mandatory multiplicity constraint at the constrained table side . . . 25
2.19 SQL trigger definition realizing a mandatory multiplicity constraint at the related table side . . . 25
2.20 SQL trigger definition realizing a special multiplicity constraint at the constrained table side . . . 25
2.21 SQL trigger definition realizing a special multiplicity constraint at the related table side . . . 26
2.22 SQL trigger definition realizing an exclusivity constraint at the constrained table side . . . 26
2.23 SQL trigger definition for an exclusivity constraint guarding the insertion of records to a related table . . . 26
2.24 SQL trigger definition for an exclusivity constraint guarding the change of records in a related table . . . 27
2.25 SQL CHECK constraint definition . . . 27
2.26 SQL trigger definition realizing an immutable column . . . 27
2.27 SQL trigger definition realizing immutable source side of an association . . . 28

3.1 OCL enumeration constraint . . . 36

4.1 Algorithm of the Transformation Engine 41

5.1 Instantiating the transformation engine and executing a transformation . . . 54
5.2 Usage of transformation rule repository 54
5.3 Displaying transformation transcript 68
5.4 Prefilled code in Moco Playground 70
5.5 Installation of Moco using Metacello 71

List of Tables

2.1 Tabulated example relation 10

4.1 List of data types in the relational data model 45
4.2 List of transformation rounds 48

5.1 Methods for converting between naming conventions 54
5.2 OpenPonk OntoUML to Moco OntoUML class transformation rules 56
5.3 OpenPonk OntoUML to Moco OntoUML association transformation rules . . 56
5.4 OpenPonk OntoUML to Moco OntoUML part–whole relationship transformation rules 57
5.5 OntoUML to UML class transformation rules 58
5.6 OntoUML to UML mixin transformation rules 58
5.7 UML to RDB data type mapping 64
5.8 Column data type mapping for Oracle Database 66
5.9 RDB to SQL transformation rules for OCL constraints 67
5.10 Package list of Moco 70

6.1 Functional test suites 74
6.2 Difference in the number of generated elements 77

A.1 List of elements in the example OntoUML model 88
A.2 List of elements in the example UML model 90
A.3 List of elements in the example RDB model 92
A.4 List of generated SQL statements 94

B.1 List of OpenPonk OntoUML to Moco OntoUML transformation rules 95
B.2 List of OntoUML to UML transformation rules 97
B.3 List of UML to UML for RDB transformation rules 98
B.4 List of UML to RDB transformation rules 98
B.5 List of RDB to SQL transformation rules 98
B.6 List of Moco UML to OpenPonk UML transformation rules 99
B.7 List of Moco RDB to OpenPonk UML transformation rules 99

Abbreviations

API Application Programming Interface

DBMS Database Management System

DDL Data Definition Language

DML Data Manipulation Language

E/R Entity–Relationship Model

GUI Graphical User Interface

OCL Object Constraint Language

OOP Object–Oriented Programming

RDB Relational Database

RDBMS Relational Database Management System

SQL Structured Query Language

UFO Unified Foundational Ontology

UML Unified Modeling Language

Chapter **1**

Introduction

In software engineering, an approach known as *model-driven development* proposes that software systems be developed by constructing a model of the problem domain (i.e. the area the software intends to address), and then transforming the model to the final realization of the software. A model provides abstraction — the engineer can focus on solving the problem domain while not being concerned with the details of the final implementation. [1, p. 1–2]

Several modeling standards and notations exist, providing varying levels of abstraction and precision [2]. The higher the level of abstraction is, the bigger the gap between the model and its realization may be. Moreover, the model may be built without even considering a particular target platform, or multiple realizations of the same model may be derived. Yet, when an appropriate approach to the construction of the model is employed, even an abstract model can be precise.

OntoUML is a conceptual modeling language — it describes the structure of the domain, its elements, and relationships. Therefore, the gap between the model and its potential realization can be quite high. It is based on the Unified Modeling Language (UML) and the Unified Foundational Ontology (UFO), and is *ontologically well-founded*: based on cognitive science, philosophy and mathematical theory of sets and relations [3, p. 3]. As a result, OntoUML models can be precise and accurate.

In *computer-aided software engineering*, computer modeling tools are used to build and work with the model. When a suitable modeling tool and a quality model are combined, the transformation of the model to its implementation can be automated [4, p. 2–5].

The target platform of a software model may be, for example, the source code that implements the software system in a particular programming language, or the definition of the data storage for the system. A common approach to data management — especially in complex systems — is the usage of a database.

OpenPonk is a modeling tool that provides support for OntoUML models [5]. In *Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases*, a possible approach to the realization of an OntoUML model in a database is presented [6].

In this thesis, we adopt the transformation approach from [6], and implement it in OpenPonk — creating a tool capable of automatically transforming any OntoUML model to its realization in a relational database.

1.1 Goals of This Thesis

This thesis aims to implement the transformation of an OntoUML model to its realization in a relational database proposed in [6] as an extension of the OpenPonk modeling tool. The implementation intends to preserve all integrity constraints defined in the referenced approach, and to realize all OntoUML to UML transformation alternatives.

Ultimately, the objective is to create a software product capable of automatically transforming OntoUML models to SQL.

1.2 Structure of the Thesis

The thesis is structured in a way that follows the development process of the implemented software.

- In *Chapter 1: Introduction*, the motivation and goals of the thesis are established.
- In *Chapter 2: State of the Art*, the background for this thesis is introduced. Conceptual modeling, UML and OntoUML are discussed, as well as relational databases and the specification of model constraints. Finally, the approach to the transformation of an OntoUML model to a relational database from [6] is explained.
- In *Chapter 3: Analysis*, an overview of the technologies used during the implementation is provided. OpenPonk and the Pharo programming language are discussed, and software requirements for the implementation of the transformation are developed. Also, the proposed transformation approach is revised to work with the latest version of OntoUML.
- In *Chapter 4: Design*, the approach to the implementation is established — including the basis of the algorithms that realize the transformation, the data model, and the structure of the source of the implementation. Also, a design of the user interface is provided.
- In *Chapter 5: Implementation*, the details of the implementation and its integration with OpenPonk are described.
- In *Chapter 6: Evaluation*, the implementation is tested, and verified against an example OntoUML model.
- In *Chapter 7: Conclusion*, the results of the thesis are summarized.

Chapter **2**

State of the Art

In this chapter, the current state of the art related to conceptual and data modeling, database management systems, and model constraint specification is described. The chapter is structured as follows.

- In section 2.1, the Unified Modeling Language, the UML Class diagram, the Unified Foundational Ontology, and OntoUML are described.
- In section 2.2, the relational model, the SQL language, and Oracle Database are discussed.
- In section 2.3, the entity–relationship data model, and the Data Modeling Profile for UML are explained.
- In section 2.4, the OCL language for specifying model constraints is introduced.
- Finally, in section 2.5, the transformation of an OntoUML model to SQL proposed in [6], which is implemented in this thesis, is summarized.

2.1 Conceptual Modeling

Models in the context of software engineering aid the analysis of a particular application domain, construction of a software system, precise communication about the software system and the domain, and can be used to assess and further improve the implemented system. [7, p. 543]

The software engineering industry utilizes various kinds of models, depending on the use case for the model. Conceptual models in particular are used to describe the application domain in an abstract way while utilizing common concepts shared by all involved stakeholders. In other words, a conceptual model reflects the way people think about the domain. [7, p. 544–545]

To further distinguish between different types of software models, we can use the following distinction that Unified Modeling Language makes [8, p. 13].

Structural Models. Their main focus is on the static features of the system. They describe entities, their properties, and the relationships between them.

Behavioral Models. They are used to model dynamic features of the system, such as interactions between components or processes in the system.

The topic of this thesis is the transformation of an OntoUML conceptual model to a relational database; as OntoUML models software structure and also the relational model is a structural model, we focus on structural models.

Typical use cases in software engineering require all stakeholders to agree on the meaning of the model, as ambiguities could prevent efficient work with the model or even cause incorrect implementations to be developed. Therefore, it is common to use a standardized form of modeling, where the meaning of model features is precisely defined and a particular graphical notation is prescribed. In the case of structural software models, the most widely used and recognized standard is the Unified Modeling Language [9, p. 7].

2.1.1 Unified Modeling Language

The Unified Modeling Language (UML) is a standardized language for the specification, visualization, construction, and documentation of software systems [10, p. 13], currently developed at the Object Management Group. It provides standard semantics and notation for many types of models used in software engineering.

2.1.1.1 UML Metamodel

Elements in a UML model may represent a wide range of different concepts. For this reason, the language includes a concept of *classifiers*, which are part of the UML metamodel. They define structural and behavioral characteristics (called *features*), and visual style (or *shape*) of model elements that are instances of these classifiers. [10, p. 119–120]

Figure 2.1 illustrates instances of three different UML classifiers: Class, Node and Actor.

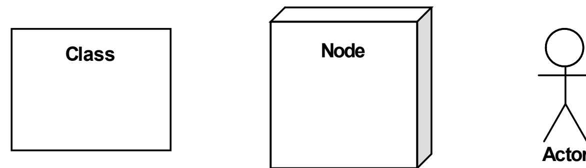

Figure 2.1: Instances of Class, Node and Actor UML classifiers

To support specialized modeling use cases, UML users may define their own *stereotypes* [10, p. 80]. They can bring additional semantics or change the shape of the element they are applied to.

A set of stereotypes (plus other types of extensions) can be packaged to form a *profile* [10, p. 83–84]. A UML profile can be reused (*applied*) in other models. By providing this standardized extension mechanism, UML can be customized while maintaining compatibility with conformant general modeling tools [8, p. 3].

2.1.1.2 UML Class Diagram

A Class diagram in UML refers to a model that uses classifiers and stereotypes that describe the structure of an object-oriented software system: classes, interfaces, generalizations, and associations. This subsection provides a non-exhaustive overview of the notation used in class diagrams.

Classes

Elements drawn as simple rectangles represent the *classes* in the software system. The name of the class is shown at the top; in case the class is abstract, its name will be in

italics. The class diagram models classes (object templates), not individual class instances.

Class elements can contain *compartments*, separated from the name or other compartments by a line. The two most used compartment types are *attributes* and *operations* compartments. Attributes compartment contains the list of attributes of the class and their data types. Operations compartment lists the methods of the class, their parameters, and return type. Additionally, features in these compartments also indicate their visibility using symbols in front of their name and are underlined when they are static.

An annotated example of a class element is provided in figure 2.2.

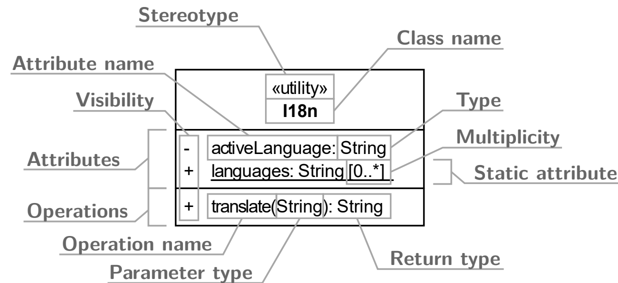

Figure 2.2: Annotated example of a UML class element

Relationships

Classes within the system usually do not operate in isolation. To show that there exists a *relationship* between two classes, a line is drawn connecting the both of them. UML also allows relationships between more than two elements, which are drawn as diamonds with lines connecting them to all participating elements.

A name that describes the relationship may be written above the line, the individual relationship sides can be named as well. Each side can define its *multiplicity*, that is, the minimal and maximal amount of instances of the individual that are in a relationship with the other side.

An annotated example of a relationship between two classes is provided in figure 2.3.

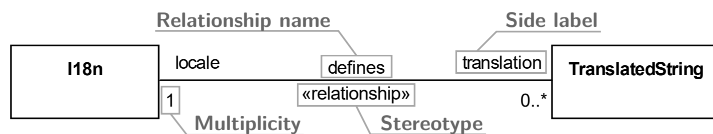

Figure 2.3: Annotated example of a UML relationship

Additionally, the relationship may be specified as an *aggregation*, in which the instance on one side can be considered to be a part of the instance on the other side. The meaning of aggregation in UML is purely conceptual. [10, p. 67–68]

Alternatively, the relationship may be specified as a *composition*. In this case, the part may be included in at most one composite object at a time, and upon destruction, the composite object has a responsibility to destroy all its owned objects as well. [8, p. 110]

In a class diagram, an aggregation is visualized by an empty diamond at the side of the container, a composition by a filled diamond at the side of the owner.

Generalizations

To indicate a *generalization* relationship, a line between the general and specific classifiers is drawn with a triangle at the side of the more general individual. As per the UML

standard, the specific classifier then inherits certain features of the general classifier [8, p. 98]. A classifier may be the general for multiple specific classifiers, as well as a specific classifier may have multiple general classifiers.

In order to better define classification hierarchies, multiple generalizations with the same general classifier may be grouped into a *generalization set*. A generalization set may be defined as either *complete* or *incomplete*: in a complete set, every instance of the general classifier must also be an instance of at least one of the specific classifiers. On the other hand, an incomplete generalization set permits instances of the general classifier that are not an instance of any of the specific ones. [8, p. 117-118]

Additionally, a generalization set may be defined as either *disjoint* or *overlapping*: in an overlapping set, an instance of the general classifier may be an instance of multiple specific classifiers, whereas in a disjoint set, the instance can be an instance of at most one of the specific classifiers. [8, p. 118]

Since UML 2.5, generalization sets are by default incomplete and overlapping, in prior versions, they defaulted to incomplete and disjoint. [8, p. 119, 11, p. 79]

An example of a generalization with a generalization set is provided in figure 2.4.

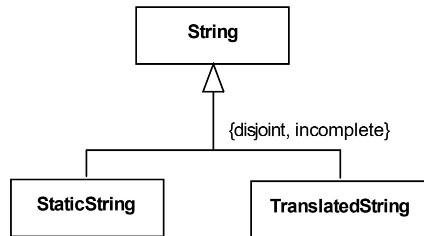

Figure 2.4: UML generalization

2.1.2 Unified Foundational Ontology

The Unified Foundational Ontology (UFO) is a universal ontology that combines theories from formal ontology in philosophy, cognitive science, linguistics and philosophical logic [12, p. 1], and provides foundations for domain analysis in conceptual modeling [12, p. 4].

The ontology is divided into three layers [3, p. 3–4].

UFO–A. Deals with aspects of structural conceptual modeling — types, properties and relationships.

UFO–B. Deals with behavioral aspects, such as events or processes.

UFO–C. Builds on UFO–A and UFO–B and studies social aspects of entities.

This thesis focuses on structural conceptual models, which are covered by UFO–A. The theory is summarized in section 2.1.3, where its realization in a UML model is discussed.

2.1.3 OntoUML

OntoUML was developed as an ontologically well-founded version of UML, realizing the UFO–A theory [3, p. 4]. OntoUML can be considered a profile of UML, as it defines class and relationship stereotypes, which correspond to types introduced in UFO.

An OntoUML model defines *types*: abstract bundles of characteristics shared among their concrete instances; this corresponds to classes in standard UML. Types are instantiated into *individuals*. [13, p. 3]. A single individual may instantiate multiple types.

Additionally, the language operates with the ontological notion of *identity*, which is used to distinguish individuals from one another. Each individual must have exactly one source of identity (in other words, exactly one type of all types the individual might instantiate can be an identity provider). Types that have an identity (whether inherited or provided by themselves) are classified as *sortals*, types that have no identity are classified as *non-sortals*. Non-sortals are used to characterize individuals that follow different identity principles. [13, p. 4–5]

Finally, the set of types that a particular individual instantiates can change during the course of its lifetime. This is a notable difference between OntoUML and regular UML or common OOP languages. Types that the individual must instantiate in all situations are classified as *rigid*, types that may or may not be instantiated by the individual at any given time are classified as *anti-rigid* [13, p. 4]. As anti-rigid types are optional to the existence of an individual, they cannot provide identity.

2.1.3.1 Class stereotypes

Each type in an OntoUML model must have a stereotype. Stereotypes declare the rigidity and identity principle of the type, and provide additional ontological information and constraints to the type. This subsection lists all class stereotypes that OntoUML provides.

Rigid Sortals

Kind. A standalone rigid construct that provides identity. [14, p. 5]

SubKind. Represents a rigid specialization of identity providers. May specialize all rigid sortals and aspects that provide identity. Does not provide identity itself. [14, p. 5]

Quantity. In general, represents a maximally topologically connected amount of matter. For example, water is a Quantity, as it can be virtually infinitely divided into multiple smaller instances. A Quantity may contain other Quantities via a Part–Whole relationship with «SubQuantityOf» stereotype. Provides identity. [15, p. 33]

Relator. A rigid construct that provides identity, but is existentially dependent on other individuals: it must always be connected directly to at least one Mediation relation. It represents the *truth-maker* of a Material relation, (in other words, it represents the properties of that relation). It can be specialized by SubKinds, Phases, and Roles, and generalized by Categories and Mixins. [15, p. 36–37]

Collective. Represents a whole with a homogeneous internal structure: all parts are perceived equally, and there are no distinctions between roles and responsibilities of the individual parts. Members are connected with the Collective via a relationship with MemberOf stereotype. Provides identity. [15, p. 29]

Anti-Rigid Sortals

Role. Represents an anti-rigid existentially dependent specialization of an identity provider. Must be always connected to at least one mandatory relation with Mediation stereotype. [15, p. 24]

Phase. Specializations whose instantiation depends on the intrinsic properties of the individual may be represented as Phases. A Phase must always be part of a *phase partition*: a generalization set disjoint and complete. [15, p. 20]

Rigid Non-Sortals

Category. An abstract mixin that aggregates common properties of individuals with different identity principles. [15, p. 43]

Anti-Rigid Non-Sortals

RoleMixin. Represents a mixin specifically for Roles that aggregate instances with different identity principles. [15, p. 48]

PhaseMixin. Similarly to a RoleMixin, a PhaseMixin is a supertype for Phases with different identity principles. [15, p. 47]

Semi-Rigid Non-Sortals

A *semi-rigid* type acts as a rigid type for its rigid subtypes, and as an anti-rigid type for its anti-rigid subtypes.

Mixin. Represents a generic mixin that aggregates common properties of its subtypes, suitable for generalization of rigid and anti-rigid individuals at the same time. [15, p. 51]

Aspects

Both *aspect* types are rigid and provide identity.

Quality. When the intrinsic property has a structured value (e.g. color or position), it can be represented by a Quality. A Quality is existentially dependent on its *bearer*, and must be connected to it via a Characterization relation. [15, p. 60]

Mode. They are similar to Qualities but represent properties that have no structured value. A Mode must be connected to a bearer via a Characterization relation. [15, p. 56]

2.1.3.2 Relationship stereotypes

Similarly to types, relations are also categorized by OntoUML based on their ontological meaning using various stereotypes.

Associations

Formal. Represents a relation that holds between two or more individuals directly and can be reduced to a comparison of the characterizations of the related individuals (e.g. cheaper–than). [15, p. 63–64]

Material. Constructed as derived relations from Relators and Mediation relations. The Relators provide structure to the Material relation and they mediate the *relata* of the relation. [15, p. 66]

Mediation. Forms the existential dependence relation between a Relator and the individuals it mediates. [15, p. 68]

Characterization. Represents the relation between a bearer and a Quality or Mode (the *feature*). [15, p. 69]

Derivation. When Material relations are derived from Relators, the Derivation relation is used to connect the Material relation with the corresponding Relator [15, p. 70]. This is a special kind of relation, where one side of the relation is a relation, not a class.

Part–Whole Aggregations

Parthood relations are significant from a cognitive perspective, and for this reason, they are fundamental in OntoUML as well [16, p. 141]. OntoUML provides deeper semantics for part–whole relations than standard UML via the following means:

Shareability. Refines the concept of aggregation from UML: a shareable part may belong to multiple wholes at a single time, an exclusive part may belong to at most one whole at any given time.

Essential Parthood. Specifies that the whole is existentially dependent on the part, and the instance of the part cannot change during the lifetime of the whole. [17, p. 6–7]

Inseparability. The same principles as in essentiality apply, except from the part side: the part is existentially dependent on the whole, and the whole instance cannot change. [17, p. 9]

The visual representation of shareability in an OntoUML diagram overrides UML aggregation: a shareable part has an empty diamond on the opposite side, an exclusive part a filled diamond.

Relations with an essential part have an *essential* constraint drawn next to the relationship line, similarly, relations with an inseparable part carry an *inseparable* constraint. The essential constraint is by definition applicable only to relations with rigid wholes; anti-rigid types have an equivalent *immutable part* constraint. Similarly, only relations with a rigid part can be inseparable, those with an anti-rigid part must be constrained with *immutable whole*.

Part–Whole relations can be expressed by applying the following stereotypes:

ComponentOf. Represents a relation between a part and the whole of a functional complex. Transitivity of the relation is not prescribed by OntoUML, it depends on the context. [15, p. 76]

Containment. Used to associate a Quantity with its container [15, p. 77]. As the Quantity represents a maximally topologically connected amount of matter within the container, multiplicities of both sides of the relation must be exactly one.

MemberOf. Represents a relation between a Collective and its members. A Collective may have another Collective as its member, but the MemberOf relation is not transitive. [18, p. 149–151]

SubCollectionOf. Allows nesting of Collectives: both parts of the relation must be Collective individuals and, as opposed to the MemberOf relation, SubCollectionOf is transitive. [18, p. 149–150]

SubQuantityOf. Represents further division of a Quantity (e.g. oxygen is a part of air). All parts of a Quantity are essential, both sides of the SubQuantityOf relation must have multiplicities of exactly one. This relation is transitive. [19, p. 13]

2.2 Relational Database Management Systems

Lots of software systems have the requirement to process and store data, sometimes of complex structure and large volume. Instead of designing and implementing custom storage solutions, these systems can use a Database Management System (DBMS).

A DBMS is a dedicated software system that should possess certain capabilities [20, p. 3–4], as follows.

Large Datasets. A DBMS should be capable of efficiently storing and processing even large amounts of data, given adequate computing power and good database design.

Persistence. The managed data should be stored until the software system requests them to be deleted. They are not limited to e.g. a single execution of a procedure within the software system, and should persist between system restarts.

Sharing. The DBMS should support concurrent access to the managed data, that is, multiple clients should be able to query and manipulate the data without the possibility of corrupting it.

Reliability. The DBMS should handle software and hardware errors, and ensure the managed data remains consistent after events such as a system crash or a power outage.

Efficiency. Clients should be able to query and manipulate managed data quickly. Usually, it is the responsibility of the database designer to construct efficient queries, and to use features provided by the DBMS to build an efficient database structure.

Various kinds of database management systems exist: relational, object-oriented, document, and graph are some examples. Each kind may be suitable for different use-cases, and presents a different model for the managed data. In this thesis, we focus on Relational Database Management Systems (RDBMS).

2.2.1 Relational Model

The relational model for databases was originally proposed by Edgar Codd in 1970: Given sets $D_1, D_2, \dots, D_n$, a *relation* is a subset of the Cartesian product $D_1 \times D_2 \times \dots \times D_n$. In other words, it is a set of n-tuples $(a_1, a_2, \dots, a_n)$ where $a_i \in D_i$. The sets $D_1, \dots, D_n$ are referred to as the *domains* of the relation. [21]

Relations are often tabulated, and labels are added to each domain to convey their meaning [22, p. 3]. An example of a tabulated relation is provided in table 2.1. By assigning labels to domains, a non-positional notation is introduced, where domains are called *attributes* and their labels can be used to refer to them [20, p. 18].

Name	Age	City
John Doe	42	London
Fulano de Tal	64	Madrid

Table 2.1: Tabulated example relation

Since relations are sets, the order of elements is unimportant, and as such it cannot be used to refer to an element. Codd states that a relation normally contains a combination of one or more domains that uniquely identify the relation's elements, and proposes designating that combination as the *primary key* of the relation [21, p. 4].

To support cross-referencing other elements, Codd further introduces *foreign keys* [21, p. 4] as combinations of one or more domains in a relation that correspond to the primary key of another relation (not excluding the same one).

2.2.1.1 Implementation of the Relational Model

Relational database management systems are an inexact implementation of the relational model. Relations correspond to *tables* in RDBM systems, tuples to *rows*, and elements

of the tuples to *columns*. Usually, there is more than one table in a single database instance. [23, p. 3–5]

In the relational model, relations are sets — they are unordered and cannot contain duplicate elements. In database systems that implement the relational model, these two constraints typically do not apply: rows in tables are ordered (although no particular order is guaranteed) and two or more rows with exactly the same values may be present.

Additionally, relational databases may allow incomplete data to be inserted: rows where some of the columns do not have a value. A special value, typically referred to as `NULL`, is then inserted in place of the empty columns by the database management system. [23, p. 5]

Concepts of primary and foreign keys are also implemented in common relational databases, the RDBMS then ensures the constraint expressed by those keys are not violated. More types of constraints may be supported: usually a *unique key*, which forces the combination of participating columns to have a unique value within the whole table, and a `NOT NULL` constraint, which prevents empty values in the column.

2.2.2 Structured Query Language

To increase interoperability between clients and various RDBM systems, Structured Query Language (SQL), originally developed for IBM's System R, was standardized and adopted by many other relational databases. [20, p. 85]

An example SQL statement that retrieves columns `Name` and `Age` from rows in table `People` that have `London` stored in the `City` column is provided in listing 2.1.

Listing 2.1: SQL `SELECT` statement

```
SELECT Name, Age FROM People WHERE City = 'London'
```

SQL is used not only to retrieve data from a database, its features can be divided into multiple parts:

Data Manipulation Language. Retrieval and manipulation (i.e. inserting, updating and deleting) data in tables.

Data Definition Language. Definition of tables and integrity constraints.

Transaction Control Language. Management of transaction processing within a particular connection to the database.

Data Control Language. Assignment of access rights to objects in the database.

Although SQL is standardized, vendors of RDBM systems often provide their own extensions, have different sets of supported data types, or make minor changes to the SQL syntax in general. Therefore it is not always guaranteed that an SQL statement written for one RDBM system can be successfully executed on another. For this reason, we consider SQL to be an implementation-specific model.

2.2.3 Oracle Database

Oracle Database is an enterprise-grade object–relational DBMS. It uses SQL for data querying and manipulation, and a procedural extension PL/SQL to implement stored procedures and triggers [24, p. 10]. This thesis aims to implement the transformation of an OntoUML model to a series of SQL DDL statements for Oracle Database version 12c.

The object-oriented extensions of Oracle Database allow users to define custom complex data types, inheritance and behavior [24, p. 11]. However, since our aim is to realize the OntoUML model in a relational database only, we will not make use of these OO aspects.

