UNIVERSITÄT
KOBLENZ · LANDAU

Masterarbeit

Mapping ORM to TGraph

Alicia Owen
November 30, 2017

Gutachter: Prof. Dr. Jan Jürjens
Dr. Volker Riediger

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz
https://rgse.uni-koblenz.de

Alicia Owen
owen@uni-koblenz.de
Matrikelnummer: 207110010
Studiengang: Master Informatik
Prüfungsordnung: PO2012

Thema: Mapping ORM to TGraph

Eingereicht: November 30, 2017

Betreuer: Katharina Großer

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz

Ehrenwörtliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-ROM).

Koblenz, den 30. November 2017

Alicia Owen

Abstract

Object Role Modeling (ORM) ist eine semantische Modelliersprache, die Objekte und deren Beziehungen untereinander beschreibt. Sowohl die Objekte als auch ihre Beziehungen können bestimmten Regeln (*constraints*) unterliegen.

TGraphen sind geordnete, attributierte, typisierte und gerichtete Graphen. Der Typ eines TGraphen und seiner Bestandteile, der Kanten und Knoten, wird in Form eines *graph UML* (grUML) Schemas definiert. GrUML ist eine durch Profile erweiterte Form von UML Klassendiagrammen. Das Ziel dieser Arbeit ist es ORM-Schemata in grUML-Schemata umzuwandeln um anschließend Instanzen eines ORM-Schemas in Form eines TGraphen darstellen zu können.

Bis zu diesem Zeitpunkt ist die bevorzugte Darstellung und Speicherung von Instanzen eines ORM-Schemas in Form von relationalen Tabellen. Obwohl es Regeln für die Umwandlung von ORM-Schemata in relationale Schemata gibt, unterstützen die öffentlich verfügbaren Umwandlungstools nur wenige der in ORM definierbaren *constraints*.

Diese *constraints* können in einem grUML-Schema mithilfe der TGraphen-Anfragesprache *GReQL* formuliert werden. GReQL erlaubt eine effiziente Überprüfung, ob TGraphen die definierten Regeln erfüllen oder nicht.

Die Graphbibliothek *JGraLab* liefert eine effiziente Implementation von TGraphen und ihrer Anfragesprache GReQL und unterstützt das Erstellen von grUML-Schemata.

Das erste Ziel dieser Arbeit ist es, eine vollständige und korrekte Umwandlung eines ORM-Schemas in ein grUML-Schema zu definieren. Das zweite Ziel ist es ORM-Schema-Instanzen als TGraphen darzustellen.

Im Rahmen dieser Arbeit wird ein Überblick über ORM, TGraphen, grUMl und GReQL sowie über die theoretische Transformation eines ORM-Schemas in ein grUML-Schema gegeben. Des weiteren wird die Implementation der Transformation vorgestellt. Außerdem befasst sich diese Arbeit mit der Repräsentation von ORM-Schema-Instanzen als TGraphen und der Frage, wie die im grUML-Schema definierten *constraints* überprüft werden können.

Abstract

Object Role Modeling (ORM) is a semantic modeling language used to describe objects and their relations amongst each other. Both objects and relations may be subject to *rules* or ORM *constraints*.

TGraphs are ordered, attributed, typed and directed graphs. The type of a TGraph and its components, the edges and vertices, is defined using the schema language graph UML (grUML), a profiled version of UML class diagrams. The goal of this thesis is to map ORM schemas to grUML schemas in order to be able to represent ORM schema instances as TGraphs.

Up to this point, the preferred representation for ORM schema instances is in form of relational tables. Though mappings from ORM schemas to relational schemas exist, those publicly available do not support most of the constraints ORM has to offer.

Constraints can be added to grUML schemas using the TGraph query language *GReQL*, which can efficiently check whether a TGraph validates the constraint or not. The graph library *JGraLab* provides efficient implementations of TGraphs and their query language GReQL and supports the generation of grUML schemas.

The first goal of this work is to perform a complete mapping from ORM schemas to grUML schemas, using GReQL to sepcify constraints. The second goal is to represent ORM instances in form of TGraphs.

This work gives an overview of ORM, TGraphs, grUML and GReQL and the theoretical mapping from ORM schemas to grUML schemas. It also describes the implementation of this mapping, deals with the representation of ORM schema instances as TGraphs and the question how grUML constraints can be validated.

Contents

List of Figures

1 Introduction

1.1 Motivation

The semantic modeling language *Object Role Modeling* (*ORM*) [Hal09] views the world in terms of *objects* which play *roles*. It is well established in the conceptual design of relational databases. Relational schemas, which define relational tables, have a considerable drawback: while a modeler can define numerous different *constraints* in ORM, these do not translate easily to relational schemas. ORM also offers the possibility of deriving information from the relations between objects through *derivations*. Though it is possible, their implementation in relational tables is tedious and error-prone.

Still, relational tables are the established format to store ORM schema instances. But having a rich modeling language one side and a schema language which struggles to keep up on the other side, seems wasteful.

For this reason, this thesis attempts a different approach to the representation of ORM schema instances.