2.2.3.1 Realization of Constraints

Oracle Database natively supports common constraints explained in subsection 2.2.1.1: primary keys, unique constraints, and foreign keys. Custom constraints can be implemented in various ways:

Check Constraints. A `CHECK` constraint defines a boolean expression that must evaluate to `TRUE` for a DML statement to succeed [24, p. 119]. `CHECK` constraints are defined at table level and cannot contain subqueries [25].

Views. A view is defined by an SQL query; when querying the view, the underlying query gets executed and its result is returned to the client [24, p. 103]. Views may be updatable: in this case, `INSERT`, `UPDATE`, and `DELETE` statements may be executed on them, and the database will ensure only records visible by the view may be inserted. However, only views conforming to certain criteria are updatable. [26]

Triggers. Triggers are stored procedures that run whenever an `INSERT`, `UPDATE`, or `DELETE` operation occurs in a table. Being written in PL/SQL, they are the most powerful way of defining database constraints. [24, p. 120–121] An invalid DML operation can be prevented by the trigger raising an error.

2.3 Data Modeling

A data model is a kind of structural model that describes data elements, their properties, and the relationships between them. Data models usually have a lower level of abstraction than conceptual models — they are closer to the platform that will realize them.

2.3.1 Entity–Relationship Model

Various types and notations for data modeling exist, suitable for different use cases. For modeling the *schema* of a relational model, Peter Chen in 1976 proposed the Entity–Relationship (E/R) model [27]. An E/R model has the following components:

Entities. To reiterate, in a relational model, a relation is a set of tuples. In an entity–relationship model, a tuple represents an *entity* and, as such, a set of tuples is an *entity set*. In the E/R model, the entity set receives a name and is diagrammed as a rectangle.

Attributes. Attributes of a particular relation are inscribed in ellipses and connected to the appropriate entity set.

Relationships. Related entity sets are connected by a line with a diamond containing the label of the relationship in the middle. At the side of each entity set, its multiplicity is written out to express how many entities are participating in the relationship. Optionally, the *role* of an entity set may be indicated by attaching a label to its side.

An example E/R model in Chen's original notation, which describes a relation set with people and their passports, is shown in figure 2.5.

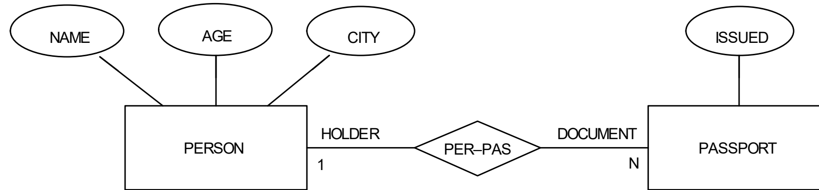

Figure 2.5: Entity–Relationship model

Following the principles of Chen's work, alternative notations for the E/R model later emerged. In the following subsection, we explain a UML-based notation, which is used in database diagrams throughout this thesis.

2.3.2 Data Modeling Profile of the Unified Modeling Language

The schema of a relational database can be modeled using the standard UML Class diagram, with classes representing tables, attributes representing columns, associations representing foreign keys, and operations representing constraints.

To introduce RDBMS-specific concepts in UML, Rational Software[1] proposed the UML Data Modeling Profile [28]. It is not a ratified extension by the Object Management Group, but is supported by a number of UML modeling software.

It provides stereotypes for tables, columns, primary, unique and foreign keys, and several other constructs commonly used by RDBM systems. It is not tied to a specific RDBMS implementation, however.

An example UML model that uses the Data Modeling Profile and models the same schema as the E/R model in figure 2.5 is provided in figure 2.6.

Figure 2.6: UML model using the Data Modeling Profile

2.4 Object Constraint Language

Object Constraint Language is a extension to UML, standardized by the Object Management Group [29, p. 3]. It is a declarative language used to specify constraints in the model that cannot be expressed by UML alone, and, since version 2, OCL can be used to define any expression (e.g. the body of an operation or a derived attribute) [29, p. 16].

In this thesis, the constraint definition aspect of OCL is considered. Three types of constraints can be defined, depending on when the condition should be checked:

[1] The company Rational Software was the original developer of UML in 1994, before it was adopted by the Object Management Group.

Invariants. An invariant is a constraint that must hold true for the complete lifetime of its context [29, p. 27]. In OCL, invariants are declared with the keyword `inv`.

Preconditions. A precondition is a constraint defined in the context of an operation that must be satisfied before the operation can be executed [29, p. 33]. In OCL, preconditions are declared with the keyword `pre`.

Postconditions. A postcondition constraints the effect of an operation — that is, the state of the system after the operation has been executed [29, p. 33]. In OCL, postconditions are declared with the keyword `post`. The expression in a postcondition refers to values after the completion of the operation; to access the value before the operation was executed, a `@pre` postfix can be used. An example of an OCL postcondition is provided in listing 2.2.

Listing 2.2: OCL postcondition asserting the `value` attribute is incremented by one

```
context Counter::increment() post:
  self.value = self.value@pre + 1
```

2.5 Realization of an OntoUML Model in SQL

The transformation of an OntoUML model to a series of SQL DDL statements for Oracle Database was proposed by Dr. Zdeněk Rybola in [6]. This section contains a summary of this transformation.

In his approach, the author divides the transformation into three phases [6, p. 3]:

1. transformation of the OntoUML model to plain UML,
2. transformation of the UML model to a relational model,
3. and conversion of the relational model to SQL DDL statements for Oracle Database.

The referenced work also focuses on preserving as many constraints derived from the OntoUML model as possible in the intermediate UML and relational models, as well as in the final realization in SQL.

2.5.1 Transformation of an OntoUML Model to UML

In this step, the types and relationships defined in an OntoUML model are transformed to standard classes and relationships in a UML model. In cases where the semantics of the OntoUML model cannot be accurately represented in standard UML, OCL constraint are derived to preserve the meaning of the original model.

Kinds and SubKinds

As Kind and SubKind types do not require special handling, they can be transformed to UML by simply converting them to classes. Their attributes, relationships, generalizations, and generalization sets are also transformed to the UML model as-is. [6, p. 60] An example of such transformation is provided in figure 2.7.

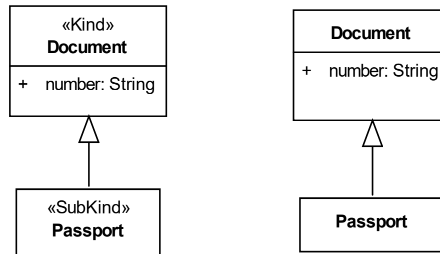

Figure 2.7: Transformation of an OntoUML Kind and SubKind (left) to UML (right)

In the discussion, an optimization of generalization sets with empty subclasses is proposed [6, p. 70–71]: When all subtypes in a generalization set are empty, they can be discarded and replaced by a *discriminator* attribute on the supertype. The attribute then indicates which types the individual currently instantiates. An example of such optimization is provided in figure 2.8.

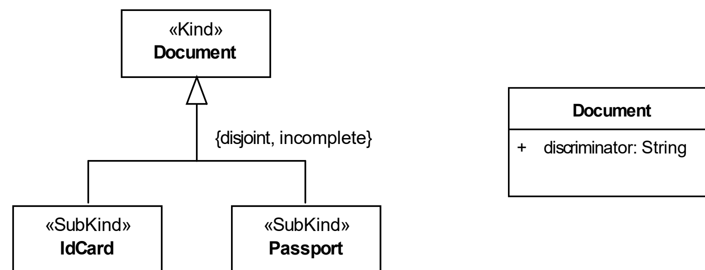

Figure 2.8: Optimization (right) of a generalization set (left)

Additionally, an OCL constraint that ensures only valid values can be stored in the discriminator column must be derived [6, p. 71]. The set of allowed values depends on the disjoint and covering properties of the generalization set: when the generalization set is incomplete, instantiation of the supertype must be allowed, when the generalization set is overlapping, instantiation of all combinations of the subtypes must be allowed. The OCL enumeration constraint derived from the OntoUML model in figure 2.8 is provided in listing 2.3.

Listing 2.3: OCL constraint restricting the values of a discriminator column

```
context Document inv EN_Document_Type:
  self.discriminator = 'Document' OR self.discriminator = 'IdCard'
    OR self.discriminator = 'Passport'
```

Roles and Relators

Role types are transformed to UML classes. The generalization relationship between the role and its identity bearer cannot be represented as generalization in UML, since roles are anti-rigid. Instead, the class generated from the role is connected to its identity bearer using a regular association, with multiplicity of `0..1` at the side of the role, and multiplicity of `1..1` at the side of the identity bearer. Additionally, the side of the identity bearer must be *immutable*. [6, p. 61–62]

Relators are also transformed to UML classes, and Mediation relationships to associations. Since relators are existentially dependent on other individuals, the opposite side

of the relator must be mandatory and immutable. [6, p. 65–66] Furthermore, as Material relationships are only derived from mediations and relators, they are not transformed to the UML model at all [6, p. 66].

An example of the transformation of roles, including the mandatory relator mediating them, is provided in figure 2.9.

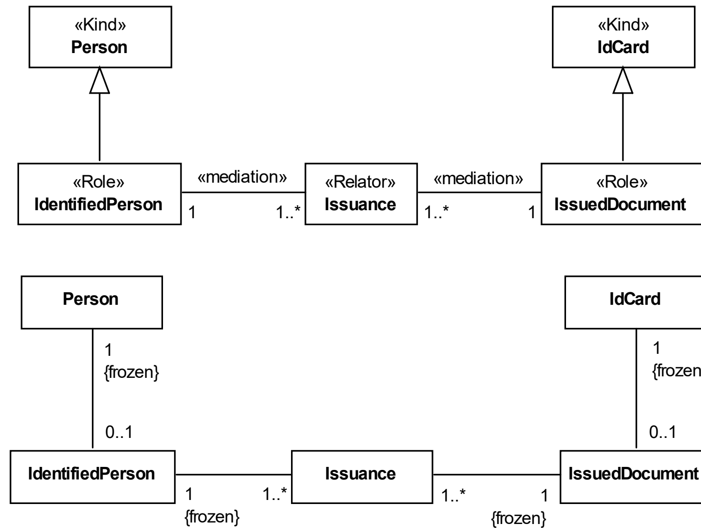

Figure 2.9: Transformation of an OntoUML Role (top) to UML (bottom)

Additionally, the possibility of optimization of empty roles is discussed: When only an empty class representing the original role is generated, it can be removed from the optimized model, and the identity bearer can be connected directly to the truth-maker. The minimal multiplicity at the side of the truth-maker must be set to zero so that individuals that do not instantiate the role can exist. [6, p. 73–74]

Furthermore, a discussion on the optimization of empty relators is provided: The empty class generated from a relator may be discarded, and the classes generated from the mediated roles connected by an association generated from the derived material relationship. [6, p. 74–75]

Phases

OntoUML uses the generalization relationship to connect phases to their identity bearer. Similar to the transformation of roles, this relationship cannot be represented by a generalization in UML. Since the phase partition is disjoint and covering, simple associations cannot be used either. Two alternative transformation options are discussed. [6, p. 62–63]

The first option is to use *exclusive phase associations*. In this approach, classes generated from phases are connected to their identity bearer via an association, and an OCL constraint is derived to ensure the identity bearer is connected to exactly one phase of the phase partition. [6, p. 63–64] An example of such transformation is provided in figure 2.10 and the derived OCL constraint in listing 2.4.

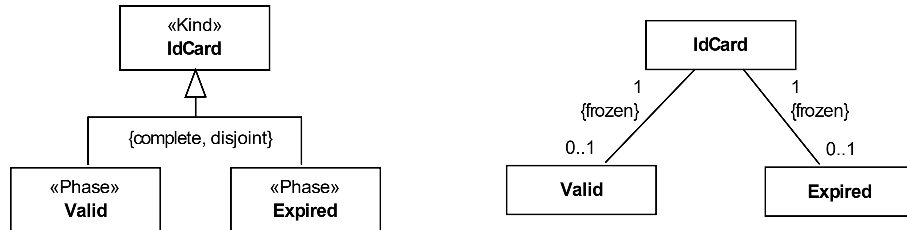

Figure 2.10: Transformation of an OntoUML Phase (left) to UML (right) using exclusive phase associations

Listing 2.4: OCL invariant realizing exclusive phase associations

```
context IdCard inv EX_IdCard_Condition:
  self.valid <> OclVoid XOR self.expired <> OclVoid
```

The second option is to create an *abstract phase* connected to the identity bearer using an association. Standard UML generalization set can then be used to specialize the abstract phase with concrete phases, without the necessity for additional OCL constraints. [6, p. 64–65] An example of the OntoUML model introduced in figure 2.10 using an abstract phase is provided in figure 2.11.

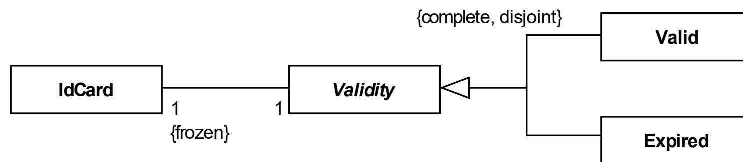

Figure 2.11: OntoUML Phase transformed to UML via an abstract phase

Optimization of empty phases is also discussed. When all classes generated from phases of a particular phase partition are empty, the entire partition can be replaced by a discriminator attribute on the identity bearer. As in the case of the optimization of generalization sets, an additional enumeration OCL constraint must be generated. [6, p. 72–73] An example of such enumeration constraint is provided in listing 2.5.

Listing 2.5: OCL invariant realizing phase attribute

```
context IdCard inv EN_IdCard_Condition:
  self.condition = 'Valid' OR self.condition = 'Expired'
```

Mixins

Mixins are transformed to classes in the UML model with their generalization relationships preserved. Since mixins are abstract, the generated classes must also be abstract. [6, p. 65]

Aspects

Mode and Quality types are transformed to standard UML classes. The Characterization relationship between an aspect and its bearer is transformed to a UML association.

Additionally, because aspects are existentially dependent on their bearer, the side of the bearer is mandatory and immutable. [6, p. 66–67]

An example of a transformation of a mode is provided in figure 2.12.

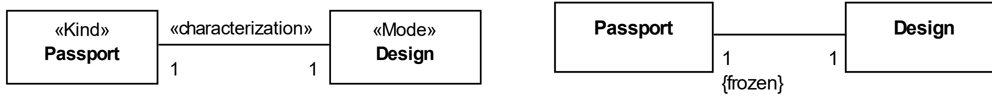

Figure 2.12: Transformation of an OntoUML Mode (left) to UML (right)

Furthermore, in the case of qualities, both sides of associations generated from Characterization relationships are to be made immutable. [6, p. 67]

Part–Whole Relationships

Aggregations expressing part–whole relationships are transformed to standard associations in the UML model — the aggregation feature of standard UML is not utilized. [6, p. 68–69]

As explained in subsection 2.1.3.2, part–whole relationships in OntoUML can express part shareability. However, as this constraint can be effectively represented by proper multiplicity at the side of the whole, only the multiplicity is used in the UML model to realize the shareability property. [6, p. 67–68]

To express the essentiality and inseparability of the relationship, appropriate sides of the UML association are set immutable. [6, p. 68]

2.5.2 Transformation of a UML Model to a Relational Model

After the OntoUML model is transformed to UML, the resulting UML model is transformed to a relational model. In [6], the author focuses on the transformation of constructs and constraints generated when processing the original OntoUML model. This section contains a summary of that transformation.

Classes and Attributes

UML classes are transformed to tables, and attributes are usually transformed to table columns. Additionally, an artificial primary key column is generated, so individuals can be uniquely identified within the database. [6, p. 78–79] An example of the transformation of a simple UML class is provided in figure 2.13.

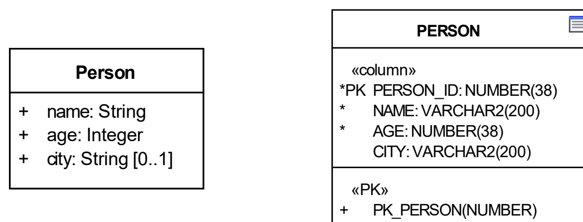

Figure 2.13: Transformation of a UML class (left) to a table (right)

When the minimal multiplicity of an attribute is zero, the corresponding column is defined as `NULLABLE`, otherwise as `NOT NULL`. When the maximal multiplicity of an attribute is greater than one, it cannot be realized as a simple column. Instead, it must be refactored to a new table that will store the individual values. The attribute table will be related to the

original table via a many–to–one association with the originally requested multiplicity. [6, p. 79]

Associations

In general, associations are realized by foreign keys. As this realization is uni-directional, the direction of the relationship is important:

- In the case of one–to–one associations with both mandatory or both optional sides, any direction can be used. [6, p. 82]
- For one–to–one associations where only one side is optional, the foreign key is placed in the table corresponding to the optional side of the association. [6, p. 82]
- In the case of many–to–one associations, the foreign key is placed in the table representing the many side of the association. [6, p. 81]
- Many–to–many associations cannot be directly realized by foreign keys. First, they must be decomposed to two many–to–one relationships; then the preceding rules apply. [6, p. 80–81]

An example of the transformation of a many–to–one association is provided in figure 2.14.

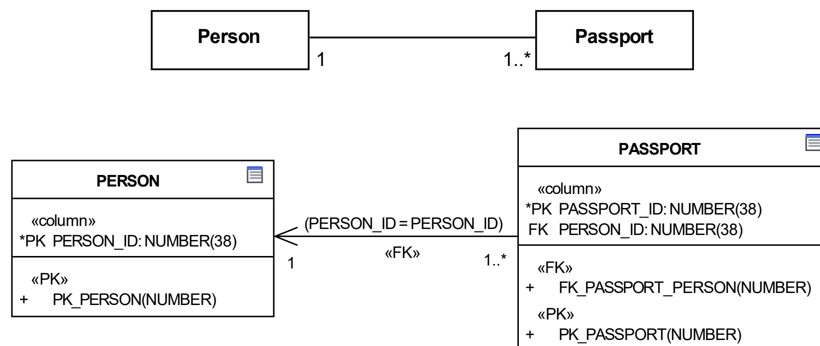

Figure 2.14: Transformation of a one–to–many association (top) to a foreign key (bottom)

Mandatoriness of the target side (in other words, the table that is being referenced by the foreign key constraint), if applicable, is ensured by setting the referencing column `NOT NULL` [6, p. 82]. Additionally, if the source maximal multiplicity is one, a unique key constraint is also added to the referencing column [6, p. 93].

In cases where the source side is mandatory, an additional OCL constraint must be generated [6, p. 93]. An example is provided in listing 2.6.

Listing 2.6: OCL mandatory multiplicity constraint

```
context PERSON inv MUL_PASSPORT_PERSON_ID:
  PASSPORT.allInstances()->exists(p | p.PERSON_ID = self.PERSON_ID)
```

UML also allows special multiplicities, that is, values different from `0`, `1` and `*`. In the case such multiplicity is used, a special multiplicity OCL constraint must be generated to ensure data validity [6, p. 93]. An example of a special multiplicity constraint that requires all people to own at least two but no more than four passports is provided in listing 2.7.

Listing 2.7: OCL special multiplicity constraint

```
context PERSON inv MUL_PASSPORT_PERSON_ID:
  def count: Integer =
    PASSPORT.allInstances()->count(p | p.PERSON_ID = self.PERSON_ID)
  2 <= count AND count <= 4
```

Generalizations

In [6], three approaches to the realization of a generalization relationship in a relational database are discussed:

1. single table for the entire generalization set [6, p. 85–88],
2. separate tables for all possible combinations of instantiated classes [6, p. 88–89],
3. and separate tables for each subclass with a reference to the table representing the superclass [6, p. 89–93] — *referencing tables*.

In their discussion, the author considers the realization of a generalization relationship via referencing tables the most useful [6, p. 105]. For this reason, in our approach, we focus on this option.

To realize the referencing tables approach, a table for the superclass and a table for each of the subclasses are generated. Additionally, the primary key in tables corresponding to the subclasses is constrained by a foreign key referencing the table representing the superclass. [6, p. 90]

Furthermore, a discriminator column is generated in the superclass table. The column indicates which classes are instantiated by the record. [6, p. 89–90] Possible values in the column depend on the disjoint and covering properties of the generalization set: for incomplete generalization sets, allowed values include the superclass, for overlapping generalization sets, allowed values include all combinations of subclasses. [6, p. 91]

An example of the transformation of a generalization set via referencing tables is provided in figure 2.15.

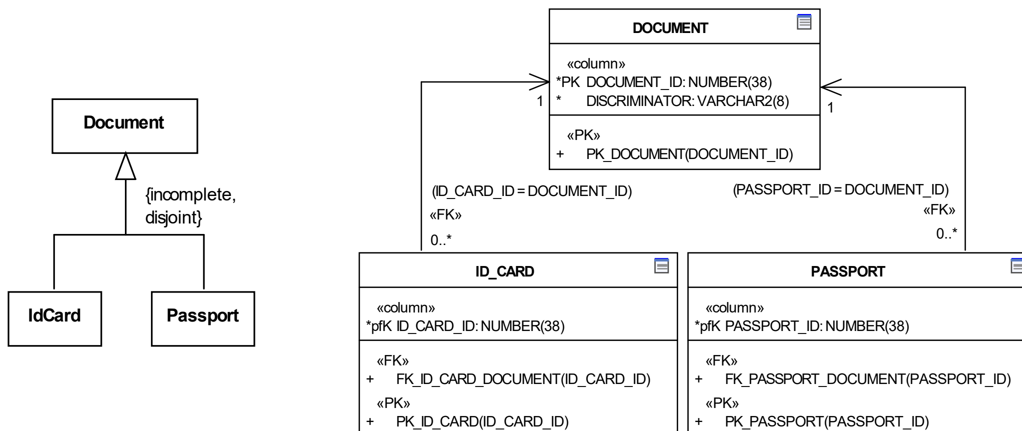

Figure 2.15: Transformation of a UML generalization set (left) to referencing tables (right)

An additional OCL constraint must be introduced to ensure the validity of the discriminator column, as well as the existence (and non-existence) of rows in tables representing the subclasses [6, p. 91–92]. An example of such constraint is provided in figure 2.8.

Listing 2.8: OCL constraint for a disjoint and incomplete generalization set realized by referencing tables

```
context DOCUMENT inv GS_Document_Type:
  def Document_Instance: Boolean = self.DISCRIMINATOR = 'Document'
    AND NOT(ID_CARD.allInstances()->exists(i | i.DOCUMENT_ID = self.DOCUMENT_ID))
    AND NOT(PASSPORT.allInstances()->exists(p | p.DOCUMENT_ID = self.DOCUMENT_ID))
  def IdCard_Instance: Boolean = self.DISCRIMINATOR = 'IdCard'
    AND ID_CARD.allInstances()->exists(i | i.DOCUMENT_ID = self.DOCUMENT_ID)
    AND NOT(PASSPORT.allInstances()->exists(p | p.DOCUMENT_ID = self.DOCUMENT_ID))
  def Passport_Instance: Boolean = self.DISCRIMINATOR = 'Passport'
    AND NOT(ID_CARD.allInstances()->exists(i | i.DOCUMENT_ID = self.DOCUMENT_ID))
    AND PASSPORT.allInstances()->exists(p | p.DOCUMENT_ID = self.DOCUMENT_ID)
  Document_Instance OR IdCard_Instance OR Passport_Instance
```

Immutability

In UML, associations (more precisely, their sides) and attributes may be set immutable. To preserve this constraint in the realization in a relational database, additional OCL constraints must be derived [6, p. 99]. Depending on the position of the immutability constraint, three cases are distinguished:

Immutability of the Target Table. Since the target side of the association is defined by a foreign key in the source side, it is possible to realize the immutability constraint by an OCL postcondition that ensures the value of the referencing column does not change [6, p. 100]. An example of this condition is provided in listing 2.9.

Listing 2.9: OCL postcondition realizing the immutability of target table

```
context PASSPORT::UPDATE() post IM_PASSPORT_PERSON_ID_UPD:
  self.PERSON_ID = self.PERSON_ID@pre
```

Immutability of the Source Table. To realize this immutability constraint, the disassociation of the record in the source table from the target record must be prevented. Therefore, changing the value of the foreign key that realizes the association must be disallowed, and removal of the entire source record must be disallowed as well. [6, p. 101–102] An example of an OCL postcondition that prevents changing the value of the foreign key is provided in listing 2.9, and an example of a constraint that prevents the deletion of the source record is provided in listing 2.10.

Listing 2.10: OCL postcondition preventing the deletion of the source side of an association

```
context PASSPORT::DELETE() post IM_PASSPORT_PERSON_ID_DEL:
  self.PERSON_ID = OclVoid OR NOT (PERSON.allInstances()->
    exists(p | p.PERSON_ID = self.PERSON_ID@pre))
```

Immutability of an Attribute. To disallow the change of an attribute, an OCL postcondition that ensures the value of the corresponding column is unchanged must be generated [6, p. 103]. An example of such a condition is provided in listing 2.9.

2.5.3 Realization of a Relational Model in Oracle Database

The last phase of the transformation of an OntoUML model to SQL is the transformation of the intermediate relational model to a series of SQL DDL statements. The author in [6] focuses on the realization in Oracle Database, therefore so does this thesis. This section contains a summary of this transformation.

OCL constraints specific to the transformation of an OntoUML model to SQL do not have built-in equivalents in Oracle Database. In [6], three approaches to the realization of these constraints are discussed [6, p. 118–128]:

Views. A database view can be used to hide data that violate the defined constraints. When executing `INSERT`, `UPDATE`, and `DELETE` operations on an updatable view, it can also prevent creating invalid data. However, violations caused by DML operations on the underlying tables cannot be prevented. [6, p. 118–120]

Check Constraints. A `CHECK` constraint defines a condition that all rows in a table must meet. However, since Oracle Database does not allow subqueries in the definition of a `CHECK` constraint, it cannot be used to realize constraints that reference multiple tables. [6, p. 120–121]

Triggers. Triggers are executed on specified events, such as when inserting or updating a row in a table. The operation can be canceled by throwing an exception from the trigger, which entirely prevents creating invalid data. Because they can contain very complex logic, they can be used to realize all constraints derived from the original OntoUML model during the transformation. [6, p. 122]

In [6], the author discusses the possible realization of each type of constraint using all three options. Later, they conclude the best realization is using triggers, except for the enumeration constraint, which is best realized by a `CHECK` constraint [6, p. 169]. In this thesis, only the recommended approaches are considered.

Tables and Columns

In SQL, tables and columns are defined via a `CREATE TABLE` statement [6, p. 116]. An example of this statement is provided in listing 2.11.

Listing 2.11: SQL `CREATE TABLE` statement

```
CREATE TABLE "PERSON" (
  "PERSON_ID" NUMBER(32) NOT NULL,
  "NAME" VARCHAR2(200) NOT NULL,
  "AGE" NUMBER(32) NOT NULL,
  "CITY" VARCHAR2(200));
```

Additionally, an `ALTER TABLE` statement must be generated for the definition of the primary key and unique constraints [6, p. 116]. An example of these statements is provided in listing 2.12.

Listing 2.12: SQL `ALTER TABLE` statements for primary key and unique constraint

```
ALTER TABLE "PERSON" ADD CONSTRAINT "PK_PERSON" PRIMARY KEY ("PERSON_ID");

ALTER TABLE "PERSON" ADD CONSTRAINT "UQ_PERSON_NAME" UNIQUE ("NAME");
```

Foreign Keys

To realize the foreign key constraint, an appropriate `ALTER TABLE` statement must be generated [6, p. 117]. To support efficient manipulation of a large number of records [6, p. 117–118], foreign keys are declared as *deferred* — the RDBMS enforces them only at the end of the transaction, not immediately upon inserting or updating data [30].

An example of the definition of a foreign key in SQL is provided in listing 2.13.

Listing 2.13: SQL `ALTER TABLE` statement realizing a foreign key

```
ALTER TABLE "PASSPORT" ADD CONSTRAINT "FK_PASSPORT_PERSON_ID"
  FOREIGN KEY ("PERSON_ID") REFERENCES "PERSON" ("PERSON_ID")
  DEFERRABLE INITIALLY DEFERRED;
```

Generalization Set Constraints

To realize the generalization set constraint, which ensures the existence of records in tables representing the subclasses based on the value of the discriminator column, a number of `CREATE TRIGGER` statements is generated.

First, the original OCL invariant may be violated by an `INSERT` or `UPDATE` operation on the table representing the superclass [6, p. 133]. An example trigger definition for this case is provided in listing 2.14.

Listing 2.14: SQL trigger for the superclass table realizing a generalization set constraint

```
CREATE TRIGGER "GS_DOCUMENT_TYPE"
BEFORE INSERT OR UPDATE ON "DOCUMENT" FOR EACH ROW
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(*) INTO "l_count" FROM DUAL WHERE (
(:new."DISCRIMINATOR" = 'Document'
 AND NOT EXISTS (SELECT 1 FROM "PASSPORT" WHERE "PASSPORT_ID" = :new."DOCUMENT_ID")
 AND NOT EXISTS (SELECT 1 FROM "ID_CARD" WHERE "ID_CARD_ID" = :new."DOCUMENT_ID"))
OR (:new."DISCRIMINATOR" = 'IdCard'
 AND NOT EXISTS (SELECT 1 FROM "PASSPORT" WHERE "PASSPORT_ID" = :new."DOCUMENT_ID")
 AND EXISTS (SELECT 1 FROM "ID_CARD" WHERE "ID_CARD_ID" = :new."DOCUMENT_ID"))
OR (:new."DISCRIMINATOR" = 'Passport'
 AND EXISTS (SELECT 1 FROM "PASSPORT" WHERE "PASSPORT_ID" = :new."DOCUMENT_ID")
 AND NOT EXISTS (SELECT 1 FROM "ID_CARD" WHERE "ID_CARD_ID" = :new."DOCUMENT_ID"))
);
IF "l_count" = 0 THEN
 raise_application_error(-20101, 'OCL constraint GS_Document_Type violated!');
END IF; END;
```

Then, inserting records in tables representing the subclasses, which reference an already existing record in the superclass table, is forbidden [6, p. 133]. As generalization sets are rigid, records in subclass tables must be created in the same transaction as the record in the superclass table. An example trigger definition for this case is provided in listing 2.15.

Listing 2.15: SQL subclass `INSERT` trigger realizing a generalization set constraint

```
CREATE TRIGGER "GS_DOCUMENT_TYPE_PASSPORT_INS"
BEFORE INSERT ON "PASSPORT" FOR EACH ROW
```

```
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(*) INTO "l_count" FROM DUAL WHERE EXISTS (
 SELECT 1 FROM "DOCUMENT" "d" WHERE "d"."DOCUMENT_ID" = :new."PASSPORT_ID");
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint GS_Document_Type violated!');
END IF; END;
```

Furthermore, in tables representing the subclasses, changing the reference to the superclass record in an `UPDATE` operation is forbidden [6, p. 133]. An example trigger definition for this case is provided in listing 2.16.

Listing 2.16: SQL trigger guarding the `UPDATE` operation for a generalization set constraint

```
CREATE TRIGGER "GS_DOCUMENT_TYPE_PASSPORT_UPD"
BEFORE UPDATE ON "PASSPORT" FOR EACH ROW
DECLARE "l_count" NUMBER := 0;
BEGIN
IF :old."PASSPORT_ID" <> :new."PASSPORT_ID" THEN
 SELECT COUNT(*) INTO "l_count" FROM DUAL WHERE EXISTS (
  SELECT 1 FROM "DOCUMENT" "d" WHERE "d"."DOCUMENT_ID" = :old."PASSPORT_ID"
   OR "d"."DOCUMENT_ID" = :new."PASSPORT_ID");
END IF;
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint GS_Document_Type violated!');
END IF; END;
```

Finally, the `DELETE` operation in tables representing the subclasses is forbidden in case the referenced record in the superclass table still exists [6, p. 133]. An example trigger definition for this case is provided in listing 2.17.

Listing 2.17: SQL trigger guarding the `DELETE` operation for a generalization set constraint

```
CREATE TRIGGER "GS_DOCUMENT_TYPE_PASSPORT_DEL"
BEFORE DELETE ON "PASSPORT" FOR EACH ROW
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(*) INTO "l_count" FROM DUAL WHERE EXISTS (
 SELECT 1 FROM "DOCUMENT" "d" WHERE "d"."DOCUMENT_ID" = :old."PASSPORT_ID");
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint GS_Document_Type violated!');
END IF; END;
```

Mandatory Multiplicity Constraints

For a mandatory multiplicity constraint, two `CREATE TRIGGER` statements must be generated. First, a trigger must check that when inserting or updating a record in the constrained table (i.e. the table where the foreign key is located), a related record exists in the related table [6, p. 146]. An example of this trigger is provided in listing 2.18.

Listing 2.18: SQL trigger definition realizing a mandatory multiplicity constraint at the constrained table side

```
CREATE TRIGGER "MUL_IDCARD_PERSON_ID"
BEFORE INSERT OR UPDATE ON "PERSON" FOR EACH ROW
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM "ID_CARD" "i"
 WHERE "i"."PERSON_ID" = :new."PERSON_ID";
IF "l_count" = 0 THEN
 raise_application_error(-20101, 'OCL constraint MUL_IDCARD_PERSON_ID violated!');
END IF; END;
```

Second, a trigger must be defined on the related table verifying that after executing an `UPDATE` or `DELETE` operation, all records in the constrained table are referenced by a record in the related table [6, p. 147]. An example of such trigger is provided in listing 2.19.

Listing 2.19: SQL trigger definition realizing a mandatory multiplicity constraint at the related table side

```
CREATE TRIGGER "MUL_IDCARD_PERSON_ID_REL"
AFTER UPDATE OR DELETE ON "ID_CARD"
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM "PERSON" "p" WHERE
 NOT EXISTS (SELECT 1 FROM "ID_CARD" "i" WHERE "i"."PERSON_ID" = "p"."PERSON_ID");
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint MUL_IDCARD_PERSON_ID violated!');
END IF; END;
```

Special Multiplicity Constraints

The realization of special multiplicity constraints is similar to mandatory multiplicity constraints, except the triggers are not merely checking for existence of the required records, but verify the number of records as well [6, p. 146–147].

First, an example of a `CREATE TRIGGER` statement verifying the appropriate number of related records exist is provided in listing 2.20.

Listing 2.20: SQL trigger definition realizing a special multiplicity constraint at the constrained table side

```
CREATE TRIGGER "MUL_PASSP_PERSON_ID"
BEFORE INSERT OR UPDATE ON "PERSON" FOR EACH ROW
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(*) INTO "l_count" FROM "PASSPORT" "p"
 WHERE "p"."PERSON_ID" = :new."PERSON_ID";
IF NOT ("l_count" BETWEEN 2 AND 4) THEN
 raise_application_error(-20101, 'OCL constraint MUL_PASSP_PERSON_ID violated!');
END IF; END;
```

Second, an example of a `CREATE TRIGGER` statement that checks all records in the con-

strained table have the appropriate number of related records is provided in listing 2.21.

Listing 2.21: SQL trigger definition realizing a special multiplicity constraint at the related table side

```
CREATE TRIGGER "MUL_PASSP_PERSON_ID_REL"
AFTER INSERT OR UPDATE OR DELETE ON "PASSPORT"
DECLARE "l_count" NUMBER := 0;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM "PERSON" "p" WHERE NOT (
 (SELECT COUNT(*) FROM "PASSPORT" "p2" WHERE "p2"."PERSON_ID" = "p"."PERSON_ID")
 BETWEEN 2 AND 4);
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint MUL_PASSP_PERSON_ID violated!');
END IF; END;
```

Exclusivity Constraints

To realize the association exclusivity constraint, a number of `CREATE TRIGGER` statements must be generated. First, when inserting or updating a record in the table representing the source side, a trigger must check that exactly one table contains a record referencing the checked one [6, p. 154]. An example of this trigger is provided in listing 2.22.

Listing 2.22: SQL trigger definition realizing an exclusivity constraint at the constrained table side

```
CREATE TRIGGER "EX_ID_CARD_PHASE"
BEFORE INSERT OR UPDATE ON "ID_CARD" FOR EACH ROW
DECLARE "l_count" NUMBER;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM DUAL WHERE (
 (EXISTS (SELECT 1 FROM "EXPIRED" "e" WHERE "e"."BEARER" = :new."CARD_ID")
  AND NOT EXISTS (SELECT 1 FROM "VALID" "v" WHERE "v"."BEARER" = :new."CARD_ID"))
 OR (NOT EXISTS (SELECT 1 FROM "EXPIRED" "e" WHERE "e"."BEARER" = :new."CARD_ID")
  AND EXISTS (SELECT 1 FROM "VALID" "v" WHERE "v"."BEARER" = :new."CARD_ID")));
IF "l_count" = 0 THEN
 raise_application_error(-20101, 'OCL constraint EX_ID_CARD_PHASE violated!');
END IF; END;
```

Furthermore, a `CREATE TRIGGER` statement must be generated for all tables representing the related sides. The trigger checks that when inserting to the related table, no record in the rest of the related tables references the same record in the constrained table as the one that is being inserted. [6, p. 155] An example of such trigger is provided in listing 2.23.

Listing 2.23: SQL trigger definition for an exclusivity constraint guarding the insertion of records to a related table

```
CREATE TRIGGER "EX_ID_CARD_PHASE_VALID_INS"
BEFORE INSERT ON "VALID" FOR EACH ROW
DECLARE "l_count" NUMBER;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM DUAL WHERE (
 EXISTS (SELECT 1 FROM "EXPIRED" "e" WHERE "e"."BEARER" = :new."BEARER"));
```

```
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint EX_ID_CARD_PHASE violated!');
END IF; END;
```

Additionally, a `CREATE TRIGGER` statement must be generated for all related tables, which checks that after executing an `UPDATE` or `DELETE` operation all records in the constrained table still have a corresponding record in exactly one related table [6, p. 155]. An example of this trigger is provided in listing 2.24.

Listing 2.24: SQL trigger definition for an exclusivity constraint guarding the change of records in a related table

```
CREATE TRIGGER "EX_ID_CARD_PHASE_VALID_UPD_DEL"
AFTER UPDATE OR DELETE ON "VALID"
DECLARE "l_count" NUMBER;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM "ID_CARD" "i" WHERE NOT (
 (EXISTS (SELECT 1 FROM "EXPIRED" "e" WHERE "e"."BEARER" = "i"."CARD_ID")
  AND NOT EXISTS (SELECT 1 FROM "VALID" "v" WHERE "v"."BEARER" = "i"."CARD_ID"))
 OR (NOT EXISTS (SELECT 1 FROM "EXPIRED" "e" WHERE "e"."BEARER" = "i"."CARD_ID")
  AND EXISTS (SELECT 1 FROM "VALID" "v" WHERE "v"."BEARER" = "i"."CARD_ID")));
IF "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint EX_ID_CARD_PHASE violated!');
END IF; END;
```

Enumeration Constraints

An enumeration constraint is realized in SQL by defining a `CHECK` constraint in the constrained table [6, p. 160–161]. An example of such constraint is provided in listing 2.25.

Listing 2.25: SQL `CHECK` constraint definition

```
ALTER TABLE "ID_CARD" ADD CONSTRAINT "EN_ID_CARD_VALIDITY" CHECK (
 "VALIDITY" = 'Valid' OR "VALIDITY" = 'Expired');
```

Immutability Constraints

The immutability constraint generated by an immutable attribute or for the target side of an association is realized by a trigger, which checks that an `UPDATE` operation did not change the value of the constrained column [6, p. 163]. An example of this trigger is provided in listing 2.26.

Listing 2.26: SQL trigger definition realizing an immutable column

```
CREATE TRIGGER "IM_PASSP_PERSON_ID_UPD"
BEFORE UPDATE ON "PASSPORT" FOR EACH ROW
BEGIN
IF :old."PERSON_ID" <> :new."PERSON_ID" THEN
 raise_application_error(-20101, 'OCL constraint IM_PASSP_PERSON_ID violated!');
END IF; END;
```

In the case of an association with an immutable source side, a trigger preventing the deletion of records in the source table that reference existing records in the target table is generated [6, p. 163–164]. An example of such trigger is provided in listing 2.27.

Listing 2.27: SQL trigger definition realizing immutable source side of an association

```
CREATE TRIGGER "IM_PASSP_PERSON_ID_DEL"
BEFORE DELETE ON "PASSPORT" FOR EACH ROW
DECLARE "l_count" NUMBER;
BEGIN
SELECT COUNT(1) INTO "l_count" FROM DUAL WHERE (EXISTS (
 SELECT 1 FROM "PERSON" "p" WHERE "p"."PERSON_ID" := :old."PERSON_ID"));
IF :old."PERSON_ID" IS NOT NULL AND "l_count" > 0 THEN
 raise_application_error(-20101, 'OCL constraint IM_PASSP_PERSON_ID violated!');
END IF; END;
```

Chapter 3

Analysis

This chapter provides an overview of the technologies used for the implementation, revises the transformation of an OntoUML model to SQL, and specifies requirements for the implemented software. It is structured as follows.

- In section 3.1, the modeling tool OpenPonk is described, and an existing implementation of the transformation of an OntoUML model to UML is discussed.
- In section 3.2, Pharo, the programming language used for the implementation, is introduced.
- In section 3.3, the transformation of an OntoUML model to SQL is revised for the latest version of OntoUML.
- In section 3.4, software requirements for the implementation are listed.

3.1 OpenPonk

OpenPonk is a modeling platform developed at the Faculty of Information Technology, Czech Technical University in Prague. It aims to support processes concerning software and business engineering. At the time of publication of this thesis, OpenPonk features support for various model types and notations, notably (within the context of this thesis) the following ones [5]:

UML Class diagram. Semi-complete support for the UML 2.5 metamodel is available. UML models can be exported and imported via the XMI format.

OntoUML. Supports the ontologically-focused profile for UML, has support for model validation and detection of anti-patterns. Implements transformation of an OntoUML model to UML.

More types of models and notations are supported, though they are not relevant to the topic of this thesis. On the other hand, OpenPonk does not support any notation specific to relational modeling, such as the Data Modeling profile for UML.

A screen capture of OpenPonk editing an OntoUML model is shown in figure 3.1.

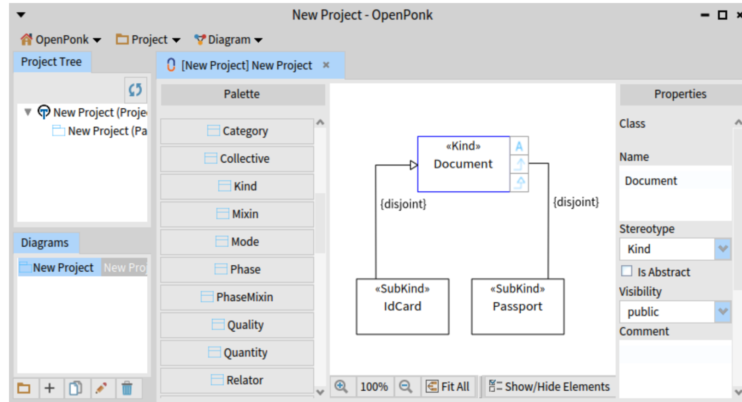

Figure 3.1: OpenPonk user interface

OpenPonk is written in Pharo 11, which is an implementation of the Smalltalk programming language and runtime environment. Its source code is publicly available under the MIT license.

This thesis aims to implement the transformation of an OntoUML model to SQL in OpenPonk, therefore this section discusses the implementation details of OpenPonk. Since no developer documentation or reference is available, the discussion is based on reverse-engineering efforts.

3.1.1 Data Model

This section explains the internal representation of both a UML and an OntoUML model in OpenPonk. Note that model and diagram are different concepts: a model is a data container for information about the investigated domain, and a diagram is a graphical visualization of the model.

The implementation in OpenPonk closely mirrors the UML standard — the elements available to the end user (for example, a class element or a Kind element) are instantiated by a chain of inheritance of classifiers, and by compositions of structural and behavioral features and stereotypes.

The inheritance hierarchy of `OPUMLClass`, which represents a class in the UML model, is shown in figure 3.2. Not all class properties are visible in the diagram.

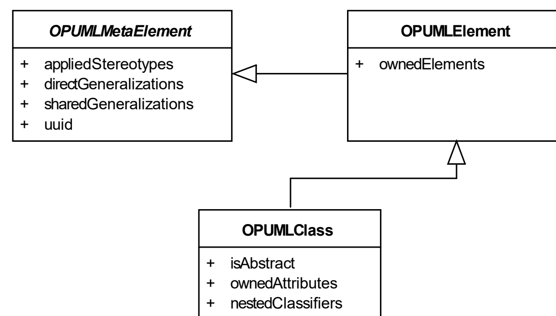

Figure 3.2: Inheritance hierarchy of an `OPUMLClass` class in OpenPonk

In figure 3.3, an object diagram is provided, which illustrates the composition of a model element that represents an OntoUML Kind. The object diagram shows a concrete instance of `OPUMLClass` class named `Person`, and the numerous classifiers and `OntoUMLKind` stereotype applied to it.

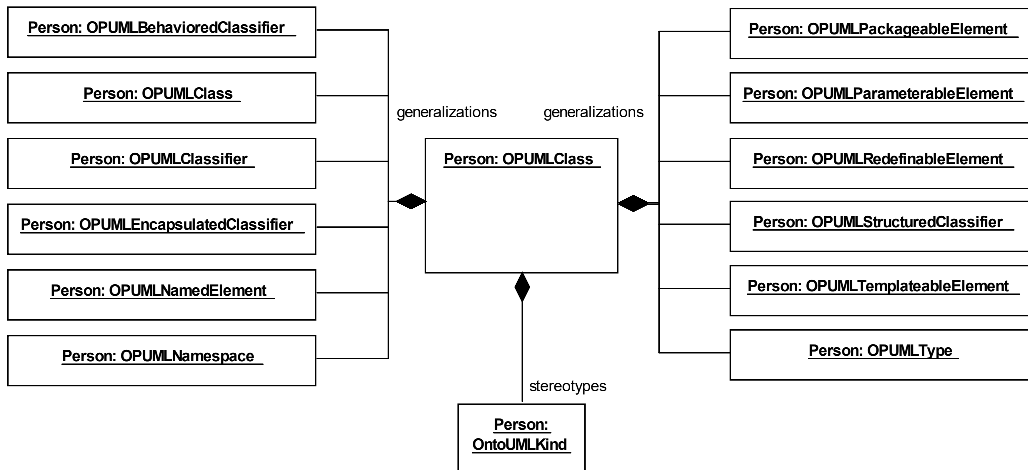

Figure 3.3: Object diagram of the composition of an element that represents an OntoUML Kind

In OpenPonk, the visual appearance of an element in a diagram, and its behavior and features in the model are defined together by the element's class and the applied classifiers and stereotypes.

The model itself is an instance of class `OPUMLPackage`, which declares the applied profiles and contains a collection of the model elements. A model belongs to a project; a project is represented by an instance of the `OPProject` class, and can contain multiple models.

3.1.2 Transformation of an OntoUML Model to UML

An existing implementation of the transformation of an OntoUML model to UML is present in OpenPonk. The transformation is initiated by a transformation controller, which delegates it to classes responsible for the transformation of specific element types. A sequence diagram illustrating the message flow for the transformation of classes is provided in figure 3.4.

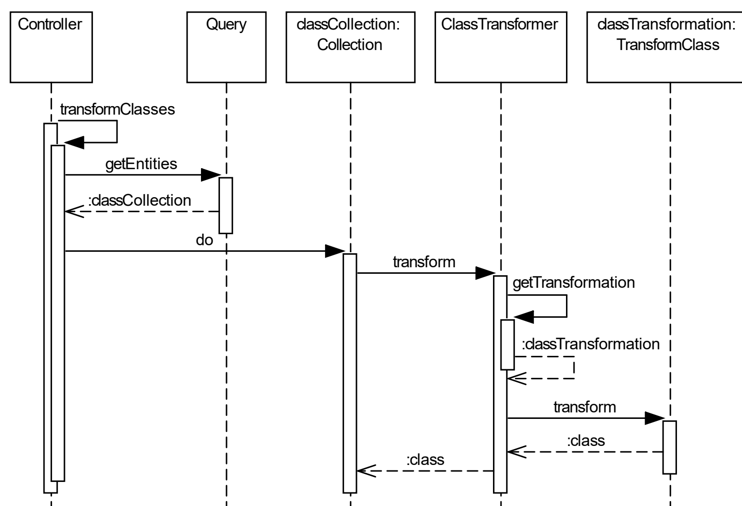

Figure 3.4: Sequence diagram of the transformation of an OntoUML class to UML in OpenPonk

3.1.2.1 Conformance to the Proposed Transformation

The implementation of the transformation of an OntoUML model to UML in OpenPonk was realized independently of the proposal in [6]. In this subsection, the differences in the existing implementation and the transformation principles developed in [6] are discussed.

Generalizations

1. Generalization sets are not transformed: they are missing from the resulting UML model.

Kinds and SubKinds

1. When the class is a part of a covering generalization set, the supertype should always be set abstract, this is not done automatically.

Roles

1. In addition to the generated association, an attribute referencing the role class is also generated in the identity bearer, and always named `attribute`. Only the association should be generated.
2. Wrong multiplicity is set at the side of the role: `1..1` is set, it should be `0..1`.
3. The side of the identity bearer should be set immutable, the current implementation leaves it mutable.

Phases

1. Only realization by exclusive phase associations is implemented, realization by an abstract phase is not available.
2. In addition to the generated association, an attribute referencing the phase class is also generated in the identity bearer, and always named `attribute`. Only the association should be generated.
3. Wrong multiplicity is set at the side of the phase: `1..1` is set, it should be `0..1`.
4. The side of the identity bearer should be set immutable, the current implementation leaves it mutable.
5. Multiple phase partitions can be defined within a single type, the current implementation does not distinguish between them in the resulting UML model.
6. An OCL exclusivity constraint should be generated, the current implementation only puts "XOR" as the name of all the associations.

RoleMixins and PhaseMixins

1. They should be transformed to abstract classes that generalize their subtypes. The current implementation realizes them as concrete classes associated with the subtype.

Mixins and Categories

1. The resulting UML class should be set abstract, they are transformed to concrete classes by the current implementation.

Relators

1. Derived material relationships should be discarded from the UML model to prevent redundancy and inconsistencies, the current implementation retains them.

2. In the transformed material associations, the side opposite of the relator should be set immutable, it is currently left as mutable.

Qualities, Modes, and Quantities

1. The side of the bearer should be set immutable, the current implementation leaves it mutable.

Part–Whole Relationships

1. Currently, the shareability of the part is transformed to UML aggregation (empty diamond to empty diamond) or composition (full diamond to full diamond). Instead, it should be discarded.

2. Essential and inseparable constraints should make the appropriate side of the transformed association immutable; currently, they are ignored.

3.2 Pharo

Pharo is a pure object-oriented and dynamically-typed programming language and run-time based on Smalltalk. It has fundamental differences in comparison to object-oriented languages that evolved from C (such as C++ or Java) [31]:

Run-Time Class and Object Modification. Classes and objects can be modified while the program that is using them is executing. For example, an instance variable can be added to already existing instances of a class, or an object can be replaced by another one and all references will immediately point to the new object.

Dynamic Inheritance. The supertype of a class can be changed during run-time.

Pure Object-Oriented Approach. By design, everything in Pharo is an object. That includes, for example, even integers, booleans, classes, or messages. Objects communicate by sending messages (instead of invoking methods).

Software as Objects. Pharo does not work with source code text, instead, it represents the software as objects. Its integrated development environment (IDE) must be used to view, edit, and export source code.

Pharo code is compiled and executed in a virtual machine, which is available for Windows, MacOS, and Linux operating systems. The virtual machine executes bytecode from an *image*, to which it can also persist the immediate state of the virtual machine and restore it during the next execution. The image is stored on disk as a single file.

3.2.1 Code Organization

In Pharo, classes are organized into *packages*. Packages may not be nested, but they can be further divided into *tags*. Instead of nesting, dashes in package or tag names are used, for example `Pkg-SubPkg`.

Packages and tags do not represent namespaces — there cannot exist multiple classes with the same name, even in different packages. To prevent collisions, it is recommended to prefix the name of the class with an abbreviation of the project name [32, p. 9].

Smalltalk, and therefore Pharo as well, uses a particular syntax for message (method) sending: a unary message (i.e. without arguments) is sent using its name only (e.g. `aFile close`), while in the case of keyword messages (i.e. with one or more arguments), the arguments are inserted in the message name (e.g. `aFile write: 'hello' atLine: 42`).

3.2.2 Source Code Versioning

Pharo does not store source code in the image — every time a change made in the IDE is saved, the code is compiled and the virtual machine works only with the compiled representation[1]. The original source code is recorded next to the image file in a *changes* file. Additionally, a *sources* file with the source code of Pharo's built-in classes is provided with each base Pharo image; the changes file contains all changes that occurred in the image since the sources file was generated. [33, p. 10–11]

Standard practice in software engineering is to check in source code to a version control system, which records history, allows annotating changes, and enables collaboration between multiple developers. Common language-agnostic version control systems record changes made to source code files.

The purpose of the changes file generated by Pharo is not to serve as a version control system — because all changes are recorded in a single file, it would make collaboration impractical, and it would not be possible to limit which classes should be watched for changes.

Instead, Pharo can integrate with Git[2], a distributed version control system, via its own implementation of a Git client called Iceberg. In Iceberg, packages are linked with a Git repository. When committing changes, Iceberg exports the source code of classes in the linked packages to individual files, which are then stored in the repository.

Pharo also includes a Smalltalk-specific version control system named Monticello[3]. Since OpenPonk packages use Git, we focus on Git as well.

3.2.3 Package Management

To use code written by other developers, users must download and install it into their Pharo images. Alternatively, the developer may offer a pre-built image with their software already included — this just shifts the responsibility of installing packages into the image on the developer, however.

Pharo includes Metacello[4], a package management system for Smalltalk. It allows users to load packages from repositories and resolve package dependencies.

A Metacello project is defined via a *baseline*. The baseline specifies what packages are included in the project, the order in which they must be loaded, and dependencies on external projects. The baseline is declared as a class with a name beginning with `BaselineOf` and ending with the name of the project, and is stored within a package with the same name as the class. The baseline package must be included in the repository with the source code of the project.

Metacello can install projects from local directories, remote repositories, and is also integrated with Iceberg.

[1] It is possible to run an image without the associated changes file. In that case, the IDE will not be able to display the source code of the objects in the image.

[2] More information about Git are available at `https://git-scm.com/`.

[3] The Monticello home page is available at `http://www.wiresong.ca/monticello/`.

[4] More information about Metacello can be found at `https://github.com/Metacello/metacello`.

3.2.4 Graphical User Interface

The graphical user interface (GUI) of Pharo is realized in a way similar to virtual machines running entire operating systems: a single window in the host system is opened, and windows and other graphics are drawn inside the host window. A screen capture of the contents of a Pharo host window on the first launch of a standard image is shown in figure 3.5.

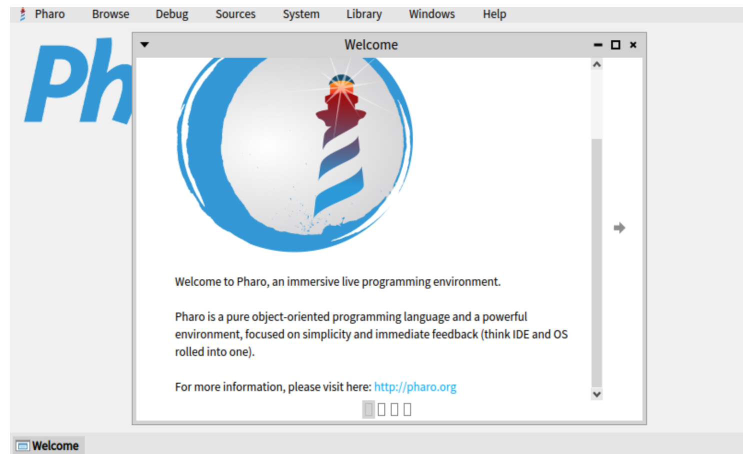

Figure 3.5: Pharo user interface

Pharo provides a standard GUI framework called Spec. Internal Pharo tools as well as OpenPonk use Spec to build their user interface. Spec provides basic GUI widgets, such as labels, buttons, text boxes, etc. Developers create more complex widgets and entire user interfaces by composing and customizing the widgets provided by Spec. [34, p. 1–2] User interfaces created in Spec are platform-independent: behavior and appearance of widgets are different from native applications, and are the same across all platforms that run Pharo.

3.3 Transformation Revision for the Latest OntoUML Version

The transformation of an OntoUML model to UML proposed in [6] was developed against the then-up-to-date version of OntoUML. In this section, the differences between the latest OntoUML version available at the time of writing of this thesis and the OntoUML version used in [6] are discussed, and the transformation principles summarized in subsection 2.5.1 are revised to be applicable to the current OntoUML version.

Note that OntoUML is not formally versioned, therefore the comparison is based on the description of OntoUML constructs provided in [6] and in subsection 2.1.3.

3.3.1 Changes in OntoUML

1. Quality and Mode are no longer called Moments, they are now categorized as Aspects. The semantics important for the realization of the transformation remain unchanged.

2. Relators were originally considered as Moments, the current version of OntoUML categorizes them as rigid sortals. The original transformation remains valid, as the

relevant ontological properties did not change: in both OntoUML revisions, relator types are rigid and provide identity to their instances.

3. Relationship stereotype «Constitution», originally used to characterize a quantity by a sortal other than a quantity [6, p. 46], no longer exists and has no direct replacement. Therefore, the part of the transformation responsible for transforming this stereotype can be discarded.

The comparison indicates that the transformation of an OntoUML model to UML summarized in subsection 2.5.1 need not be amended for the latest OntoUML version.

3.3.2 Enumerations

The transformation of UML enumerations is not considered in [6], however, they are used in real-world conceptual models. For this reason, we extend the transformation of a UML model to a relational model with the support for enumerations.

When transforming an association between a class and an enumeration, a new `VARCHAR` column is generated in the table representing the class. The literals of the enumeration are collected and a new enumeration OCL constraint is generated, which restricts the possible values of the column to the enumeration literals. An example of an OCL enumeration constraint generated from an enumeration with literals `FAIL` and `PASS` is provided in listing 3.1.

Listing 3.1: OCL enumeration constraint

```
context TEST inv EN_TEST_RESULT:
  self.RESULT = 'FAIL' OR self.RESULT = 'PASS'
```

UML enumeration elements themselves do not appear in the transformed relational model, they only materialize as columns and OCL constraints.

3.4 Software Requirements

The main objective of this thesis is to implement the transformation of an OntoUML model to SQL. Following the best practices of software engineering [35, p. 63], this section discusses the requirements imposed on the implementation. The requirements listed in this section serve as the input specification for the design and realization of the implementation.

3.4.1 Functional Requirements

First, functional requirements are discussed. Functional requirements describe the expected functionality of the software: the interface and the behavior of the implementation [35, p. 64].

Functional Req. 1 (Transformation of OntoUML Model to SQL)**.** The software transforms any input OntoUML model to a sequence of SQL DDL statements that realize the conceptual model in Oracle Database. The transformation shall adhere to [6] and its revision for the latest version of OntoUML.

In the transformation of OntoUML to UML, all proposed transformation options and optimizations shall be implemented. In the transformation of UML to a relational model, generalizations shall be transformed via referencing tables. Finally, when transforming

the relational model to SQL, only the realization of OCL constraints in SQL recommended in [6, p. 169] needs to be implemented.

Functional Req. 2 (Inspection of Intermediate Models)**.** The software allows the user to inspect the intermediate UML and relational models generated during the transformation to SQL.

Functional Req. 3 (Transformation Interactivity)**.** During the transformation, when multiple transformation options of a particular construct are implemented (e.g. transformation of a phase by exclusive associations or an abstract phase), the software allows the user to choose which option should be used. Similarly, the user chooses whether to apply the implemented model optimizations or not.

Functional Req. 4 (Model Iteration)**.** The user interface of the software shall support iterative model development: it should be convenient for the user to go back to the original OntoUML model, make changes, and execute the transformation again.

3.4.2 Non-Functional Requirements

In this subsection, non-functional requirements are listed. Non-functional requirements specify the properties and characteristics of the implementation (e.g. used technologies) and the rest of the software development process (e.g. testing) [35, p. 65].

Non-Functional Req. 1 (Integration with OpenPonk)**.** The software loads the input OntoUML model from OpenPonk. Also, the entire transformation process is started from the OpenPonk user interface. Furthermore, the software should be easily installable to OpenPonk as a plugin, and be ready to be integrated into the official distribution of OpenPonk.

Non-Functional Req. 2 (Transformation Extensibility)**.** It shall be possible to easily modify the implemented transformation later. In particular, it should be possible to add transformations to different SQL dialects while reusing the transformation to UML and the relational model.

Non-Functional Req. 3 (Verification of Correctness)**.** The correct realization of the transformation and constraints should be verified against the model in [6]; for constructs and constraints not covered by the model in [6], custom test cases should be developed.

Chapter **4**

Design

In this chapter, the basis for the implementation of the transformation, and the data model are constructed, the transformation process is refined, and user interaction is designed. The chapter is structured as follows.

- Section 4.1 lists common features and requirements for the implementation of a model transformation.
- In section 4.2, a basis for the implementation of a model transformation is defined.
- In section 4.3, the Moco library, which will contain the implementation of the transformation, is introduced.
- In section 4.4, the representation of OntoUML, UML, relational, and SQL models is designed.
- In section 4.5, the transformation of an OntoUML model to SQL is deconstructed into seven rounds.
- In section 4.6, the user interface is designed.

4.1 Framework for Transformations

As described in section 2.5, the transformation of an OntoUML model to SQL is realized in three phases. First, the OntoUML model is transformed to UML, then to a relational model, and finally to SQL. In line with the practices of good software design, a common framework should exist, which will implement common tasks related to model transformations, and which will be used by the particular transformation phases.

In this section, we describe the features that such a framework should have in order to make the implementation of transformations efficient and convenient for the developer.

1. The transformation framework should be independent of the type of the particular transformation. As at least three types of models will be transformed (OntoUML, UML and relational), a transformation framework that provides generic utilities for manipulating the models and for execution of the transformation should exist.
2. It should be possible to decompose the transformation as much as possible. Given the algorithm for the transformation of certain elements can be quite complex, the cognitive complexity should be reduced by splitting the transformation code as much

as the developer of the transformation deems appropriate. For example, the developer should not be forced to implement the transformation of the entire model within a single class.

3. The transformation framework should be able to handle transformations with dependencies. For example, it should be possible to defer an individual transformation of an element that requires a transformation of a related element to be executed first.
4. Transformations should be able to produce additional metadata that can be used in further transformations. For example, when two transformations generate otherwise indistinguishable elements that need to be transformed differently in another transformation phase, it should be possible to attach information for later distinction.
5. It should be possible to query which elements were generated from a particular input element. For example, when transforming a relationship between two elements, it is necessary to retrieve the elements that were generated from the sides of the relationship, so they can be connected in the transformed model as well.
6. The transformation framework should support the definition of multiple transformation options of a single element. During the execution of the transformation, the user should be asked which option should be executed. It should be possible to generate the options dynamically, for example, based on the contents of the currently transformed element.
7. It should be possible to generate multiple output elements from a single input element, and even no output elements at all. In other words, the cardinality of the relationship between an input element and its output elements should be 1:N, where N $\in \{0, 1, ...\}$.

OpenPonk already implements a transformation of an OntoUML model to UML (see subsection 3.1.2). However, the existing implementation does not follow the transformation proposed in [6]. Furthermore, the current implementation via controllers and transformers lacks some of the features that were identified when specifying the requirements for the transformation framework. Therefore, going forward, we will not use the existing implementation. Instead, a new transformation framework will be designed and implemented.

4.2 Transformation Engine

In this section, we design the realization of the transformation framework specified in section 4.1. The result of the design is the Transformation Engine — a framework that supports the implementation of a transformation of any kind of model.

First, the containers for the model and individual elements, which support the realization of requirements listed in section 4.1, are described in definitions 4.1 and 4.2. An example of their usage is provided in example 4.1.

Definition 4.1 (Wrapped Model)**.** Similar to a regular model, a Wrapped Model is a container for model elements. A wrapped model is constructed from another existing (regular) model by wrapping each of its elements in a wrapped element.

Definition 4.2 (Wrapped Element)**.** A Wrapped Element is a container that stores one model element and a set of additional metadata necessary for the transformation process.

Example 4.1 (Purpose of the Wrapped Model and Wrapped Elements)**.** As the transformation process is implemented using narrowly focused transformation rules (discussed later), there may be one rule that transforms OntoUML kinds to UML classes, and an entirely

separate rule for transforming associations between OntoUML kinds to associations between UML classes. The latter rule needs to know which UML classes the OntoUML kinds were transformed to, in order to create a UML association relationship between them. This information is stored in the wrapped element's metadata.

In definition 4.3, we describe the realization of the core of the transformation framework. Furthermore, the algorithm for the transformation engine is provided in listing 4.1.

Definition 4.3 (Transformation Engine)**.** The Transformation Engine (T. E.) is a set of three nested program loops. The input of the T. E. is a wrapped model. The output of the T. E. is another wrapped model, with transformation rules applied. The outmost loop in the T. E. is a *do–while* loop that repeats until the output model no longer changes. The middle loop in the T. E. iterates over a list of transformation rules, in the order of their priority. The innermost loop is again a *do–while* loop that ends when no changes to the output model are made.

Listing 4.1: Algorithm of the Transformation Engine

1: **repeat**
2: **for all** *rule* $\in$ *transformationRules* **do**
3: **repeat**
4: execute(*rule*)
5: **until** $\neg$ didChange(*outputModel*)
6: **until** $\neg$ didChange(*outputModel*)

This definition of the T. E. allows the realization of deferred transformations — when a particular element cannot be transformed due to unmet dependencies, the transformation rule can skip the element, then, the dependencies will be transformed, and the original transformation rule will be executed again in the subsequent iteration of the outermost loop.

Finally, the definition 4.4 describes how a particular transformation logic is specified for the execution within the transformation engine.

Definition 4.4 (Transformation Rule)**.** A Transformation Rule (T. R.) is an object. Each transformation rule defines a `priority` property that determines the order of execution of a collection of multiple transformation rules (from highest to lowest). Furthermore, each T. R. implements an `execute` method, which performs the transformation. In a T. R., it is forbidden to modify the input model, except for its metadata and the metadata of its elements. Additionally, a T. R. must handle repeated executions and avoid transforming a single element multiple times.

Note that the definition 4.4 imposes little restrictions on the actual transformation logic. This allows implementations of transformation rules of various complexity — for example, a single rule that atomically transforms multiple elements, or a rule that does not operate on the input model at all, but manipulates only the output model.

It follows from listing 4.1 that the transformation stops once the output model ceases to change. Therefore, the requirement for correct handling of multiple executions of a single transformation rule in definition 4.4 is important. It is the responsibility of transformation rules to ensure the transformation does not end up in an infinite loop.

Furthermore, the transformation engine does not enforce that all elements in the input model be transformed. It is the responsibility of the client to verify the transformation result is complete.

4.3 Moco Library

In order not to interfere with upstream OpenPonk source code, features implemented as a part of this thesis are bundled in a library called Moco[1] separate from OpenPonk packages. OpenPonk's source code is not modified at all.

As Moco is an extension for OpenPonk, it is also written in Pharo 11. The library consists of the following parts:

- the transformation engine and data classes for the wrapped model,
- code for the user interface and integration with OpenPonk,
- data classes for OntoUML, UML, the relational model, SQL, and OCL,
- and the implementation of transformation rules for the transformation rounds listed in section 4.5.

4.4 Data Model

In this section, data structures for representing the input, intermediate and output models are designed. In total, the transformation deals with five kinds of models:

- OntoUML,
- UML Class model,
- OCL,
- the relational model,
- and SQL statements.

The data structures are intended for use within the transformation process. Therefore, they are designed for easy manipulation by transformation rules, and they may not be able to represent all features of the particular model kind.

4.4.1 UML Data Model

To represent a UML model in memory, it is possible to either reuse the data model implemented in OpenPonk, or develop a custom one. As it was explained in subsection 3.1.1, OpenPonk's representation of UML models is versatile, however, that also makes it rather complex.

When considering models that our transformation is applicable to, only a subset of the features that OpenPonk's data model offers is required:

Classes. Name, indicator of whether the class is abstract or concrete, and attributes.

Attributes. Name, data type, multiplicity, immutability, visibility, and indicator whether the attribute is static.

Associations. Name, indicator whether the association is derived, and source and target sides. For association sides, name, class, multiplicity, navigability, visibility, aggregation, composition, and immutability need to be represented.

Enumerations. Name and enumeration literals.

Generalizations. Generalization relationship, generalization sets. For generalization sets, the disjoint and covering constraints need to be represented as well.

[1] Moco is an acronym that stands for "model converter".

In order to simplify the development of transformation rules that work with UML models, a custom data model is implemented. The data model is designed specifically for use within the transformation to a relational model — for example, behavioral features, such as operations, are ignored in our data model.

The design of the data model for UML models is provided as a class diagram in figure 4.1.

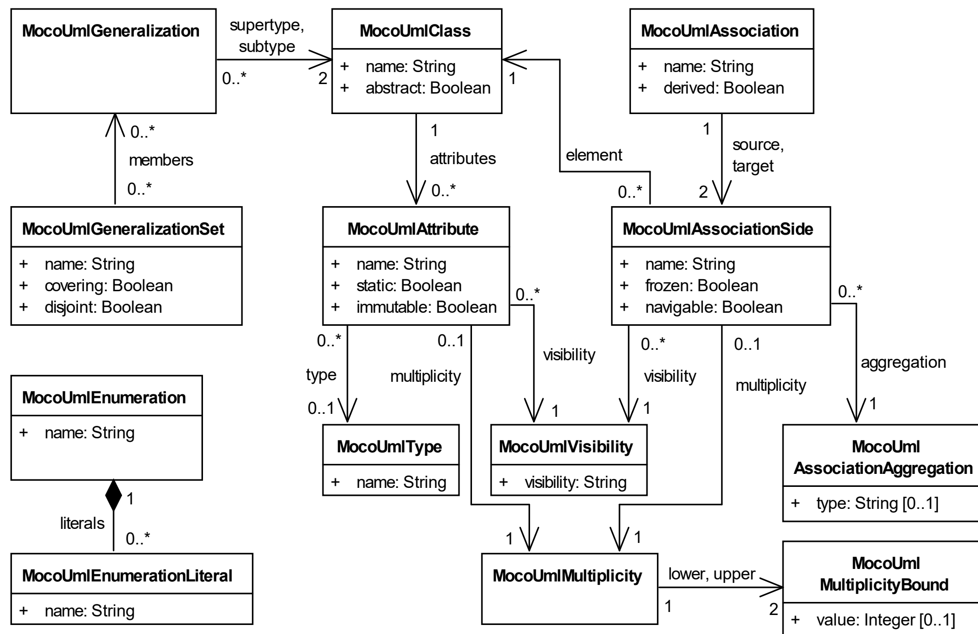

Figure 4.1: Class diagram of UML data model

4.4.2 OntoUML Data Model

As OntoUML can be considered a profile of UML, reasoning in subsection 4.4.1 is applicable to the representation of OntoUML models as well. In addition to standard UML constructs, OntoUML-specific features need to be supported:

Class Stereotypes. «Kind», «SubKind», «Role», «Phase», «Relator», «Collective», «Quantity», «RoleMixin», «PhaseMixin», «Mixin», «Category», «Mode», «Quality».

Relationship Stereotypes. «Formal», «Material», «Mediation», «Characterization», «Structuration», «ComponentOf», «Containment», «MemberOf», «SubCollectionOf», «SubQuantityOf».

Associations. Shareability of association sides, association essentiality and inseparability. In contrast with UML, aggregation and composition are not used in OntoUML.

To represent class and relationship stereotypes, a different approach than in OpenPonk is adopted: instead of attaching stereotypes to a generic UML class, the proposed data model uses inheritance, as illustrated in figure 4.2.

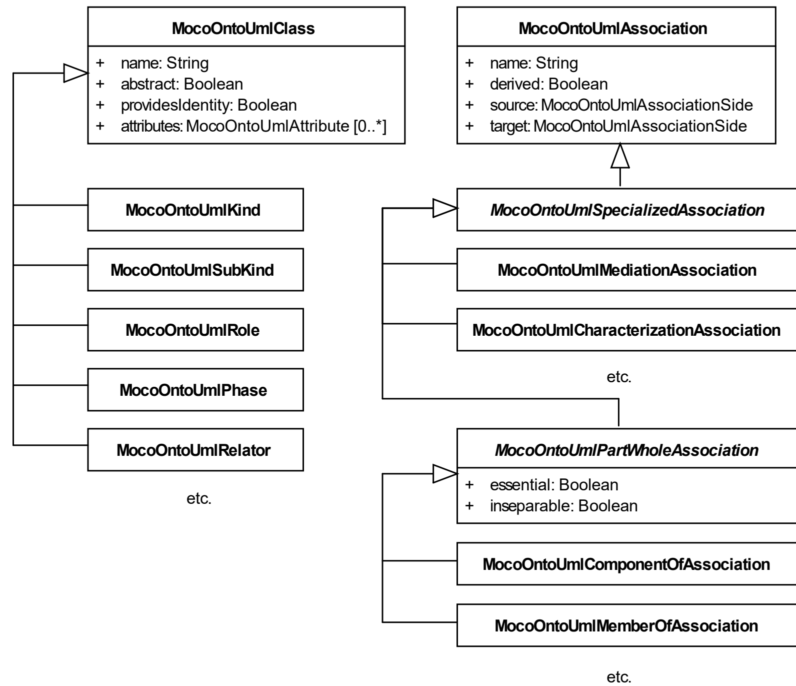

Figure 4.2: Class diagram of an excerpt from the OntoUML data model

In Pharo, multiple inheritance is not allowed [33, p. 131], whereas in UML, multiple stereotypes can be applied to a single classifier instance. Nevertheless, in OntoUML, no two stereotypes can be applied to a type at the same time, therefore the realization by inheritance is sufficient.

Some classes representing OntoUML class and relationship stereotypes (as listed in subsection 2.1.3) are omitted from figure 4.2. They follow the same nomenclature and generalization pattern as the classes shown in the diagram.

Primitives, such as `MocoUmlMultiplicity`, are reused by the OntoUML data model. Complex structures and concepts with changed meaning or additional properties (e.g. `MocoOntoUmlAttribute`) are not reused nor inherited but reimplemented.

4.4.3 Relational Data Model

To perform transformations to and from the relational model, it is necessary to represent the following constructs:

Tables. Name, columns, and constraints.

Columns. Name, nullability, and data type.

Constraints. Primary keys, unique constraints, and foreign keys need to be represented. Primary keys and unique constraints declare the columns to which the constraint applies, foreign keys declare the referencing columns and the referenced table and columns referenced within. In addition to constraint-specific properties, each constraint is identified by its name.

Relational data modeling is not supported in OpenPonk, therefore no data model can be readily reused. In line with the realization of UML and OntoUML data models, the relational data model is also implemented from scratch. A class diagram of the proposed data model is provided in figure 4.3.

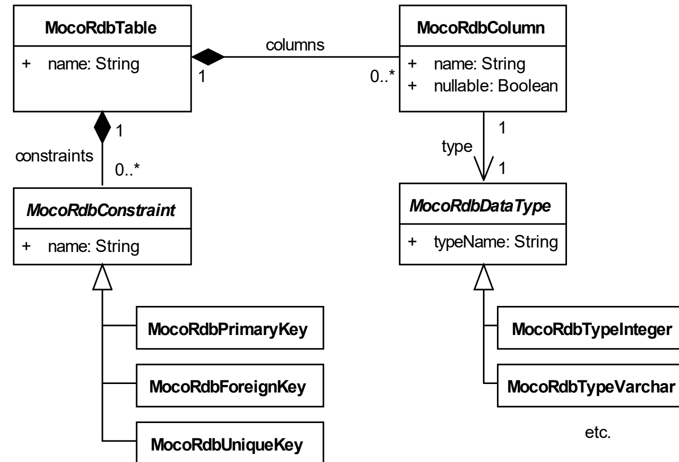

Figure 4.3: Class diagram of an excerpt from the relational data model

In figure 4.3, some classes representing data types are omitted. The full list of data type classes is provided in table 4.1.

4.4.3.1 Data Types

The proposed relational data model is designed as implementation-independent. As far as our approach is concerned, all of the mentioned RDB constructs are implemented equally by common RDBM systems, except for data types.

Although several data types are predefined in the SQL standard, RDBMS implementations deviate from this part of the standard, and the supported sets of data types vary between them. For this reason, in table 4.1, we introduce a custom set of data types, independent of any particular RDBMS implementation.

Data Type	Description	Class Name
`INTEGER`	Whole number.	`MocoRdbTypeInteger`
`DOUBLE`	Floating-point number.	`MocoRdbTypeDouble`
`CHAR`	Single character.	`MocoRdbTypeChar`
`VARCHAR`	Variable-length string.	`MocoRdbTypeVarchar`
`DATE`	Year, month, and day (without timezone).	`MocoRdbTypeDate`
`DATETIME`	Year, month, day, hours, minutes, and seconds (without timezone).	`MocoRdbTypeDateTime`
`TIMESTAMP`	Number of seconds since the Unix epoch.	`MocoRdbTypeTimestamp`
`BOOLEAN`	True or false value.	`MocoRdbTypeBoolean`
`BLOB`	Binary data.	`MocoRdbTypeBlob`

Table 4.1: List of data types in the relational data model

Note that the proposed data types do not necessarily conform to the SQL standard either. For example, the `DATETIME` type is not defined in the standard [36, p. 11].

4.4.4 Oracle SQL Data Model

We define the elements of an SQL data model as SQL DDL statements that instantiate a relational database. As summarized in subsection 2.5.3, there are three types of DDL statements used in the proposed transformation:

`CREATE TABLE`. In this statement, the name of the table, columns, primary keys, and unique constraints are defined.

`ALTER TABLE`. This statement is used to define foreign keys.

`CREATE TRIGGER`. The majority of OCL constraints are realized by defining a trigger in the database.

Since the SQL model exclusively acts as the output model of the transformation (i.e. it is only written, never read), it is sufficient to store raw SQL fragments instead of representing the SQL syntax in an object model. This approach also greatly reduces the implementation complexity.

For each statement type, our SQL data model defines a lightweight container for raw SQL text, as shown in figure 4.4.

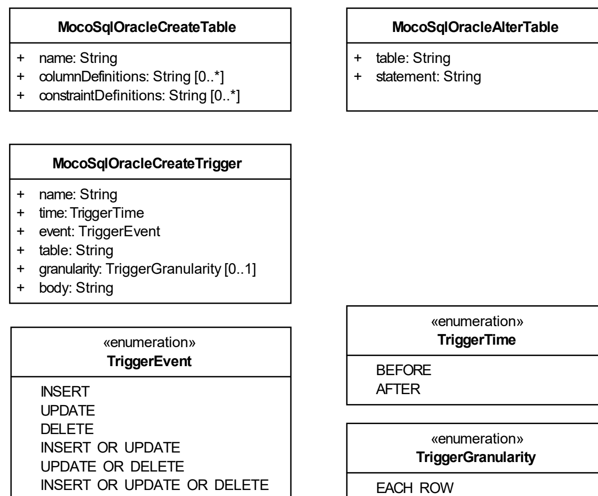

Figure 4.4: Class diagram of the Oracle SQL data model

Fragments of SQL text that shall be printed as-is when the model is exported to an executable SQL file are stored in properties:

- `columnDefinitions` and `constraintDefinitions` of `MocoSqlOracleCreateTable`,
- `statement` of `MocoSqlOracleAlterTable`,
- and `body` of `MocoSqlOracleCreateTrigger`.

4.4.5 OCL Data Model

The OCL data model needs to be able to represent the invariants and conditions derived during the various transformation phases. To limit complexity, we do not attempt to design a generic OCL model that would be able to represent every possible OCL constraint. Therefore, we represent each identified type of OCL constraint as a separate class with appropriate properties. The OCL data model is shown in figure 4.5.

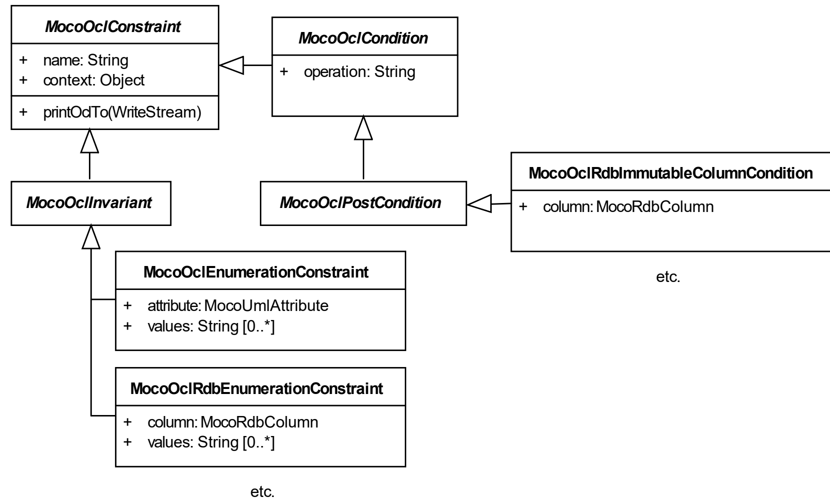

Figure 4.5: Class diagram of an excerpt from the OCL data model

With this design, no OCL code is generated when constructing elements of the OCL data model — instead, it is produced by the particular instance when calling the `printOclTo` method. Similarly, when the OCL data model is used as an input to a transformation rule, the rule can determine the type and meaning of a constraint element from its class, without needing to parse and understand the OCL source code.

The diagram in figure 4.5 omits some model classes that represent constraint types (as listed in section 2.5). They follow the same naming and generalization patterns as those shown in the diagram.

Note the distinction between a model and a data model — a model is a description of a software system using various notations (such as OntoUML, UML or a relational model), while the term *data model* in this section refers to the collection of data structures used to represent a model. In contrast to OntoUML, UML, and relational models, there is no standalone OCL model. Instead, the elements of the OCL data model appear in UML and relational models.

4.5 Transformation Rounds

As explained in section 2.5, the transformation proposed in [6] is not performed by transforming the OntoUML model directly to SQL, rather it is separated into three independent phases:

1. transformation of the OntoUML model to UML,
2. transformation of the UML model to a relational model,
3. and transformation of the relational model to an SQL script.

In addition, our approach is concerned with the integration of the transformation with OpenPonk, and with precise implementation details of the transformation. Thus, the three aforementioned phases alone are not sufficient. The complete implementation consists of seven transformation rounds in total:

1. First, the OntoUML model must be constructed using the data structures described in subsection 4.4.2. As specified by non-functional requirement 1, the source of the OntoUML model is an OpenPonk project. Therefore, before any further transformations

can take place, the model needs to be transformed from OpenPonk data structures to Moco data structures.

2. Next, the OntoUML model is transformed to a UML model, as described in subsection 2.5.1.

3. According to [6], the next phase is the transformation of the UML model to a relational model. However, certain UML constructs need to be refactored before they can be transformed to a relational model:

   - class attributes with upper multiplicity greater than one,
   - class attributes with data types referencing another class or an enumeration,
   - many–to–many associations,
   - and enumerations.

   Here, the term *refactoring* refers to the transformation of the aforementioned UML constructs to different UML constructs. For this reason, these refactorings should be performed prior to the transformation of the UML model to a relational model, rather than as part of the transformation to a relational model itself.

4. Then, the preprocessed UML model is transformed to a relational model, as described in subsection 2.5.2.

5. Next, the relational model is transformed to a sequence of SQL DDL statements, as explained in subsection 2.5.3.

6. Furthermore, to realize the functional requirement 2, the UML and relational models need to be displayed to the user. Therefore, an additional transformation takes place, which converts the UML model from Moco data structures to OpenPonk structures.

7. Finally, the relational model is converted to an OpenPonk model as well. However, since OpenPonk does not support relational data modeling, the result of this transformation round is a UML model loosely corresponding to the relational model.

With the design developed in section 4.2, each of the seven rounds can be implemented as a set of transformation rules, and executed by the transformation engine. The input and output of each round are listed in table 4.2 (abbreviations $\mathcal{OP}$ and $\mathcal{M}$ are used to distinguish OpenPonk and Moco data models respectively).

No.	Input	Output
1	OntoUML $^{\mathcal{OP}}$	OntoUML $^{\mathcal{M}}$
2	OntoUML $^{\mathcal{M}}$	UML $^{\mathcal{M}}$
3	UML $^{\mathcal{M}}$	UML preprocessed for RDB $^{\mathcal{M}}$
4	UML preprocessed for RDB $^{\mathcal{M}}$	relational model $^{\mathcal{M}}$
5	relational model $^{\mathcal{M}}$	SQL $^{\mathcal{M}}$
6	UML $^{\mathcal{M}}$	UML $^{\mathcal{OP}}$
7	relational model $^{\mathcal{M}}$	UML $^{\mathcal{OP}}$

Table 4.2: List of transformation rounds

4.6 User Interface

As described in requirements in section 3.4, the user executes the transformation of the OntoUML model from the OpenPonk user interface. Furthermore, the transformation process is interactive — the user chooses between options when multiple transformation approaches are possible. Finally, the results of the transformation are presented to the user.

4.6.1 Toolbar Menu

OpenPonk already implements its own version of the transformation of an OntoUML model to UML, which is executed by clicking on the appropriate item in the toolbar menu.

We use the same approach for consistency: a new submenu labeled "Moco" is added to distinguish OpenPonk's built-in features from our additions, and a "Transform to SQL" menu item is placed in the submenu. The transformation of the currently open OntoUML model is executed when the user clicks on the menu item. A wireframe representing the toolbar menu is shown in figure 4.6.

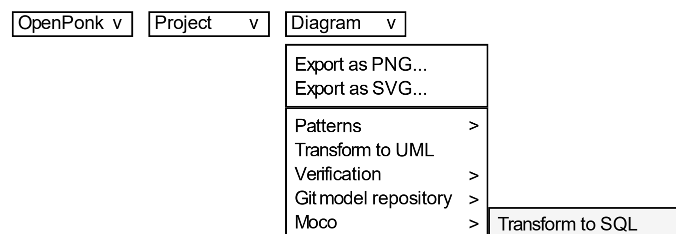

Figure 4.6: Wireframe of the OpenPonk window toolbar

4.6.2 Question Dialog

During the transformation, the transformation of a particular element may be realized in multiple ways. Then, user input is needed to determine which approach should be used. Each time such input is requested, a dialog window is presented with the question and buttons for selecting the response. A wireframe of this dialog is shown in figure 4.7.

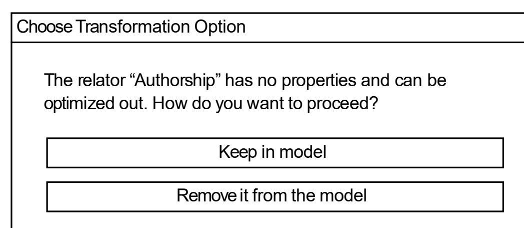

Figure 4.7: Wireframe of the transformation option choice dialog

Multiple question dialogs may appear during the transformation of a single model. To make the repeated transformation of a particular model more convenient for the user (as specified in functional requirement 4), the software remembers the answer to each question. Then, when executing the transformation again on the same model, the question dialog additionally contains a button that automatically answers all questions the same as during the previous execution.

4.6.3 Results Window

When the transformation is complete, the resulting UML and relational models, and the SQL script are displayed to the user. For showing the UML and relational diagrams, OpenPonk itself is used — either a new window displaying the diagrams is opened, or the diagrams are added to the currently opened project, depending on the user's preference.

The SQL script cannot be sensibly displayed in OpenPonk itself, therefore, a new window containing a text box with the SQL script is also opened. The window contains a "Save as" button as well, which opens the standard Pharo file selection dialog, and lets the user save the contents of the text box to a file. A wireframe of the window is shown in figure 4.8.

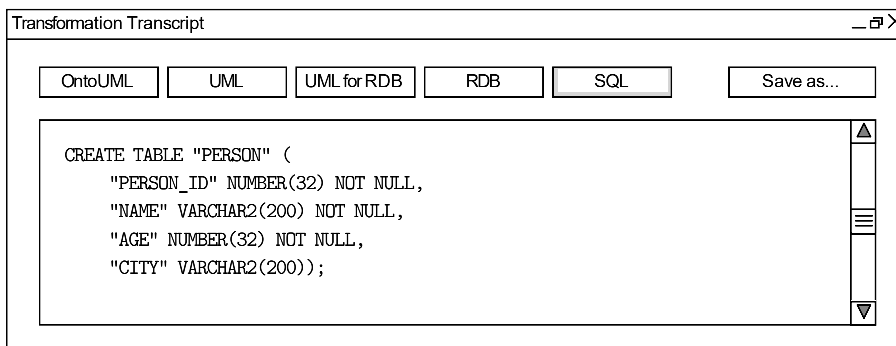

Figure 4.8: Wireframe of the transformation transcript window

In addition to showing the SQL script, the results window also shows plain-text dumps of the input OntoUML model and the intermediate UML and relational models. The user switches between them by clicking on the buttons above the text box.

4.6.4 Settings

As mentioned in subsection 4.6.3, the user can choose whether the transformation results should be displayed in a new OpenPonk window or if they should be added to the project with the original OntoUML model.

Pharo provides a settings framework, where applications can register class variables whose values are intended to be changed by users [37, p. 51]. It also has a built-in user interface where the user can view and modify all registered settings [37, p. 53].

The Pharo settings framework is used for Moco configuration. A screen capture of the user interface is provided in figure 4.9.

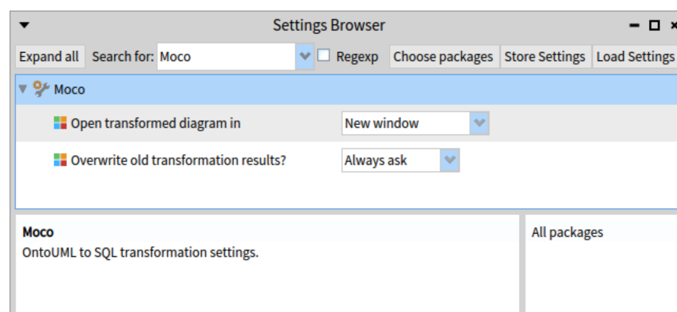

Figure 4.9: Settings Browser in Pharo

Chapter **5**

Implementation

This chapter details the implementation of the Moco library and individual transformation rules, and the integration with OpenPonk. It is structured as follows.

- In section 5.1, the API provided by the transformation engine is discussed.
- In section 5.2, the implementation of every transformation rule is detailed.
- In section 5.3, a discussion on the components that facilitate user interaction is provided.
- In section 5.4, the source code repository of the Moco library is discussed.
- In section 5.5, a build variant of OpenPonk with Moco integrated is introduced.

5.1 Transformation Engine

As discussed in section 4.2, the basis of the implementation consists of the transformation engine. The implementation of the transformation engine does not define any transformation by itself; it provides a base class for implementing transformation rules, the logic for executing a set of transformation rules, and an API for the transformation rules to manipulate the transformed model. The implementation of the transformation engine is located in package `Moco-Core`.

Throughout this section, we use the term *input element* to refer to an element in the input model, *output element* for an element in the output model, and *source element* for an input element linked with a particular output element.

5.1.1 Adding Elements and Linking Source with Output Elements

The `MocoTransformationEngine` provides API for adding output elements, removing output elements, and for creating and querying links between an output element and its source elements.

- A new element can be added to the output model by sending `addOutputElement:`. However, it is often useful to send `addOutputElement:withSource:`, which creates a link between the element in the input model that is the source of the new element and the output element, and this information can be used by other transformation rules.

- Additionally, if an object needs to be linked with an input element, but not added to the output model as a separate element, the transformation rule can send `atInput:putOutputElement:`. For example, when a UML association causes a table column to be generated, the transformation rule can use the message to record which association is the source of the column, yet the column remains contained in the table and is not added as a standalone element to the output model.
- To retrieve the source element of a particular element in the output model, `detectInputElement:`[1] is sent. Additionally, `detectInputElement:ofKind:` returns only a source element that is an instance of the given class, while `detectInputElement:ofKind:notGuarded:` also filters out elements with the given guard symbol set.

  The transformation engine technically does not prevent a single output element from having multiple source elements. The `detect*` messages return the first matching source element (or `nil` if none exists), to retrieve the entire collection of matching elements, each `detect*` message has its counterpart beginning with `select` (e.g. `selectInputElements:`).
- To retrieve all elements that were generated from a particular input element, `selectOutputElements:` is sent. Similarly to selecting source elements, `selectOutputElements:ofKind:` and `selectOutputElements:ofKind:notGuarded:` and their `detect*` variants also exist.
- The transformation engine allows the output model to be arbitrarily modified during the transformation, and this includes removing generated output elements. To safely remove an output element, `removeOutputElement:` is sent. This message also takes care of unlinking the output element from its source elements.

5.1.2 Selecting Elements and Managing Metadata

Selecting elements and metadata from the input or the output model is realized by sending messages to the container of the particular model itself (i.e. the `MocoModel` class). Therefore, a transformation rule first sends `input` or `output` to the transformation engine, then it uses the following API to interact with the model.

- To return a collection of all elements in the model, `elements` is sent.
- Since filtering by element kind and guards is common, shortcut methods are implemented as well. `selectElementsOfKind:` is used to select elements that are an instance of the given class, `selectElementsNotGuarded:` select elements that do not have the given guard symbol set, and `selectElementsOfKind:notGuarded:` is a combination of the previous two.

Guard symbols are used to prevent multiple transformations of a single element. Since a single transformation rule is usually executed more than once, it needs to be able to distinguish between elements it already transformed and elements that have not yet been transformed. This is easily achieved by setting a guard symbol (which usually equals the class name of the particular transformation rule) on each input element once it is processed by the transformation rule. Then, the rule always selects only input elements that do not have the guard symbol set.

- To add a guard symbol to an element, `guard:with:` is sent. To manually check whether a particular element has a guard symbol set, `check:isGuarded:` is used.

[1] The detect/select nomenclature is consistent with method names of standard Pharo collections.

- To add custom metadata to an element, `metaOf:at:put:` is sent. To retrieve the metadata, `metaOf:at:` is used. Metadata are implemented as a dictionary in the wrapped element, therefore the argument at the position of `at:` contains the key under which the data is stored.

- Since the transformation engine executes the transformation rules in a loop that ends once the output model stops changing, it needs to keep track of whether a rule changed it or not. This is done automatically when sending messages such as `addOutputElement:`, but needs to be done explicitly when changing the output model manually (e.g. adding a column to an already existing table). To indicate the model has changed, `setModified` is sent.

5.1.3 Constructing Models

An empty model is constructed by sending `new` to `MocoModel class`. When the object is used as an input model, elements are added to it by sending `addElement:`.

For transformations that consist of multiple rounds (e.g. the transformation of an OntoUML model to SQL), `addElements:` is provided, which accepts a `MocoModel` and adds all its elements to the current instance.

Furthermore, if certain metadata should be preserved between transformation rounds, `addElements:keepMeta:`, which accepts a collection of the metadata keys that should be preserved in the new model, is used.

5.1.4 Transformation Options

When an element can be transformed in multiple different ways, the user is asked to select which option should be used. During the execution of the transformation, the transformation rule sends a message `askChoice:` to the transformation engine with an instance of `MocoChoiceBuilder`, and the engine returns the option that the user chose.

The transformation rule creates an instance of a `MocoChoiceBuilder`. The text describing the transformation options is set by sending `question:` and individual options are added by sending `addChoice:withId:`. The first argument describes the choice, and the second argument contains a symbol that `askChoice:` returns if the user picks that particular choice.

The transformation engine displays a dialog, as designed in subsection 4.6.2, which lets the user pick the desired transformation option. Then, it records their option in a *choice log*. The code that manages the transformation engine can retrieve the choice log by sending `choiceLog`. Then, for example, when the same model is transformed the next time, it can pass it back to the engine by sending `choiceLog:`. The transformation engine will offer to answer all questions automatically based on the options recorded in the choice log.

5.1.5 Transformation Execution

A transformation is executed by creating an instance of `MocoTransformationEngine`, adding input elements to the input model, adding transformation rules, and sending the `transform` message to the transformation engine. Finally, the transformed model can be retrieved from the engine by sending `output`.

Transformation rules need to have a reference to the transformation engine that is executing them. Therefore, they are instantiated by sending `newWithEngine:` to their class.

An example of a code that creates an instance of the transformation engine, adds transformation rules, and executes the transformation is provided in listing 5.1.

Listing 5.1: Instantiating the transformation engine and executing a transformation

```
| engine rules |
rules := OrderedCollection with: RuleOne with: RuleTwo.
engine := MocoTransformationEngine new.
engine rules addAll: (rules collect: [ :each | each newWithEngine: engine ]).
engine input addElement: "...".
engine transform.
^ engine output
```

To avoid having to list transformation rules manually, Moco provides a `MocoTransformationRuleDynamicRepository`. It lists transformation rules within a specific package and tag, and adds them to the transformation engine. An example of the usage of the rule repository is provided in listing 5.2.

Listing 5.2: Usage of transformation rule repository

```
| engine repository |
engine := MocoTransformationEngine new.
repository := MocoTransformationRuleDynamicRepository new
                  rulesPackage: 'Moco-O2U';
                  rulesTag: 'Rule';
                  yourself.
repository injectTo: engine
```

5.1.6 Naming Utilities

As different types of models have different naming conventions (for example, a class model usually uses `PascalCase` formatting for class names, while a relational model typically uses `UPPER_SNAKE_CASE`), Moco provides utility methods for transforming between these conventions. These methods are implemented as class methods of `MocoNamingUtils`, and are listed in table 5.1.

Message	Example Input	Output
`toFirstLowerCase:`	`MyClass`	`myClass`
`toFirstUpperCase:`	`myclass`	`Myclass`
`toCamelCase:`	`Hello world`	`helloWorld`
`toPascalCase:`	`hello world`	`HelloWorld`
`toUpperSnakeCase:`	`MyClass`	`MY_CLASS`
`toShortName:`	`MyClass`	`mc`

Table 5.1: Methods for converting between naming conventions

Additionally, it is often necessary to prevent identifier collisions. Moco provides class `MocoUniqueNameGenerator` with class method `at:seed:`, which accepts a block at the `at:` position and a string at the `seed:` position. The block accepts a string containing

the proposed unique name, and returns `true` if such a name already exists in the context where it is being requested. First, the seed is proposed, then a number is appended to the seed and a loop starts that increments the number until the block returns `false`. Then, the method returns the final unique name.

5.2 Transformation Rules

The transformation of an OntoUML model to SQL is divided into multiple transformation rounds, as discussed in section 4.5. Each transformation round consists of the execution of a set of transformation rules by the transformation engine. The actual implementation of the transformation logic is located within the transformation rules, which are described in this section. A list of transformation rules and their priorities is provided in appendix B.

As required in subsection 4.2, the transformation rules are implemented to avoid infinite transformation loops by using guard symbols. Furthermore, they are designed so that the transformation result is always complete, and there is no possibility an element is left not transformed.

The implementation assumes that only valid models are transformed. When an invalid model is passed to a transformation rule, the behavior is undefined — an invalid output model may be produced or an exception may be raised. Thus, transformation rules do not validate the model (OpenPonk already implements OntoUML model validation), and are designed to handle invalid models on a best-effort basis (for example, a transformation of an overlapping or incomplete phase partition simply treats it as if it were complete and disjoint).

5.2.1 OpenPonk OntoUML to Moco OntoUML

In this subsection, we list the transformation rules that convert an OpenPonk OntoUML model to Moco data structures. The transformation rules are located in package `Moco-OpenPonk` under the `Rule` tag. The entire transformation is encapsulated by class `MocoOpenPonkOntoUmlTransformationRound` and is executed by invoking the `createMoco:` class method.

5.2.1.1 Classes

Rule 1.1 (`MocoOpenPonkOntoUmlClassTransformationRule`). Transforms classes without a stereotype from an OpenPonk OntoUML model to Moco OntoUML model. The class name and the `isAbstract` property are copied. For each attribute in the OpenPonk model, a corresponding attribute in the Moco model is created, with the name, visibility, multiplicity, immutability, `isStatic` property, and data type copied. Operations defined in the OpenPonk model are discarded.

In `MocoOpenPonkOntoUmlClassTransformationRule`, class methods `stereotypeSelector` and `modelClass` are defined, which control which classes are selected from the input model and what class is instantiated in the output model respectively. Then, all transformation rules for stereotyped classes are implemented as subclasses of rule 1.1, with the aforementioned class methods overridden to match the stereotype of the classes they are intended to transform. This approach is known as the Template Method design pattern [38, p. 325].

The transformation logic for stereotyped OntoUML classes is inherited from 1.1. A concise list of transformation rules for classes with stereotypes is provided in table 5.2.

Rule	Rule Class Name	Stereotype
1.2	`MocoOpenPonkOntoUmlKindTransformationRule`	«Kind»
1.3	`MocoOpenPonkOntoUmlSubKindTransformationRule`	«SubKind»
1.4	`MocoOpenPonkOntoUmlQuantityTransformationRule`	«Quantity»
1.5	`MocoOpenPonkOntoUmlRelatorTransformationRule`	«Relator»
1.6	`MocoOpenPonkOntoUmlCollectiveTransformationRule`	«Collective»
1.7	`MocoOpenPonkOntoUmlRoleTransformationRule`	«Role»
1.8	`MocoOpenPonkOntoUmlPhaseTransformationRule`	«Phase»
1.9	`MocoOpenPonkOntoUmlCategoryTransformationRule`	«Category»
1.10	`MocoOpenPonkOntoUmlRoleMixinTransformationRule`	«RoleMixin»
1.11	`MocoOpenPonkOntoUmlPhaseMixinTransformationRule`	«PhaseMixin»
1.12	`MocoOpenPonkOntoUmlMixinTransformationRule`	«Mixin»
1.13	`MocoOpenPonkOntoUmlQualityTransformationRule`	«Quality»
1.14	`MocoOpenPonkOntoUmlModeTransformationRule`	«Mode»

Table 5.2: OpenPonk OntoUML to Moco OntoUML class transformation rules

5.2.1.2 Associations

Rule 1.15 (`MocoOpenPonkOntoUmlAssociationTransformationRule`)**.** Transforms non-stereotyped associations from an OpenPonk OntoUML model to a Moco OntoUML model. The name of the association and the `isDerived` property are copied.

For each side of the association, the name, visibility, shareability, immutability, navigability, and multiplicity are copied. The source and target elements are set to the instances in the output model transformed from the elements in the input model.

Similarly to rule 1.1, class `MocoOpenPonkOntoUmlAssociationTransformationRule` also defines template methods `stereotypeSelector` and `modelClass`. In subclasses listed in table 5.3, only the aforementioned methods are overridden, while the transformation logic is inherited.

Rule	Rule Class Name	Stereotype
1.16	`MocoOpenPonkOntoUmlContainmentAssociationTrans-formationRule`	«Containment»
1.17	`MocoOpenPonkOntoUmlFormalAssociationTransforma-tionRule`	«Formal»
1.18	`MocoOpenPonkOntoUmlMaterialAssociationTransforma-tionRule`	«Material»
1.19	`MocoOpenPonkOntoUmlMediationAssociationTransfor-mationRule`	«Mediation»

Table 5.3: OpenPonk OntoUML to Moco OntoUML association transformation rules

Rule 1.20 (`MocoOpenPonkOntoUmlCharacterizationAssociationTransformationRule`)**.** Transforms OntoUML Characterization associations from an OpenPonk model to a Moco

model. Inherits the transformation logic from rule 1.15. Additionally, it reverses the direction of the association if the Quality or Mode is at the source side.

Furthermore, an abstract class `MocoOpenPonkOntoUmlPartWholeAssociationTransformationRule` that inherits from the transformation rule 1.15 exists. It serves as the base class for transformation rules that realize the transformation of OntoUML associations that represent part–whole relationships. In addition to the behavior of the transformation inherited from rule 1.15, the class also copies the essential and inseparable constraints. A list of concrete transformation rules that inherit from it is provided in table 5.4.

Rule	Rule Class Name	Stereotype
1.21	`MocoOpenPonkOntoUmlComponentOfAssociationTransformationRule`	«ComponentOf»
1.22	`MocoOpenPonkOntoUmlMemberOfAssociationTransformationRule`	«MemberOf»
1.23	`MocoOpenPonkOntoUmlSubCollectionOfAssociationTransformationRule`	«SubCollectionOf»
1.24	`MocoOpenPonkOntoUmlSubQuantityOfAssociationTransformationRule`	«SubQuantityOf»

Table 5.4: OpenPonk OntoUML to Moco OntoUML part–whole relationship transformation rules

5.2.1.3 Enumerations

Rule 1.25 (`MocoOpenPonkOntoUmlEnumerationTransformationRule`)**.** Transforms UML enumerations from an OpenPonk OntoUML model to a Moco model. The name of the enumeration and the enumeration literals are copied.

5.2.1.4 Generalizations

Rule 1.26 (`MocoOpenPonkOntoUmlGeneralizationTransformationRule`)**.** Transforms generalizations from an OpenPonk OntoUML model to a Moco model. For each input generalization, a generalization in the input model between the output classes transformed from the supertype and supertype of the generalization in the input model is created.

Rule 1.27 (`MocoOpenPonkOntoUmlGeneralizationSetTransformationRule`)**.** Transforms generalization sets from an OpenPonk OntoUML model to a Moco model. The name and the `isCovering` and `isDisjoint` properties are copied. For each member of the original generalization set, the corresponding generalization is selected from the output model and added to the transformed generalization set.

5.2.2 OntoUML to UML

The second transformation round consists of the transformation of an OntoUML model to UML. The transformation rules responsible for this round are located in package `Moco-O2U` under tag `Rule`.

The transformation is invoked by calling the `transform:withEngine:` class method on `MocoOntoUmlToUmlRound`. The caller puts a block at the `withEngine:` position. The block accepts a single argument containing the transformation engine and can be used for example to provide a choice log to the engine.

5.2.2.1 Classes

Rule 2.1 (`MocoOntoUmlClassTransformationRule`). Transforms non-stereotyped classes from an OntoUML model to a UML model. The name of the class and the `abstract` property are copied. For each attribute, the name, immutability, visibility, data type, multiplicity, and the `static` property are copied.

The class `MocoOntoUmlClassTransformationRule` defines class method `modelSelector` that specifies the stereotype of the classes that are selected by the transformation rule from the input model. The transformation logic is implemented in the `toUmlClass` method. Subclasses of rule 2.1 override these two template methods to customize the transformation behavior for specific OntoUML class stereotypes.

The priority of rule 2.1 is lower than the priority of more specific transformation rules, in order to ensure that stereotyped classes are transformed by the specific rules first. Since stereotyped classes ultimately inherit from `MocoOntoUmlClass`, rule 2.1 would pick them up otherwise.

A list of transformation rules that only override the `modelSelector` method, thus keeping the transformation logic the same as in rule 2.1, is provided in table 5.5.

Rule	Rule Class Name	Stereotype
2.2	`MocoOntoUmlKindTransformationRule`	«Kind»
2.3	`MocoOntoUmlSubKindTransformationRule`	«SubKind»
2.4	`MocoOntoUmlRelatorTransformationRule`	«Relator»
2.5	`MocoOntoUmlQuantityTransformationRule`	«Quantity»
2.6	`MocoOntoUmlCollectiveTransformationRule`	«Collective»
2.7	`MocoOntoUmlModeTransformationRule`	«Mode»
2.8	`MocoOntoUmlQualityTransformationRule`	«Quality»

Table 5.5: OntoUML to UML class transformation rules

Rule 2.9 (`MocoOntoUmlCategoryTransformationRule`). Transforms classes with the «Category» stereotype from an OntoUML model to UML. Inherits from rule 2.1, plus sets the class abstract.

Rule 2.10 (`MocoOntoUmlMixinTransformationRule`). Transforms classes stereotyped with «Mixin» from an OntoUML model to UML. Inherits from rule 2.1, plus sets the class abstract.

Transformation rules for the «RoleMixin» and «PhaseMixin» stereotype inherit from rule 2.10, as listed in table 5.6.

Rule	Rule Class Name	Stereotype
2.11	`MocoOntoUmlRoleMixinTransformationRule`	«RoleMixin»
2.12	`MocoOntoUmlPhaseMixinTransformationRule`	«PhaseMixin»

Table 5.6: OntoUML to UML mixin transformation rules

Rule 2.13 (`MocoOntoUmlRoleTransformationRule`). Transforms classes with the «Role» stereotype and generalization relationships that include those classes from an OntoUML

model to UML. Inherits from rule 2.1. The transformation of the Role class is the same as for a generic class.

After the transformation of classes is finished, all generalization relationships where the subtype is a Role and the supertype is not a Mixin or a RoleMixin are transformed to UML associations.

At the source side of the association is the supertype, the side is labeled `identity-Bearer`, is set immutable, and has a minimal and maximal multiplicity of `1`. At the target side of the association is the role with the label `role`, the minimal multiplicity is set to `0` and maximal multiplicity to `1`. The name of the association is the name of the Role class suffixed with `Role`.

Rule 2.14 (`MocoOntoUmlPhaseTransformationRule`)**.** Transforms classes stereotyped with «Phase» and associated phase partitions from an OntoUML model to UML. Inherits from rule 2.1. Transformation of the Phase class is the same as for a generic class.

After the transformation of classes is finished, all generalization sets that form a phase partition are transformed. A phase partition is detected when the generalization set contains a Phase subtype and the supertype has identity in the OntoUML model. For each phase partition, the user is asked to select how to transform the partition.

In case all phases in the phase partition have no attributes and are not part of any associations nor generalizations, the partition is considered as optimizable. If the user chooses to optimize the partition, all UML classes created from the phases are removed from the output model. Instead, a discriminator attribute with the same name as the phase partition is added to the supertype. The data type is `String` and the minimal and maximal multiplicity is `1`. Additionally, an OCL enumeration constraint (see listing 2.5) is generated that restricts the values of the discriminator column to the names of the Phase classes. The name of the constraint is constructed by joining `EX_`, the name of the supertype, the name of the phase partition, and `Condition` with an underscore.

The name of a phase partition is either the name of the generalization set or `Phase` if empty. In the case of multiple unnamed generalization sets, a counter is appended to the partition name beginning with the second partition.

When the user chooses to transform a phase partition using exclusive phase associations, the generalizations contained in the partition are replaced by associations. At the source side of each association is the supertype with the label `identityBearer`, with a minimal and maximal multiplicity of `1` and set as immutable. At the target side is the Phase class with the label `phase`, with minimal multiplicity of `0` and maximal multiplicity of `1`. The name of the association is set to the name of the Phase class. For the phase partition, an OCL exclusive associations constraint (see listing 2.4) is generated with a name constructed from `EX`, the name of the supertype, the name of the phase partition and `Condition`, all joined with an underscore.

When the user chooses to transform a phase partition using an abstract phase, a new abstract class is generated with the name constructed from the name of the supertype and the name of the partition. Then, an association between the supertype and the abstract class is generated. The supertype is at the source side of the association with the label `identityBearer`, a minimal and maximal multiplicity of `1`, and is set immutable. The abstract class is at the target side of the association with the label `condition` and minimal and maximal multiplicity of `1`. The name of the association is the same as the name of the phase partition. The generalization set realizing the phase partition is transformed to a new generalization set with the same name, where the generalizations are rewired to have the abstract class as their supertype.

5.2.2.2 Associations

Rule 2.15 (`MocoOntoUmlAssociationTransformationRule`). Transforms associations with either no stereotype or a «Formal» stereotype from an OntoUML model to a UML model. The name of the association and the `derived` property are copied. For each side, the label, visibility, multiplicity, navigability, and immutability are copied.

The implementation of rule 2.15 also uses the template method design pattern: class method `modelSelector` defines which kind of association is selected from the input model, and instance method `toUmlAssociation` realizes the transformation.

The priority of rule 2.15 is lower than the priority of more specific transformation rules, which ensures that stereotyped associations will be transformed by specific rules first. Otherwise, rule 2.15 would pick up the stereotyped associations, as they inherit from `MocoOntoUmlAssociation`.

Rule 2.16 (`MocoOntoUmlCharacterizationAssociationTransformationRule`). Transforms associations with the «Characterization» stereotype from an OntoUML model to UML. Inherits from rule 2.15. Additionally, if the association is connected to a Quality, both sides are set immutable. Otherwise, if a side contains a Mode, that side is set immutable.

Rule 2.17 (`MocoOntoUmlMediationAssociationTransformationRule`). Transforms associations with the «Mediation» stereotype from an OntoUML model to UML. Inherits from rule 2.15. When the association connect a Relator and a non-relator type, the non-relator side is set immutable. Additionally, if a Role is at the target side of the association, the direction is reversed.

Rule 2.18 (`MocoOntoUmlMaterialAssociationTransformationRule`). Transforms associations with the «Material» stereotype from an OntoUML model to UML. Within the class hierarchy, this rule inherits from rule 2.15. However, the `toUmlAssociation` method returns `nil`, which results in the association being removed from the transformed model.

Rule 2.19 (`MocoOntoUmlPartWholeAssociationTransformationRule`). Transforms associations with the «ComponentOf», «MemberOf», «SubCollectionOf», and «SubQuantityOf» stereotypes from an OntoUML model to UML. Inherits from rule 2.15. Additionally, when the essential constraint is set, the side of the part is set immutable. Similarly, when the inseparable constraint is set, the side of the whole is set immutable.

Rule 2.20 (`MocoOntoUmlContainmentAssociationTransformationRule`). Transforms associations with the «Containment» stereotype from an OntoUML model to UML. Inherits from rule 2.19. Additionally, always makes the side opposite to the Quantity immutable.

5.2.2.3 Enumerations

Rule 2.21 (`MocoOntoUmlEnumerationTransformationRule`). Transforms enumerations from an OntoUML model to a UML model. The entire enumeration element is copied.

5.2.2.4 Generalizations

Note that transformation rules 2.22 and 2.23, responsible for transforming generalizations, have a lower priority that rules 2.13 and 2.14, which transform roles, phases, and their associated generalization relationships. As a result, role generalizations and phase partitions are ignored by rules 2.22 and 2.23.

Rule 2.22 (MocoOntoUmlGeneralizationTransformationRule)**.** Transforms generalizations from an OntoUML model to a UML model. For all generalizations in the input model, a corresponding generalization is generated in the output model. Generalizations that have already been transformed to a different construct (e.g. an association in the case of roles) are not processed by this rule.

Rule 2.23 (MocoOntoUmlGeneralizationSetTransformationRule)**.** Transforms generalization sets from an OntoUML model to a UML model. For all generalization sets in the input model, a corresponding generalization set is generated in the output model containing the transformed members of the generalization set.

Additionally, if the generalization set is optimizable, the user is asked whether the set should be optimized. If so, the generalization set is discarded and all members and subtypes are removed from the output element. Instead, a discriminator attribute is generated in the class of the supertype. The name of the discriminator attribute is either the name of the generalization set or `discriminator` if empty, appended with a counter starting with the second unnamed generalization set. The discriminator attribute has the minimal and maximal multiplicity set to `1`, its data type is `String`, and is set immutable. An OCL enumeration constraint is also generated for the attribute (see listing 2.3), with the name consisting of `EN`, the name of the class and the name of the attribute, joined with an underscore. The allowed values depend on the complete and disjoint properties of the generalization set: when incomplete, the name of the superclass is included, when overlapping, all combinations of subtype names are included.

The generalization set is optimizable when the supertype has identity in the OntoUML model and all subtypes have no attributes, no associations, are not part of any other generalizations and are not a Role or a Phase.

5.2.2.5 Optimizations

In addition to the phase optimization and the generalization set optimization, which are included directly in the transformation rules 2.14 and 2.23 respectively, role and relator optimizations are implemented in separate transformation rules. These transformation rules have lower priority than the rest of the rules, and modify already transformed elements in the output model.

Rule 2.24 (MocoOntoUmlRoleOptimizationTransformationRule)**.** Selects roles that have no attributes, do not participate in more than one Mediation nor in more than one Material relations, do not participate in any other kind of association, have identity in the OntoUML, do not have any subtypes, and have no more than one supertype. For each selected role, the user is asked whether to optimize the role or to leave it unchanged.

The optimization removes the class generated from the role from the output model and rewires associations so they reference the identity bearer instead of the role and changes the minimal multiplicity of sides originally containing the role to `0`. Additionally, for associations where the role is at the source side[2], the label of the source side is set to the name of the role.

Rule 2.25 (MocoOntoUmlRelatorOptimizationTransformationRule)**.** Selects relators that have no attributes, participate in exactly two associations, and are not part of any generalization relationship. For each such relator, the user is asked whether it should be optimized or not.

When optimizing a relator, a new association directly connecting the mediated classes is generated, and the relator with the original associations is removed from the output

[2] Matches Mediation associations between a role and a relator.

model. The name of the association is set to the name of the relator. The source side of the generated association references the class from the first original association, and the multiplicity is calculated as the product of the multiplicity at the class side of the first original association and the multiplicity at the relator side of the second original association. Conversely, the target side of the generated association references the class from the second original association, and its multiplicity equals the product of the multiplicity at the relator side of the first original association and the multiplicity at the class side of the second original association. A product of two multiplicities is defined as $[a..b] \cdot [c..d] = [ac..bd]$.

5.2.3 Preprocessing UML for the Relational Model

As explained in subsection 4.5, the UML model needs to be refactored before it can be transformed to a relational model. The transformation rules that realize this preprocessing are located in package `Moco-U2D` under tag `Rule-Preprocess`.

Class method `preprocess:` of `MocoUmlToRdbRound` is used to execute this transformation round.

Rule 3.1 (`MocoUmlCopyTransformationRule`). Copies all elements from the input model to the output model.

After all elements are copied to the output model, the remainder of the transformation rules in this round work on the output model.

Rule 3.2 (`MocoUmlCopyFixReferencesTransformationRule`). Walks through all elements in the output model and changes references to elements in the input model to references to copies in the output model.

Rule 3.3 (`MocoUmlEnumerationTransformationRule`). Transforms UML enumerations to `String` attributes with an enumeration OCL constraint.

First, associations that reference an enumeration are collapsed into attributes: the association is removed and an attribute is generated in the class. The data type of the attribute is set to reference the enumeration, the name of the attribute is either the label at the enumeration side of the association, the name of the association or the name of the enumeration, depending on which of the aforementioned names is not empty. The multiplicity of the attribute is set to the multiplicity at the side of the enumeration in the original association.

Then, for all attributes that reference an enumeration, an OCL enumeration constraint is generated (see listing 3.1). Allowed values are the names of the literals in the referenced enumeration. The name of the OCL invariant is constructed from joining `EN`, the name of the class, and the name of the attribute with an underscore. Finally, the data type of the attribute is changed to `String`.

Transformation rules 3.4, 3.5 and 3.6 are listed in order of decreasing priority. The order is important, as later rules pick up the changes made in former rules.

Rule 3.4 (`MocoUmlNonNativeRdbTypeTransformationRule`). Transforms class attributes that reference another class within the UML model to associations. Each such attribute is removed from the output model and an association is generated instead.

At the source side is the original class, the minimal multiplicity is set to `0` and the maximal multiplicity to `*`. At the target side is the referenced class, the label of the side is set to the name of the original attribute, visibility, immutability, and multiplicity are copied from the original attribute as well. The name of the association is constructed by joining the name of the original class and the name of the attribute.

Rule 3.5 (`MocoUmlMultiValueAttributeTransformationRule`)**.** Replaces class attributes with maximal multiplicity greater than `1` by a new value class and an association.

For each such attribute, a new class is generated in the output model. The name of the class is constructed by joining the name of the original class and the name of the attribute with an underscore. The value class has a single attribute named `value`, with the data type of the original attribute and multiplicity copied from the original attribute as well.

Then, an association from the original class to the new value class is generated. At the source side is the original class with minimal and maximal multiplicity set to `1`. At the target side is the value class, with multiplicity, visibility, and name copied from the original attribute. The name of the association is constructed by joining the name of the original class and the name of the attribute.

When the original attribute is immutable, the `value` attribute and the target side of the generated association are set immutable as well.

Then, any OCL enumeration constraints on the original attribute are changed in the output model, so they restrict the new value class and the `value` attribute. Finally, the original attribute is removed from the class.

Rule 3.6 (`MocoUmlManyToManyAssociationTransformationRule`)**.** Decomposes many–to–many associations to two one–to–many associations. For each association where the maximal multiplicity at both sides is greater than `1`, an intermediating class is generated. The name of the class is constructed by joining the following parts with an underscore:

1. the label at the source side of the association or the name of the class at the source side, if the label is empty,
2. the name of the association or `To`, if the name is empty,
3. and the label at the target side of the association or the name of the class at the target side, if the label is empty.

Then, an association from the intermediating class to the source class from the original association is generated. The multiplicity at the side of the intermediating class is copied from the source side in the original association, and the multiplicity at the side of the source class is set to `1`. The label from the source side of the original association is copied to the side with the source class in the generated association.

Second, an association from the intermediating class to the target class from the original association is generated as well. The multiplicity at the side of the intermediating class is copied from the target side in the original association, and the multiplicity at the side of the target class is set to `1`. The label from the target side of the original association is copied to the side with the target class in the generated association.

When the source side in the original association is immutable, the side with the source class in the first association and the side with the intermediating class in the second association are set immutable. Similarly, when the target side in the original association is immutable, the side with the intermediating class in the first association and the side with the target class in the second association are set immutable.

Finally, the original many–to–many association is removed from the output model. Also, metadata indicating that the class was generated by this rule is attached to the element wrapping the intermediating class.

5.2.4 UML to a Relational Model

The fourth transformation round realizes the transformation of a UML model to a relational model. Transformation rules for this round are located in package `Moco-U2D`

under tag Rule. Class method transform: of MocoUmlToRdbRound is used to execute this transformation round.

Rule 4.1 (MocoUmlClassTransformationRule). Transforms UML classes to tables in a relational model. For each class, a new table is created with the same name as the original class.

An identity column is generated with the name constructed by suffixing the name of the table with _ID. The data type of the identity column is INTEGER, it is set non-nullable and a primary key is also generated for the identity column. In case an attribute with name ID exists in the original class, the identity column is not generated, and the existing ID attribute is used instead.

Each class attribute is transformed to a table column: the name is copied, the column is set nullable if the minimal multiplicity of the attribute is 0, otherwise it is set non-nullable. If the attribute is immutable, an additional OCL immutability constraint is generated (see listing 2.9). The data type of the column is resolved from the data type of the attribute according to the mapping in table 5.7.

UML Type	Relational Model Type
int, Integer, uint, long, short	INTEGER
String	VARCHAR(4000)
Date	DATE
DateTime	DATETIME
Timestamp	TIMESTAMP
bool, Boolean	BOOL
float, double, Decimal, Number	DOUBLE
List, Map, Array, Sequence	BLOB
mixed, Object, ByteArray	BLOB
char, Character, Symbol, byte	CHAR(1)

Table 5.7: UML to RDB data type mapping

Rule 4.2 (MocoUmlAssociationTransformationRule). Transforms UML associations to foreign keys in the relational model. For each association, a column is generated in the referencing table. The referencing side is determined as listed below, in list order.

1. When the maximal multiplicity of the source side of the association is greater than 1, the source side is the referencing side.
2. Conversely, the target side having maximal multiplicity greater than 1 makes it the referencing side.
3. When the source side is optional and the target side is mandatory, the source side is the referencing side.
4. Then, when the target side is optional while the source side is not, the target side is the referencing side.
5. When the target side is navigable, the source side is the referencing side, and vice versa.
6. Otherwise, the source side is the referencing side.

The name of the referencing column is either the label at the referenced side of the association (i.e. the side opposite of the referencing side), or the name of the primary key column if the label is empty. The column is set nullable if the minimal multiplicity of the referenced side of the association equals `0`, otherwise it is set non-nullable. A foreign key is generated on the referencing column.

When maximal multiplicities at both sides of the association are equal to `1`, a unique constraint is generated on the referencing column.

When the minimal or maximal multiplicity (or both) at the referencing side of the association is greater than `1` and not `*`, an OCL special multiplicity constraint is generated on the referencing column (see listing 2.7). Otherwise, if the minimal multiplicity at the referencing side is `1`, an OCL mandatory multiplicity constraint is generated (see listing 2.6).

When either side of the association is immutable, an OCL immutability constraint is generated on the referencing column (see listing 2.9). Additionally, when the referencing side is immutable, an OCL immutability postcondition is generated (see listing 2.10) as well.

Rule 4.3 (`MocoUmlGeneralizationTransformationRule`)**.** Transforms UML generalizations to foreign keys using the referencing tables approach. For each generalization, a foreign key constraint is generated on the primary key of the table representing the subclass, which references the primary key of the table representing the superclass.

Additionally, an OCL immutability constraint (see listing 2.9) and postcondition (see listing 2.10) are generated as well.

Rule 4.4 (`MocoUmlGeneralizationSetTransformationRule`)**.** Transforms UML generalization sets to discriminator columns. For each generalization set, a discriminator column is generated in the table representing the superclass. The name of the discriminator column is either constructed by joining `DISCR_` and the name of the generalization set with an underscore, in case the name of the set is empty, it defaults to `DISCRIMINATOR`. A numeric counter is appended to the name of the column starting with the second unnamed generalization set. The data type of the discriminator column is `VARCHAR` with a maximum length equal to the longest valid value of the column. The column is set non-nullable.

Additionally, an OCL generalization set constraint is generated (see listing 2.8). When the generalization set is incomplete, the name of the superclass is also a valid value of the discriminator column. When the generalization set is overlapping, all combinations of the names of the subtypes are valid values of the discriminator column.

The decomposition of many–to–many associations results in an additional intermediating class (see rule 3.6). The class is transformed to a database table by rule 4.1, which automatically generates an identity column. However, per [6, p. 81], such identity column must not be generated for classes mediating many–to–many associations. Rule 4.5 retroactively (due to having a lower priority than rule 4.1) removes the identity column from such classes.

Rule 4.5 (`MocoUmlMediatingClassOptimizationTransformationRule`)**.** Removes generated identity column from tables representing intermediating classes of decomposed many–to–many associations. In all tables that originated from classes whose metadata indicate that they were generated by rule 3.6, the identity column and the primary key are removed. Instead, a primary key is defined on the two columns included in the existing foreign key.

Rule 4.6 (`MocoUmlOclEnumerationConstraintTransformationRule`)**.** Transforms OCL enumeration constraints from a UML model to a relational model by changing the context

to the table generated from the class and the constrained attribute to the column generated from it.

Rule 4.7 (`MocoUmlOclExclusiveAssociationConditionTransformationRule`)**.** Transforms OCL exclusive associations constraints from a UML model to a relational model. Changes the context of the constraint to the table generated from the original class and replaces the original condition with a condition that checks for the existence of records that reference the constrained table.

5.2.5 Relational Model to Oracle SQL

The final step of the transformation of an OntoUML model to SQL is the conversion of a relational model to a sequence of SQL statements. The transformation rules that realize this round are located in package `Moco-D2S` under tag `Rule`.

This transformation round is executed by calling the `transform:` class method of `MocoRdbToSqlOracleRound`.

Rule 5.1 (`MocoRdbTableTransformationRule`)**.** Transforms tables in the relational model to `CREATE TABLE` SQL statements (see listing 2.11). When transforming column definitions, data types are mapped according to table 5.8.

Additionally, `ALTER TABLE` statements are generated to define the primary key and possible unique constraints (see listing 2.12).

Data type mapping in table 5.8 is constructed to match the meaning of relational model data types (see table 4.1) as accurately as possible. The mapping of `INTEGER` and `DOUBLE` is chosen to support the largest possible range of values.

Relational Model Type	Oracle Data Type
INTEGER	NUMBER(38)
VARCHAR(n)	VARCHAR2(n)
DATE	DATE
DATETIME	DATE
TIMESTAMP	TIMESTAMP(0)
BOOLEAN	CHAR(1 BYTE)
DOUBLE	FLOAT(126)
BLOB	BLOB
CHAR(n)	CHAR(n CHAR)

Table 5.8: Column data type mapping for Oracle Database

Rule 5.2 (`MocoRdbForeignKeyTransformationRule`)**.** Transforms foreign keys in the relational model to `ALTER TABLE` SQL statements for their realization in Oracle Database (see listing 2.13).

Finally, transformation rules that realize various OCL constraints in SQL are implemented. They generate SQL statements listed in subsection 2.5.3. The rules, including the type of the OCL constraint and a reference to the listing that contains the SQL statement that realizes the constraint, are listed in table 5.9.

Transformation Rule	
OCL constraint	**Realization in SQL**
Rule 5.3 (`MocoRdbOclEnumerationTransformationRule`).	
Enumeration (listing 2.5)	listing 2.25
Rule 5.4 (`MocoRdbOclExclusiveAssociationTransformationRule`).	
Exclusive association (listing 2.4)	listings 2.22, 2.23 and 2.24
Rule 5.5 (`MocoRdbOclGeneralizationSetTransformationRule`).	
Generalization set (listing 2.8)	listings 2.14, 2.15, 2.16 and 2.17
Rule 5.6 (`MocoRdbOclImmutableColumnTransformationRule`).	
Immutability (listing 2.9)	listing 2.26
Rule 5.7 (`MocoRdbOclImmutableAssociationDeleteTransformationRule`).	
Immutability (listing 2.10)	listing 2.27
Rule 5.8 (`MocoRdbOclMandatoryMultiplicityTransformationRule`).	
Mandatory multiplicity (listing 2.6)	listings 2.18 and 2.19
Rule 5.9 (`MocoRdbOclSpecialMultiplicityTransformationRule`).	
Special multiplicity (listing 2.7)	listings 2.20 and 2.21

Table 5.9: RDB to SQL transformation rules for OCL constraints

5.2.6 Moco UML to OpenPonk UML

In order to visualize the transformed UML model, it must be converted from Moco data structures back to an OpenPonk model. The transformation rules that realize this transformation round are located in package `Moco-OpenPonk` under tag `Rule-Uml`.

Rule 6.1 (`MocoOpenPonkUmlClassTransformationRule`)**.** Converts an UML class from a Moco model to an OpenPonk model. The name of the class, the `abstract` property, and attributes (name, data type, visibility, multiplicity, immutability, the `static` property) are copied.

Rule 6.2 (`MocoOpenPonkUmlAssociationTransformationRule`)**.** Converts an UML association from a Moco model to an OpenPonk model. The name of the association and the `derived` property are copied. The label, visibility, multiplicity, immutability, aggregation, and navigation of both sides are copied.

Rule 6.3 (`MocoOpenPonkUmlEnumerationTransformationRule`)**.** Converts an UML enumeration from a Moco model to an OpenPonk model. The name of the enumeration and the list of literals are copied.

Rule 6.4 (`MocoOpenPonkUmlGeneralizationTransformationRule`)**.** Converts an UML generalization from a Moco model to an OpenPonk model. For each generalization, a corresponding generalization is generated in the OpenPonk model.

Rule 6.5 (`MocoOpenPonkUmlGeneralizationSetTransformationRule`)**.** Converts an UML generalization set from a Moco model to an OpenPonk model. The name of the generalization set and the disjoint and complete constraints are copied. For each member of the original generalization set, the corresponding generalization in the output model is placed in the transformed generalization set.

Rule 6.6 (`MocoOpenPonkUmlOclTransformationRule`)**.** Writes out OCL constraints as class comments in an OpenPonk model. For each constraint, a comment with the OCL code is added to the class referenced by the context of the constraint.

5.2.7 Moco RDB to OpenPonk UML

Transformation rules that perform the conversion of a relational model to an OpenPonk UML model are located in package `Moco-OpenPonk` under tag `Rule-Rdb`. These rules do not perform the reverse of the transformation of a UML model to a relational model, instead, they work around the missing support for relational data modeling in OpenPonk by creating UML classes that look similar to tables in a relational data diagram.

Rule 7.1 (`MocoOpenPonkRdbTableTransformationRule`)**.** Converts a database table to a UML class in an OpenPonk model. The name of the generated class is set to the name of the table. Each column of the table is transformed to an attribute of the class, and the primary key, foreign keys, and unique constraints are transformed to operations in the class.

Rule 7.2 (`MocoOpenPonkRdbForeignKeyTransformationRule`)**.** Transforms foreign keys to UML associations in an OpenPonk model. Each foreign key is transformed into a navigable association originating at the class representing the table that contains the foreign key column and targeting the class representing the referenced table. The name of the association is set to the plain-text definition of the foreign key.

The maximal multiplicity of the target side is always `1`, the minimal multiplicity is either `0` or `1` when the column is nullable or non-nullable respectively. The multiplicity of the source side is by default `0..*` and is further refined if any mandatory multiplicity or special multiplicity constraints are detected.

Rule 7.3 (`MocoOpenPonkRdbOclTransformationRule`)**.** Writes out OCL constraints as class comments in an OpenPonk model. For each constraint, a comment with the OCL code is added to the class representing the table referenced by the context of the constraint.

5.3 User Interface

In this section, the implementation details of the components that are responsible for interaction with the user are discussed.

5.3.1 Transformation Transcript

To display the results of the transformation, the client code can open a transcript window provided by Moco. As illustrated in figure 4.8, the window contains a plain-text dump of one or more models. The window is implemented using the Spec UI framework in class `MocoTranscriptPresenter`. An example of how to display the transformation transcript window is provided in listing 5.3.

Listing 5.3: Displaying transformation transcript

```
| window transcript |
window := (MocoSpApplication instance) new: MocoTranscriptPresenter.
transcript := String new writeStream.
MocoModelPrinter print: model to: transcript.
window addSection: (transcript contents) withLabel: 'Model';
       open
```

The transformation engine allows any object to be used as a model element. Therefore, it is the responsibility of the caller to construct the plain-text representation of the model. However, if all elements of the model understand message `printTo:` (with a `WriteStream`), the `MocoModelPrinter` class can be used to print all elements in a `MocoModel` to the given stream (as shown in listing 5.3).

5.3.2 Toolbar Item

As designed in subsection 4.6.1, the transformation of an OntoUML model to SQL is invoked via a toolbar menu within OpenPonk. OpenPonk provides an extension mechanism, where third-party code can annotate a class method and provide custom menu items. For Moco, this is realized in the `toolbarMenu:` method of `MocoToolbarUi`.

The `toolbarMenu:` method creates a "Moco" submenu with items "Transform to SQL", which executes the transformation command, and "Open Playground", which opens the Moco Playground.

5.3.3 Transformation Command

The entire transformation of an OntoUML model to SQL is performed by calling the `MocoOpenPonkRunTransformationsCommand` class. The caller passes a reference to the OpenPonk instance that contains the input OntoUML model, and then sends the `execute` message.

The logic for loading transformation rules, executing transformation rounds, and displaying the results is contained within the command class. This approach is commonly known as the Command design pattern [38, p. 233].

First, the command retrieves the currently active model from the OpenPonk instance. Then, it executes transformation rounds to ultimately transform the OntoUML to SQL. After that, a transcript is prepared containing the resulting model of each round, and the UML and RDB models are transformed back to OpenPonk format.

The command uses the Settings framework, as described in subsection 4.6.4, to determine how the results should be presented. In order to remain consistent with the behavior of the built-in transformation of an OntoUML model to UML, a new OpenPonk instance with the transformed models is opened by default. Alternatively, the user can configure Moco to add the results of the transformation to the project of the original OntoUML model.

Finally, the transformation transcript window is also opened. A reference to the transcript window and the input OntoUML model is saved in a class variable. When settings indicate that the results should be added to the original OpenPonk project and it already contains the results of a previous transformation, the user is asked whether the new results should replace the previous ones. If so, the reference to the transcript window is used to replace its contents as well. To prevent memory leaks, once the transcript window is closed, the reference to it and to the source OntoUML model are deleted from the class variable, and the objects are free to be reclaimed by the garbage collector.

The command class also stores the choice log (see subsection 5.1.4) for the OntoUML model. During subsequent executions of the transformation command on the same OntoUML, the log is restored in the transformation engine. Again, to prevent memory leaks, the log and the reference to the OntoUML model are deleted when the corresponding OpenPonk instance is closed.

5.3.4 Moco Playground

Pharo contains a Playground: a console for ad-hoc execution of code. Moco Playground prefills the code required to load an OntoUML model and to execute the transformation from OntoUML to SQL. This feature can be used for debugging, transformation customization, and easy inspection of the raw data models.

When the user chooses the “Open Playground” item in the toolbar menu, a reference to the currently active OntoUML model is stored as global state in a class variable. This is necessary so the user can easily retrieve the model, as shown in listing 5.4.

Listing 5.4: Prefilled code in Moco Playground

```
| openPonkModel ontoModel umlModel dbModel sqlModel |
openPonkModel := MocoOpenPonkModelStorage ontoUmlModel.

ontoModel := MocoOpenPonkOntoUmlTransformationRound createMoco: openPonkModel.
umlModel := MocoOntoUmlToUmlRound transform: ontoModel.
dbModel := MocoUmlToRdbRound transform: umlModel.
sqlModel := MocoRdbToSqlOracleRound transform: (dbModel second).
```

5.4 Code Repository

The Moco library, including transformation rules, is versioned in a Git repository. The repository also contains the baseline specification for Metacello, which is used to install the library into an existing OpenPonk image. A list of packages included in the repository is provided in table 5.10.

Package Name	Purpose
`BaselineOfMoco`	Metacello baseline definition.
`Moco-Core`	Implementation of the transformation engine.
`Moco-OpenPonk`	Integration with OpenPonk, transformation rules for conversion between Moco and OpenPonk data structures.
`Moco-OpenPonk-Tests`	Functional tests for rules in `Moco-OpenPonk`.
`Moco-OntoUml`	Data model for OntoUML.
`Moco-O2U`	OntoUML to UML transformation rules.
`Moco-O2U-Tests`	Functional tests for rules in `Moco-O2U`.
`Moco-Uml`	Data model for UML.
`Moco-U2D`	UML to relational model transformation rules.
`Moco-U2D-Tests`	Functional tests for rules in `Moco-U2D`.
`Moco-Rdb`	Relational data model.
`Moco-D2S`	Relational model to SQL transformation rules.
`Moco-D2S-Tests`	Functional tests for rules in `Moco-D2S`.

continued on next page

Table 5.10: Package list of Moco

Package Name	Purpose
Moco-SqlOracle	Oracle SQL data model.
Moco-Ocl	OCL data model.

Table 5.10: Package list of Moco

In listing 5.5, a code snippet that installs Moco from a local copy of the repository (specified by variable `mocoDir`) is provided.

Listing 5.5: Installation of Moco using Metacello

```
Metacello new baseline: 'Moco';
	repository: 'tonel://' , mocoDir , '/src';
	load.
```

5.5 OpenPonk Build

A separate repository with a build script that packages OpenPonk, the Moco library, and the Pharo runtime is provided for the convenience of end users. The build script is executed in a GitLab continuous integration pipeline, which publishes prebuilt archives — the user downloads and extracts the archive, runs a launcher script, and can immediately start using OpenPonk and Moco. Archives are provided for platforms Windows, Linux, macOS, and macOS ARM.

Instead of building the entire Pharo image from scratch, release archives of OpenPonk Class Editor nightly version `76c3099` obtained from the official website[3] are included in the repository. The archives contain the OpenPonk image, the launcher script and the Pharo runtime. The build script installs the Moco library in the OpenPonk image and repackages the release archive. Because the OpenPonk version is pinned, inadvertent breakage due to upstream changes is avoided.

5.5.1 Project Saving Workaround

OpenPonk is not able to save projects that contain class attributes with unknown types[4]. An unknown type is any type that is not a UML primitive type, nor a class or enumeration defined in the model. For example, an attribute with type `Date` triggers this bug, and an exception is raised instead of saving the project.

This severely limits the usability of OpenPonk for precise conceptual modeling, as data types of attributes need to be preserved. To work around this issue, Moco provides a workaround — before saving, unknown data types are replaced with instances of `OPUMLPrimitiveType` with `name` property set to the original type name.

This workaround is implemented in class `MocoOpenPonkSaveWorkaround`. Furthermore, the build script that packages OpenPonk with Moco recompiles OpenPonk's “Save” and “Save As” commands so they invoke the workaround.

After the workaround is applied, saving and opening projects works correctly. Moco reads only the name of the attribute type, therefore it is not affected by an attribute wrongly claiming it contains a UML primitive type.

[3] OpenPonk release archives can be downloaded from `https://openponk.org/#download`.
[4] A bug report is available at `https://github.com/OpenPonk/class-editor/issues/21`.

Chapter 6

Evaluation

In this chapter, the testing of the implementation and the evaluation of the results of a transformation of an OntoUML by the implemented library are discussed. The chapter is structured as follows.

- In section 6.1, automated functional testing, which validates the implemented transformation logic, is described.
- In section 6.2, the differences in names generated during the transformation between the transformation proposed in [6] and the implementation are listed.
- In section 6.3, the implementation is evaluated using an example OntoUML model, and the conformance to a reference transformation result from [6] is discussed.

6.1 Functional Testing

The resulting implementation is validated by automated functional tests, which cover the transformation rules. The aim is to ensure each transformation rule produces a correct output (as defined in section 2.5), not necessarily to test their internal logic. For this reason, each test prepares an input model, then executes a single transformation rule, and asserts that the output model contains the expected content.

Since the tests are automated and designed to run without user interaction, they use a modified implementation of the transformation engine. It does not display question dialogs when the transformation rule contains multiple transformation options. Instead, the answer is provided by the test case. Additionally, it allows the test case to inspect the options provided by the transformation rules, so it can assert whether the rule correctly detected the possible transformation options.

Functional tests are implemented using the SUnit framework, which is included in standard Pharo images. All transformation rule tests inherit from class `MocoTransformationRuleTestCase`, which prepares an empty instance of the testing implementation of the transformation engine for each test case.

Every transformation rule has an accompanying test class. The test class has the same name as the tested class plus a `Test` suffix. Test classes are located in separate packages: the test packages have the same name as the tested packages plus a `-Tests` suffix. For example, a test class for the transformation rule `MocoUmlClassTransformationRule` from package `Moco-U2D` is named `MocoUmlClassTransformationRuleTest` and is located in package `Moco-U2D-Tests`.

In the case of transformation rules with inherited behavior, the test class only covers the changed parts, inherited parts are tested by the test class of the rule's superclass. For example, since the transformation of an OntoUML Kind from an OpenPonk model to the Moco model inherits from a generic rule for class transformation, the test class for the Kind transformation only ensures the correct stereotype is applied, and the transformation of class attributes, etc. is covered in tests of the generic class transformation rule.

Depending on the particular transformation rule, a single test class may define multiple test cases: For example, the rule responsible for the transformation of UML associations to a relational model needs to support one–to–one and one–to–many associations, and various multiplicity and immutability constraints. The test class creates a separate input model for each case and verifies all instances are transformed correctly.

As follows from the description of the implementation of transformation rules (see section 5.2), the order of execution is important, since some rules may depend on transformations executed in earlier rules. For this reason, additional test classes are provided that assert the correct order of the transformation rules. Transformation rules express their order by returning a number indicating their priority. The assertions do not enforce specific priority values, only the relative order is checked.

Packages with test classes are included in the Moco source repository (see section 5.4), as well as in the Metacello project. Furthermore, each package contains a *test suite* definition: a list of test classes. Instead of executing each individual test class, the entire suite is executed. The list of test suites is provided in table 6.1.

Test Suite	**Number of Test Cases**
`MocoOpenPonkTestSuite`	64
`MocoO2UTestSuite`	67
`MocoU2DTestSuite`	44
`MocoD2STestSuite`	11
Total	186

Table 6.1: Functional test suites

Finally, the source repository contains pipeline definitions for GitHub and GitLab code hosting services, which runs all test suites after each push to the repository. This way, the implementation is automatically tested after change, which reduces the risk of introducing bugs.

6.2 Differences in Generated Names

The transformation principles were summarized in section 2.5 and revised in section 3.3. During implementation and testing, further adjustments were made to generated names of various model elements. Additional name components, more specific names, and entirely new labels were introduced to resolve potential name conflicts and to improve traceability (i.e. to better understand the origins of generated elements).

In this section, we list the changes made to the names of generated model elements in comparison with the transformation proposed in [6].

6.2.1 OntoUML to UML

1. When transforming the generalization relationship between a Role and its bearer, the

generated association is named after the role with the `Role` suffix. Additionally, the side of the bearer is labeled `identityBearer` and the side of the role is labeled `role`.

2. When realizing a phase partition by exclusive phase associations, an OCL condition is generated. The name of the OCL condition was originally constructed by concatenating `EX`, the name of the type containing the partition, and `Condition` with an underscore. Our implementation adds the name of the generalization set after the name of the type. If empty, `Phase` is used instead. Furthermore, each association is named after the particular phase, and the side of the bearer is labeled `identity-Bearer`, while the side of the phase is labeled `phase`.

3. In the transformation of a phase partition by an abstract phase, the name of the abstract phase class is set to the name of the generalization set. If empty, `Phase` is used instead. The name of the association between the type containing the partition and the abstract phase is set to the name of the generalization set as well, and the side of the abstract phase is labeled `condition`, while the side of the type is labeled `identityBearer`.

4. During the optimization of a phase partition, a discriminator attribute is generated. The name of the discriminator attribute is set to the name of the generalization set or `phase` if the name is empty. Additionally, the name of the OCL enumeration constraint is constructed by joining `EX`, the name of the type containing the partition, the name of the generalization set, and `Condition` with an underscore.

5. In the optimization of generalization sets, the names of the discriminator attribute and the enumeration constraint already contain the name of the generalization set. In case the name is empty, `discriminator` is used instead.

6.2.2 UML to a Relational Model

1. When decomposing a many–to–many association into two one–to–many association, an intermediating class is generated. By default, the name of the class is constructed by concatenating the name of the source class and the name of the target class with `_To_`. In our implementation, if the source or target are labeled, their label is used instead of the name of the class. Additionally, if the name of the many–to–many association is not empty, it is used instead of the `To` infix.

2. When generating unique and foreign keys, the name of the enclosing table is included in the name of the constraint[1].

3. During the transformation of a generalization set, a discriminator column is generated in the supertype table. By default, the name of the column is `DISCRIMINATOR`. Additionally, if the generalization set is named, the name of the column is set to the name of the generalization set prefixed by `DISCR_`. Furthermore, the name of the OCL generalization set constraint is constructed by joining `GS`, the name of the supertype class, and `Type` with an underscore. However, when the name of the generalization set is not empty, it is used instead of the `Type` suffix.

6.3 Transformation of the Example Model

Functional tests in section 6.1 validate the transformation of small isolated groups of elements. To test the implementation against a larger and more interconnected model,

[1] At least in Oracle Database, constraint names must be unique within the entire schema.

we use the example OntoUML model of a library information system from [6]. The author of [6] also provides the results of the proposed transformation of the OntoUML model to UML and the relational model, which we compare against the outputs of our implementation (in this section, they are referred to as the *reference results*). The input OntoUML model is provided in appendix A.1.

6.3.1 OntoUML to UML

The output of the OntoUML to UML transformation round matches the reference transformation result, with exceptions listed here. The class diagram of the UML model generated by the implementation is provided in appendix A.2.

1. Generated names of certain elements are different, as discussed in subsection 6.2.1.
2. The implemented optimization of role `PublishedEdition` produces a minimal multiplicity of `0` at the side of the associated relator, while in the reference result, the side is mandatory (as shown in figure 6.1). As discussed in subsection 2.5.1, the minimal multiplicity at the side of the relator must be `0`, otherwise individuals that do not instantiate the role could not exist. Therefore, the output of the implementation is correct.

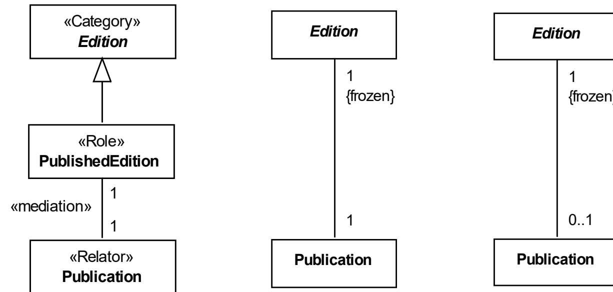

Figure 6.1: OntoUML model with a Role (left), the proposed optimization (middle), and the result of the implemented optimization (right)

6.3.2 UML to a Relational Model

The transformation from the UML model to a relational model is affected by differences from the preceding transformation round. Otherwise, it matches the reference result, again with the exceptions listed here.

1. Further differences in generated names of elements are introduced, as discussed in subsection 6.2.2.
2. Some columns have a different data type, due to a different data type mapping, as listed in tables 4.1 and 5.7.
3. In the reference result, table `BOOK_EDITION` is missing column `EDITION`, which originates in the `edition` attribute of class `BookEdition`. The implementation correctly generates this column.
4. It is not possible to perform the optimization of the generalization set containing classes `RegisteredReader` and `RegisteredLegalEntity`. First, the optimization is performed in the OntoUML to UML transformation round and is not repeated here.

Second, this particular generalization set cannot be optimized automatically, as the subclasses take part in associations (see transformation rule 2.23). The difference between the transformation applied manually in the reference result and the output of the implementation is shown in figure 6.2.

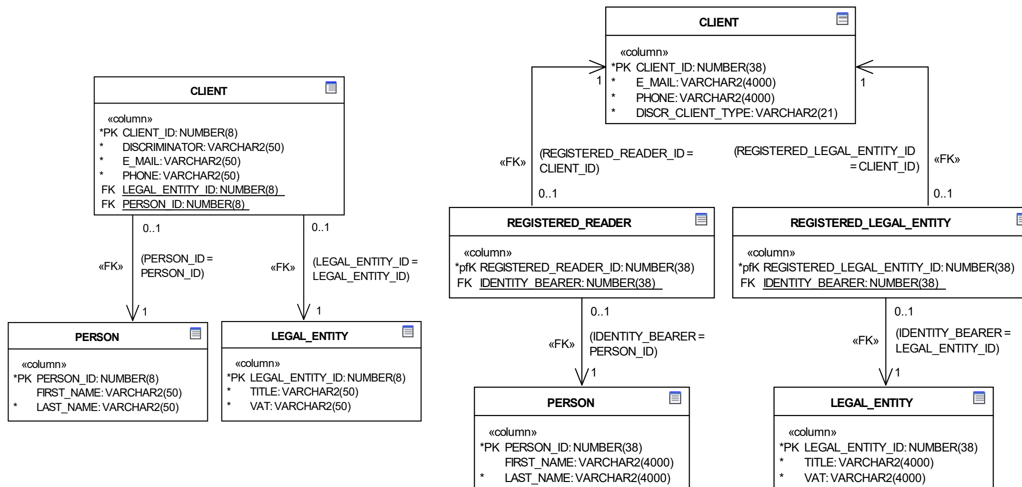

Figure 6.2: Manually applied generalization set optimization (left) and result of the implemented transformation (right)

The relational model diagram generated by the implementation of this transformation round is provided in appendix A.3.

6.3.3 Realization of Constraints and Resulting SQL Model

Due to a difference in the mandatoriness of the relation between `Edition` and `Publication`, and in the realization of roles `RegisteredReader` and `RegisteredLegalEntity`, the number of derived OCL constraints and the resulting SQL model is different as well.

In table 6.2, differences between the reference model and the results of the implemented transformation are summarized. A negative difference indicates an element is present in the reference model but not in the implementation output, and vice versa. Elements in the RDB model are typeset without indentation; then, SQL statements generated from an element are listed below it and are indented.

Type of Element	Diff.
Non-Mandatory Side at `Publication`	
mandatory multiplicity OCL	−1
↪ `CREATE TRIGGER BEFORE`	−1
↪ `CREATE TRIGGER AFTER`	−1
Non-Eliminated Roles `RegisteredReader` and `RegisteredLegalEntity`	
foreign keys	−2
↪ `ADD FOREIGN KEY`	−2

continued on next page

Table 6.2: Difference in the number of generated elements

Type of Element	Diff.
immutable column OCL	−1
↪ `CREATE TRIGGER BEFORE`	−1
tables	+2
↪ `CREATE TABLE`	+2
foreign keys	+4
↪ `ADD FOREIGN KEY`	+4
immutable column OCL	+4
↪ `CREATE TRIGGER BEFORE`	+4
immutable association delete OCL	+2
↪ `CREATE TRIGGER BEFORE`	+2
generalization set OCL	+1
↪ `CREATE TRIGGER BEFORE`	+7

Table 6.2: Difference in the number of generated elements

When comparing the total count of element types in the output of the implemented transformation (see appendix A) with the reference model and accounting for the differences listed in table 6.2, the same number of elements is obtained[2].

6.3.4 Summary

The implemented library successfully transforms the example OntoUML model to SQL. The resulting script can immediately be used to instantiate a database in Oracle RDBMS.

There are a number of differences between the reference model and the output of the implemented transformation. All of them are described in sections 6.2 and 6.3, and no other unexpected differences exist.

[2]In [6, p. 210], 7 generalization set OCL constraints are recorded. However, considering the RDB model in [6, p. 211], at most 6 such constraints may exist. Additionally, in [6, p. 212], 3 `CHECK` constraints are indicated. However, only enumeration OCL constraints may be realized as `CHECK` constraints [6, p. 169], and the model in [6, p. 210] contains only 2 such constraints.

Chapter **7**

Conclusion

This thesis focused on the implementation of the transformation of an OntoUML model to SQL. The implementation was realized in the Pharo programming language as an extension to the OpenPonk modeling tool.

- In *Chapter 1: Introduction*, an introduction and motivation for modeling of software systems was provided, and goals of the thesis were established.
- In *Chapter 2: State of the Art*, technologies and theories related to the topic of the thesis were discussed. First, the basics of conceptual modeling were explained, then Unified Modeling Language, Unified Foundational Ontology, and OntoUML were discussed. The relational model, Entity–Relationship Model, Data Modeling Profile for UML, SQL, and Oracle Database were introduced. Specification of constraints in the OCL language was described. Finally, the transformation of an OntoUML model to SQL proposed in [6] was summarized.
- In *Chapter 3: Analysis*, prerequisites for the implementation were discussed. The OpenPonk modeling tool was introduced; its data model was described and its built-in transformation of an OntoUML model to UML was compared to the transformation proposed in [6]. Then, relevant concepts of the Pharo programming language were introduced: its distinguishing features, and its approach to source code versioning, package management, and GUI programming. Furthermore, the transformation approach in [6] was revised for the current version of OntoUML and extended with support for UML enumerations. Finally, software requirements were specified.
- In *Chapter 4: Design*, a new rule-based approach for the implementation of model transformations was designed. Then, Moco library containing the implementation was introduced. Data models for the representation of OntoUML, UML, relational, SQL and OCL models, and the user interface were designed. Finally, the transformation approach was further refined into seven transformation rounds.
- In *Chapter 5: Implementation*, the structure of the implementation was described, and an overview of the API for building transformation rules was provided. Then, the implementation details of transformation rules realizing the transformation of an OntoUML model to SQL were explained. Finally, the implementation of the user interface and the integration into OpenPonk were discussed.
- In *Chapter 6: Evaluation*, functional tests that validate each transformation rule were developed. Furthermore, changes to the originally proposed transformation made dur-

ing the implementation were explained. Finally, the implementation was evaluated using an example OntoUML model.

- In *Chapter 7: Conclusion*, the results of the thesis are summarized.

7.1 Summary

In this thesis, we successfully analyzed the transformation of an OntoUML model to SQL proposed in [6], then designed and developed its implementation. During evaluation, the validity of the implementation was successfully demonstrated on a complex OntoUML model and further verified by functional tests.

7.2 Contributions of this Thesis

The contributions made in this thesis are as follows.

1. A transformation framework for the implementation of arbitrary model transformations was created.
2. The transformation of an OntoUML model to its realization in Oracle Database was implemented.
3. A ready-to-use distribution of the OpenPonk modeling tool, which automates the transformation of an OntoUML model to SQL, was created.

Bibliography

1. MELLOR, Stephen J.; CLARK, Anthony N.; FUTAGAMI, Takao. Model-Driven Development. *IEEE Software*. 2003, vol. 20, no. 5, p. 14.
2. BORK, Dominik; KARAGIANNIS, Dimitris; PITTL, Benedikt. A Survey of Modeling Language Specification Techniques. *Information Systems*. 2020, vol. 87, p. 101425. ISSN 03064379. Available from DOI: `10.1016/j.is.2019.101425`.
3. GUIZZARDI, Giancarlo; WAGNER, Gerd; ALMEIDA, João Paulo Andrade; GUIZZARDI, Renata S.S. Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story. *Applied Ontology*. 2015, vol. 10, no. 3-4, pp. 259–271. Available from DOI: `10.3233/AO-150157`.
4. KAUTZ, Oliver; ROTH, Alexander; RUMPE, Bernhard. Achievements, Failures, and the Future of Model-Based Software Engineering. In: *The Essence of Software Engineering*. Cham: Springer, 2018, pp. 221–236. ISBN 978-3-319-73896-3.
5. FIT CTU. *OpenPonk modeling platform* [online]. 2023. [visited on 2024-04-11]. Available from: `https://openponk.org/`.
6. RYBOLA, Zdeněk. *Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases*. Prague, 2017. PhD thesis. Czech Technical University in Prague.
7. EMBLEY, David W.; THALHEIM, Bernhard. *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*. Berlin, Germany: Springer, 2011. ISBN 978-3-642-15865-0.
8. OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language Version 2.5* [online]. 2015. [visited on 2024-04-04]. Available from: `https://www.omg.org/spec/UML/2.5/PDF`.
9. BADREDDIN, Omar; KHANDOKER, Rahad; FORWARD, Andrew; MASMALI, Omar; LETHBRIDGE, Timothy C. A Decade of Software Design and Modeling: A Survey to Uncover Trends of the Practice. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Copenhagen Denmark: ACM, 2018, pp. 245–255. ISBN 978-1-4503-4949-9. Available from DOI: `10.1145/3239372.3239389`.
10. BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide*. 2nd ed. Upper Saddle River, New Jersey: Addison-Wesley, 2005. ISBN 978-0-321-26797-9.

11. OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language Superstructure Version 2.4* [online]. 2011. [visited on 2024-04-22]. Available from: `https://www.omg.org/spec/UML/2.4/Superstructure/PDF`.

12. GUIZZARDI, Giancarlo; BOTTI BENEVIDES, Alessander; FONSECA, Claudenir M.; PORELLO, Daniele; ALMEIDA, João Paulo A.; PRINCE SALES, Tiago. UFO: Unified Foundational Ontology. *Applied Ontology*. 2022, vol. 17, no. 1, pp. 167–210. Available from DOI: `10.3233/AO-210256`.

13. GUIZZARDI, Giancarlo; FONSECA, Claudenir M.; BENEVIDES, Alessander Botti; ALMEIDA, João Paulo A.; PORELLO, Daniele; SALES, Tiago Prince. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: *Conceptual Modeling*. Cham, Switzerland: Springer, 2018, vol. 11157, pp. 136–150. Available from DOI: `10.1007/978-3-030-00847-5_12`.

14. GUIZZARDI, Giancarlo; WAGNER, Gerd; SINDEREN, Marten van. A Formal Theory of Conceptual Modeling Universals. In: *Proceedings of the First International Workshop on Philosophy and Informatics, Cologne*. Cologne, Germany, 2004.

15. SUCHÁNEK, Marek. *OntoUML specification Documentation* [online]. 2022. [visited on 2024-04-04]. Available from: `https://ontouml.readthedocs.io/_/downloads/en/latest/pdf/`.

16. GUIZZARDI, Giancarlo. *Ontological Foundations for Structural Conceptual Models*. Enschede, The Netherlands, 2005. PhD thesis. University of Twente.

17. GUIZZARDI, Giancarlo. Modal Aspects of Object Types and Part-Whole Relations and the de re/de dicto Distinction. In: *Advanced Information Systems Engineering*. Berlin, Germany: Springer, 2007, vol. 4495, pp. 5–20. ISBN 978-3-540-72987-7. Available from DOI: `10.1007/978-3-540-72988-4_2`.

18. GUIZZARDI, Giancarlo. Ontological Foundations for Conceptual Part-Whole Relations: The Case of Collectives and Their Parts. In: *Advanced Information Systems Engineering*. Berlin, Germany: Springer, 2011, vol. 6741, pp. 138–153. ISBN 978-3-642-21639-8. Available from DOI: `10.1007/978-3-642-21640-4_12`.

19. GUIZZARDI, Giancarlo. On the Representation of Quantities and Their Parts in Conceptual Modeling. In: *Proceedings of the 2010 Conference on Formal Ontology in Information Systems: Proceedings of the Sixth International Conference (FOIS 2010)*. 2010, pp. 103–116. ISBN 978-1-60750-534-1. Available from DOI: `10.5555/1804715.1804728`.

20. ATZENI, Paolo; CERI, Stefano; PARABOSCHI, Stefano; TORLONE, Riccardo. *Database Systems: Concepts, Languages & Architectures*. New York: McGraw-Hill, 1999. ISBN 978-0-07-709500-0.

21. CODD, Edgar F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. 1970, vol. 13, no. 6, pp. 377–387. ISSN 0001-0782, ISSN 1557-7317. Available from DOI: `10.1145/362384.362685`.

22. CHAMBERLIN, Donald D. Relational Data-Base Management Systems. *ACM Computing Surveys*. 1976, vol. 8, no. 1, pp. 43–66. ISSN 0360-0300. Available from DOI: `10.1145/356662.356665`.

23. ATZENI, Paolo; DE ANTONELLIS, Valeria; BATINI, Carlo. *Relational Database Theory*. Redwood City, California: Addison Wesley, 1993. ISBN 978-0-8053-0249-3.

24. GREENWALD, Rick; STACKOWIAK, Robert; STERN, Jonathan. *Oracle Essentials: Oracle Database 12c*. 5th ed. Sebastopol, California: O'Reilly, 2013. ISBN 978-1-4493-4303-3.

25. ORACLE. *Oracle Database SQL Language Reference: constraint* [online]. 2017. [visited on 2024-04-10]. Available from: `https://docs.oracle.com/database/121/SQLRF/clauses002.htm`.

26. ORACLE. *Oracle Database SQL Language Reference: CREATE VIEW* [online]. 2017. [visited on 2024-04-10]. Available from: `https://docs.oracle.com/database/121/SQLRF/statements_8004.htm`.

27. CHEN, Peter Pin-Shan. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*. 1976, vol. 1, no. 1, pp. 9–36. ISSN 0362-5915, ISSN 1557-4644. Available from DOI: `10.1145/320434.320440`.

28. GORNIK, Davor. *UML Data Modeling Profile*. 2003.

29. WARMER, Jos B.; KLEPPE, Anneke G. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd ed. Boston, Massachusetts: Addison-Wesley, 2003. ISBN 978-0-321-17936-4.

30. ORACLE. *Oracle Database SQL Language Reference: SET CONSTRAINT[S]* [online]. 2017. [visited on 2024-04-10]. Available from: `https://docs.oracle.com/database/121/SQLRF/statements_10003.htm`.

31. PHARO. *Pharo - features* [online]. [visited on 2024-04-12]. Available from: `https://pharo.org/features`.

32. DUCASSE, Stéphane. *Pharo with Style*. Paris, France: Books on Demand, 2021. ISBN 978-2-322-18201-5.

33. DUCASSE, Stéphane; RAKIC, Gordana; KAPLAR, Sebastijan; DUCASSE, Quentin. *Pharo 9 by Example*. Paris, France: Books on Demand, 2022. ISBN 978-2-322-39410-4.

34. FABRY, Johan; DUCASSE, Stéphane. *The Spec UI framework*. Switzerland: Square Bracket Associates, 2017. ISBN 978-1-326-92746-2.

35. INGENO, Joseph. *Software Architect's Handbook*. Birmingham, UK: Packt Publishing, 2018. ISBN 978-1-78862-406-0.

36. *Database Language SQL — Part 2: Foundation (SQL/Foundation)*. Geneva, Switzerland, 1999-09. Standard, ISO/IEC 9075-2:1999. International Organization for Standardization.

37. BERGEL, Alexandre; CASSOU, Damien; DUCASSE, Stéphane; LAVAL, Jannik. *Deep into Pharo*. Switzerland: Square Bracket Associates, 2013. ISBN 978-3-9523341-6-4.

38. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object–Oriented Software*. 1st ed. Boston, Massachusetts: Addison-Wesley, 1994. ISBN 978-0-201-63361-0.

Attachments

This master's thesis is accompanied by an archive containing the electronic version of the thesis and its source code, and the source code and compiled binaries of the implementation created as part of this thesis. The structure of the archive is shown below.

- `readme.txt` .... brief overview of the archive contents
- `bin/` .... compiled binaries of OpenPonk and Moco
- `thesis/` .... LaTeX sources of the text of the thesis
  - `thesis.pdf` .... text of the master's thesis in PDF format
  - `thesis.tex` .... main source file of the thesis
  - `figures/` .... directory with figures used in the thesis
  - `text/` .... sources of individual chapters
- `src/` .... source code of the implementation
  - `moco/` .... Moco repository
  - `openponk-moco/` .... OpenPonk+Moco CI repository

Appendix A

Example Model

To evaluate the implemented transformation, we use the model of a library information system from [6]. The input OntoUML model and the results generated by the implementation are structured as follows.

- In section A.1, the input OntoUML model is introduced.
- The result of the transformation round from the OntoUML model to a UML model is provided in section A.2.
- In section A.3, the result of the transformation of the UML model to a relational model is provided.
- The result of the transformation of the relational model to SQL is summarized in section A.4.

Additionally, the OpenPonk project files, plain-text model dumps, OCL constraints and SQL scripts are provided in the attached archive.

A.1 OntoUML Model

The OntoUML model used as the input to the transformation is provided in figure A.1. A list of types and relations used in the model is provided in table A.1.

Type	Count	Relation	Count
Kind	4	generalization	31
SubKind	4	generalization set	10
Relator	4	Formal	1
Collective	1	Mediation	8
Quantity	0	Material	0
Role	9	Characterization	4
Phase	8	ComponentOf	0
Category	3	Containment	0
RoleMixin	1	MemberOf	1
PhaseMixin	0	SubCollectionOf	0
Mixin	1	SubQuantityOf	0
Mode	3		
Quality	1		

Table A.1: List of elements in the example OntoUML model

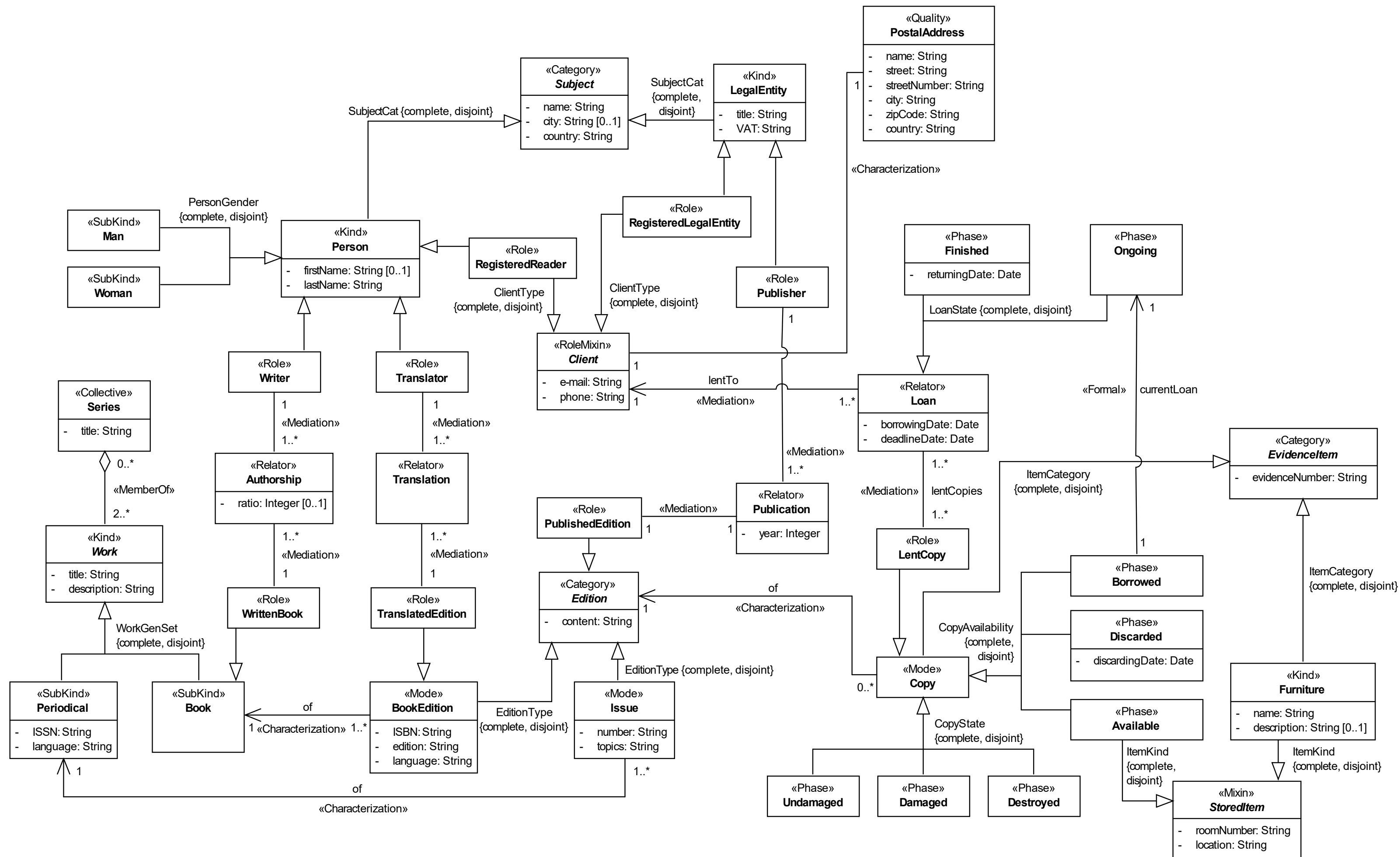

Figure A.1: Example OntoUML model

A.2 UML Model

A class diagram of the output of the implemented transformation of the example OntoUML to UML is provided in figure A.2. A list of transformed elements is provided in table A.2.

Classes	**Count**	**Relations**	**Count**
classes	30	generalizations	14
immutable attributes	1	generalization sets	7
		associations	22
		immutable association sides	20
OCL Constraints			**Count**
exclusive association			1
enumeration			2

Table A.2: List of elements in the example UML model

A list of available options during the transformation from OntoUML to UML and selected choices follows below.

- A phase partition was found. Please select how to transform `Loan` and its phases `Ongoing`, `Finished`.
  ≫ Transform by an abstract phase.
- A phase partition was found. Please select how to transform `Copy` and its phases `Destroyed`, `Damaged`, `Undamaged`.
  ≫ Transform to a phase attribute.
- A phase partition was found. Please select how to transform `Copy` and its phases `Available`, `Borrowed`, `Discarded`.
  ≫ Transform using exclusive phase associations.
- Types `Woman`, `Man` in generalization set `PersonGender` at `Person` have no properties and can be optimized out.
  ≫ Replace with a discriminator attribute.
- Role `Writer` has no properties and can be optimized out.
  ≫ Keep as is.
- Role `WrittenBook` has no properties and can be optimized out.
  ≫ Remove from model.
- Role `Translator` has no properties and can be optimized out.
  ≫ Keep as is.
- Role `TranslatedEdition` has no properties and can be optimized out.
  ≫ Remove from model.
- Role `Publisher` has no properties and can be optimized out.
  ≫ Keep as is.
- Role `PublishedEdition` has no properties and can be optimized out.
  ≫ Remove from model.
- Role `LentCopy` has no properties and can be optimized out.
  ≫ Remove from model.
- Relator `Translation` has no properties and can be optimized out.
  ≫ Remove from model.

Figure A.2: Transformed UML model

A.3 Relational Model

A diagram using the UML profile for data modeling that shows the result of the implemented transformation of the UML model to a relational model is provided in figure A.3. A list of transformed elements is provided in table A.3.

Elements	**Count**
tables	33
foreign keys	39
unique keys	12
OCL Constraints	**Count**
generalization set	7
special multiplicity	1
mandatory multiplicity	10
enumeration	2
exclusive association	1
immutable column	35
immutable delete	16

Table A.3: List of elements in the example RDB model

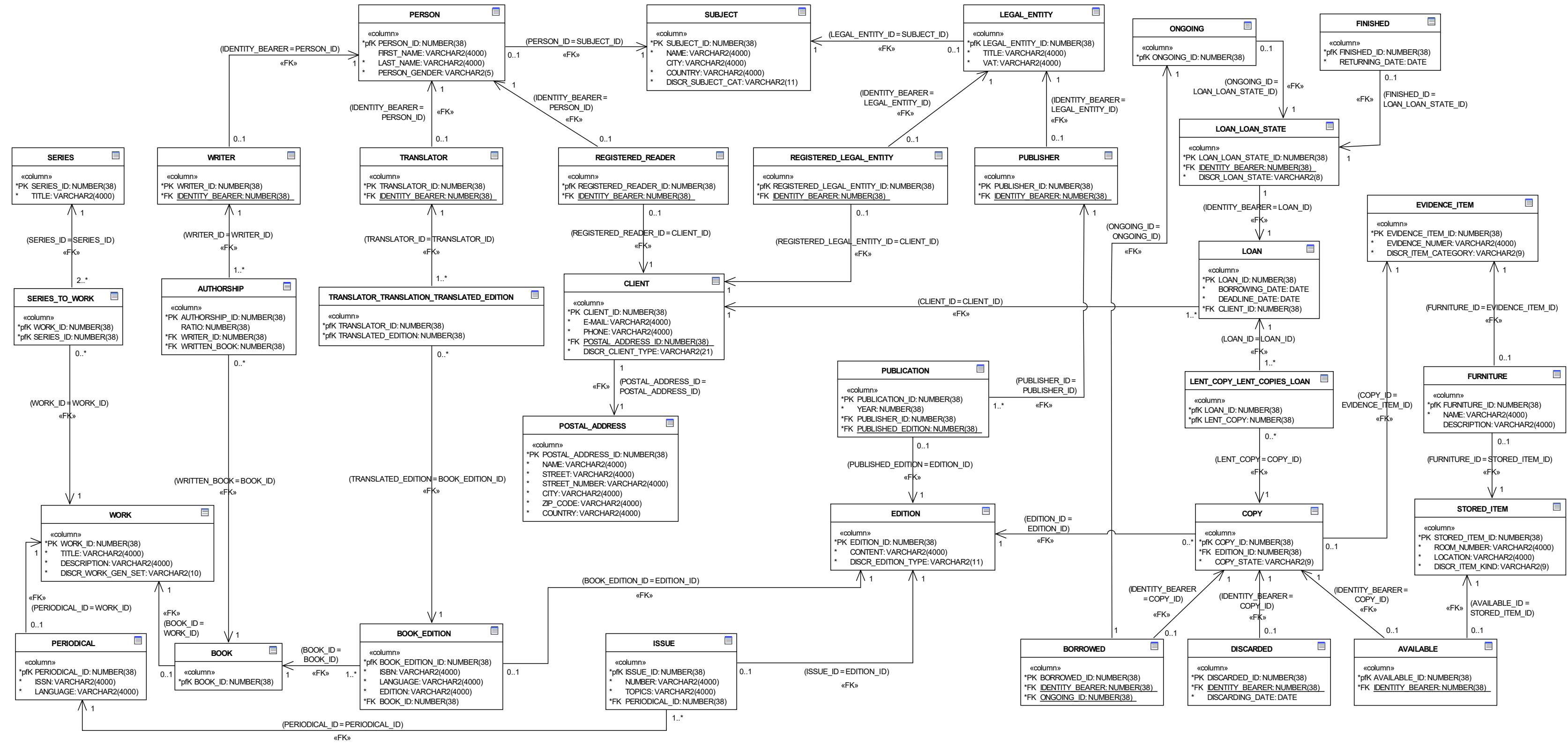

Figure A.3: Transformed relational model

A.4 Realization in SQL

A list of the types of generated SQL statements by the implemented transformation of the relational model to SQL is provided in table A.4.

Statement Type	Count
`CREATE TABLE`	33
`ADD FOREIGN KEY`	39
`ADD UNIQUE`	12
`ADD CHECK`	2
`CREATE TRIGGER BEFORE`	115
`CREATE TRIGGER AFTER`	14

Table A.4: List of generated SQL statements

Appendix **B**

Transformation Rules

In this appendix, a complete list of implemented transformation rules and their priorities is provided. It is divided according to the designed transformation rounds.

- In section B.1, a list of transformation rules that convert an OpenPonk OntoUML model to a Moco model is provided.
- In section B.2, transformation rules that transform an OntoUML model to UML are listed.
- In section B.3, transformation rules that refactor a UML model prior to the transformation to a relational model are listed.
- In section B.4, a list of transformation rules that realize the transformation of a UML model to a relational model is provided.
- In section B.5, transformation rules that convert a relational data model to SQL are listed.
- In section B.6, a list of transformation rules that convert a Moco UML model to an OpenPonk UML model is provided.
- In section B.7, a list of transformation rules that convert a Moco relational model to an OpenPonk UML model is provided.

B.1 OpenPonk OntoUML to Moco OntoUML

Transformation rules that convert an OntoUML model from OpenPonk data structures to Moco data model are listed in table B.1.

Rule	Class Name	Priority
1.1	`MocoOpenPonkOntoUmlClassTransformationRule`	100
1.2	`MocoOpenPonkOntoUmlKindTransformationRule`	100
1.3	`MocoOpenPonkOntoUmlSubKindTransformationRule`	100
1.4	`MocoOpenPonkOntoUmlQuantityTransformationRule`	100

continued on next page

Table B.1: List of OpenPonk OntoUML to Moco OntoUML transformation rules

Rule	Class Name	Priority
1.5	MocoOpenPonkOntoUmlRelatorTransformationRule	100
1.6	MocoOpenPonkOntoUmlCollectiveTransformationRule	100
1.7	MocoOpenPonkOntoUmlRoleTransformationRule	100
1.8	MocoOpenPonkOntoUmlPhaseTransformationRule	100
1.9	MocoOpenPonkOntoUmlCategoryTransformationRule	100
1.10	MocoOpenPonkOntoUmlRoleMixinTransformationRule	100
1.11	MocoOpenPonkOntoUmlPhaseMixinTransformationRule	100
1.12	MocoOpenPonkOntoUmlMixinTransformationRule	100
1.13	MocoOpenPonkOntoUmlQualityTransformationRule	100
1.14	MocoOpenPonkOntoUmlModeTransformationRule	100
1.25	MocoOpenPonkOntoUmlEnumerationTransformationRule	80
1.26	MocoOpenPonkOntoUmlGeneralizationTransformationRule	60
1.27	MocoOpenPonkOntoUmlGeneralizationSetTransformationRule	59
1.15	MocoOpenPonkOntoUmlAssociationTransformationRule	51
1.16	MocoOpenPonkOntoUmlContainmentAssociationTransforma-tionRule	50
1.17	MocoOpenPonkOntoUmlFormalAssociationTransformationRule	50
1.18	MocoOpenPonkOntoUmlMaterialAssociationTransformation-Rule	50
1.19	MocoOpenPonkOntoUmlMediationAssociationTransformation-Rule	50
1.20	MocoOpenPonkOntoUmlCharacterizationAssociationTransfor-mationRule	50
1.21	MocoOpenPonkOntoUmlComponentOfAssociationTransforma-tionRule	50
1.22	MocoOpenPonkOntoUmlMemberOfAssociationTransformation-Rule	50
1.23	MocoOpenPonkOntoUmlSubCollectionOfAssociationTransfor-mationRule	50
1.24	MocoOpenPonkOntoUmlSubQuantityOfAssociationTransforma-tionRule	50

Table B.1: List of OpenPonk OntoUML to Moco OntoUML transformation rules

B.2 OntoUML to UML

In table B.2, the list of transformation rules that realize the transformation round from an OntoUML to a UML model is provided.

Rule	Class Name	Priority
2.2	MocoOntoUmlKindTransformationRule	100
2.3	MocoOntoUmlSubKindTransformationRule	100
2.4	MocoOntoUmlRelatorTransformationRule	100
2.5	MocoOntoUmlQuantityTransformationRule	100
2.6	MocoOntoUmlCollectiveTransformationRule	100
2.7	MocoOntoUmlModeTransformationRule	100
2.8	MocoOntoUmlQualityTransformationRule	100
2.9	MocoOntoUmlCategoryTransformationRule	100
2.10	MocoOntoUmlMixinTransformationRule	100
2.11	MocoOntoUmlRoleMixinTransformationRule	100
2.12	MocoOntoUmlPhaseMixinTransformationRule	100
2.13	MocoOntoUmlRoleTransformationRule	99
2.14	MocoOntoUmlPhaseTransformationRule	99
2.1	MocoOntoUmlClassTransformationRule	95
2.21	MocoOntoUmlEnumerationTransformationRule	80
2.22	MocoOntoUmlGeneralizationTransformationRule	50
2.23	MocoOntoUmlGeneralizationSetTransformationRule	49
2.16	MocoOntoUmlCharacterizationAssociationTransformation-Rule	45
2.17	MocoOntoUmlMediationAssociationTransformationRule	45
2.18	MocoOntoUmlMaterialAssociationTransformationRule	45
2.20	MocoOntoUmlContainmentAssociationTransformationRule	45
2.19	MocoOntoUmlPartWholeAssociationTransformationRule	44
2.15	MocoOntoUmlAssociationTransformationRule	43
2.24	MocoOntoUmlRoleOptimizationTransformationRule	10
2.25	MocoOntoUmlRelatorOptimizationTransformationRule	9

Table B.2: List of OntoUML to UML transformation rules

B.3 Preprocessing UML for the Relational Model

Transformation rules responsible for refactoring a UML model before its transformation to a relational model are listed in table B.3.

Rule	Class Name	Priority
3.1	MocoUmlCopyTransformationRule	50
3.2	MocoUmlCopyFixReferencesTransformationRule	49
3.3	MocoUmlEnumerationTransformationRule	30
3.4	MocoUmlNonNativeRdbTypeTransformationRule	25
3.5	MocoUmlMultiValueAttributeTransformationRule	23
3.6	MocoUmlManyToManyAssociationTransformationRule	20

Table B.3: List of UML to UML for RDB transformation rules

B.4 UML to a Relational Model

In table B.4, the list of transformation rules that realize the transformation round from a UML to a relational model is provided.

Rule	Class Name	Priority
4.1	MocoUmlClassTransformationRule	100
4.3	MocoUmlGeneralizationTransformationRule	90
4.4	MocoUmlGeneralizationSetTransformationRule	89
4.2	MocoUmlAssociationTransformationRule	85
4.6	MocoUmlOclEnumerationConstraintTransformationRule	50
4.7	MocoUmlOclExclusiveAssociationConditionTransformation-Rule	50
4.5	MocoUmlMediatingClassOptimizationTransformationRule	35

Table B.4: List of UML to RDB transformation rules

B.5 Relational Model to Oracle SQL

Transformation rules that realize the transformation round from a relational model to SQL are listed in table B.5.

Rule	Class Name	Priority
5.1	MocoRdbTableTransformationRule	100
5.2	MocoRdbForeignKeyTransformationRule	99
5.3	MocoRdbOclEnumerationTransformationRule	80

continued on next page

Table B.5: List of RDB to SQL transformation rules

Rule	Class Name	Priority
5.4	MocoRdbOclExclusiveAssociationTransformationRule	0
5.5	MocoRdbOclGeneralizationSetTransformationRule	0
5.6	MocoRdbOclImmutableColumnTransformationRule	0
5.7	MocoRdbOclImmutableAssociationDeleteTransformationRule	0
5.8	MocoRdbOclMandatoryMultiplicityTransformationRule	0
5.9	MocoRdbOclSpecialMultiplicityTransformationRule	0

Table B.5: List of RDB to SQL transformation rules

B.6 Moco UML to OpenPonk UML

In table B.6, the list of transformation rules that convert a Moco UML model to an OpenPonk UML model is provided.

Rule	Class Name	Priority
6.1	MocoOpenPonkUmlClassTransformationRule	100
6.3	MocoOpenPonkUmlEnumerationTransformationRule	90
6.4	MocoOpenPonkUmlGeneralizationTransformationRule	65
6.5	MocoOpenPonkUmlGeneralizationSetTransformationRule	60
6.2	MocoOpenPonkUmlAssociationTransformationRule	50
6.6	MocoOpenPonkUmlOclTransformationRule	40

Table B.6: List of Moco UML to OpenPonk UML transformation rules

B.7 Moco RDB to OpenPonk UML

Transformation rules that convert a Moco relational model to an OpenPonk UML model are listed in table B.7.

Rule	Class Name	Priority
7.1	MocoOpenPonkRdbTableTransformationRule	100
7.2	MocoOpenPonkRdbForeignKeyTransformationRule	90
7.3	MocoOpenPonkRdbOclTransformationRule	50

Table B.7: List of Moco RDB to OpenPonk UML transformation rules