1.2 Research Question

The goal of this master thesis is to represent instances of schemas supplied in the modeling language ORM as graphs, more specifically *TGraphs* [EF95].

In a first step to achieve this goal, the ORM schemas need to be mapped to a TGraph schema language, *graph UML* (*grUML*)[BHR$^+$10]. ORM provides an extensive constraint notation which also needs to be represented in grUML schemas. This can be achieved by using the graph query language GReQL [KK01][EB10] to formulate grUML constraints and adding these to the grUML schemas.

Besides the definition and implementation of this mapping from ORM to grUML schemas, the second goal is the representation of ORM instances as TGraphs. This involves checking whether TGraph instances violate the grUML constraints generated from the ORM schema during mapping.

The Java library `JGraLab` developed at the University of Koblenz-Landau implements TGraphs, their schema language grUML and the TGraph query language GReQL and will be used throughout this project.

1.3 Thesis Outline

This thesis is structured as follows.

Within the introduction to this work, chapter 1, ection 1.4 will give a brief introduction into information modeling and information systems in general, while section 1.5 will introduce a specific method for information modeling called Object Role Modeling (ORM). In this section, basic concepts of ORM and their notation will be explained, followed by constraints and more complex constructs. Section 1.6 is dedicated to the introduction of TGraphs followed by their schema language grUML in section 1.7 and the graph querying language GReQL in section 1.8.

Chapter 2 introduces the proposed mapping from ORM schemas to grUML schemas. Chapter 3 is concerned with giving an overview of the Java code implemented to realize the mapping defined in chapter 2 and to generate ORM schema instances in form of TGraphs. Chapter 4 discusses the results of chapters 2 and 3 and finally, chapter 5 concludes this work.

1.4 Information Modeling

In a digital world, each person constantly produces data: when posting on social media, when tracking steps on a smartwatch, when transferring money or buying a book at a shop. At first, the data we provide may seem useless, but consider the process of buying a book. Not only could the choice of book give information on what genres I may be interested in, where I'm traveling to, what health issues I have or what I'm interested in professionally, but the place and time at which I buy it might indicate where I live and what I do for a living. The more pieces of data can be placed together, the more reliable the information gets. Connecting the *data* (title of a book) with meaning or *semantics* (it's a book about hiking in New Mexico) provides *information* (the person buying the book has some interest in hiking and/or New Mexico).

In the example above, the generation of information from data is difficult since the exact meaning of the data isn't known. However, there are many areas or *domains* in which data is produced and its semantics are well defined. In such cases it is possible to create a system which stores this information, a so-called *information system*. Its key component is the *information model* [HM08, ch. 1].

1.4.1 Information Model

The information model contains general knowledge about the domain: it describes the different kinds of pieces of data and their relationships amongst each other, rules that apply to these relationships and how new information can be gained from existing information. While the information model retains the semantics of the data, data can be added to a *database* or *information base*[1]. Adding data to a database is referred to as *populating* the database. The information model should serve as a blueprint for storing data along with its semantics and can allow the automatic generation of databases, object models and user interfaces.

Each information model represents a business domain or *Universe of Discourse* (*UoD*), which is often based on a part of the real world but may contain simplifications or ignore facts that play no role for the intended use of the information system.

An information model is created by a *modeler* who works in cooperation with a *domain expert* who is familiar with the domain that is being modeled.

[1] Please note that the term 'database' refers to a collection of data without specifying a certain format. The meaning of the data added to such a collection is provided by the information model.

In order to build a clear, concise and unambiguous model of the UoD, the modeler needs to have or obtain a deep understanding of its structure.

Information modeling is the process of creating an information model. An advantage of using information models is that they represent data along with rules for its interpretation in an organized fashion and can be shared, reused or extended.

Multiple languages are available to formally represent information models, such as *IDEF1X* [IDE12], *EXPRESS* [EXP04], *Unified Modeling Language* (*UML*) [RJB04], *Entity Relationship* (*ER*) [sC76] or *Object Role Modeling* (*ORM*) [Hal09]. While ER is a very popular high-level approach to database design, UML is the dominant language for an object-oriented approach to information modeling.

Since the design process for an information model requires a great amount of care, there are modeling methods such as the Object Role Modeling method (described in the next section) which consist of both a language for representation of the model as well as a procedure describing how to use the language to build it.

1.4.2 Information System

An information system is used to collect, store, process and distribute information [Oli07]. Four different levels can be distinguished within it: the *conceptual*, *logical*, *physical* and *external level* [HM08, ch. 2].

Conceptual level

The conceptual level is the most fundamental and thus most important level of an information system. In this level, the system's business domain is described in terms of basic, easily understandable concepts and rules that apply to them. This description is also known as the *conceptual schema*. It is of great importance for clear communication between modelers and domain experts. The conceptual schema is designed without taking implementation concerns into account.

The conceptual schema defines the structure of the UoD while the *conceptual database* provides information about concrete *instances* that populate the UoD at a given time. The conceptual schema provides a high-level overview over the UoD and is mapped to a logical data model for implementation.

Logical level

The *logical schema* describes the abstract structures in which data will be stored and operations that can be applied to it. Two examples of logical data models are the relational (table-oriented) or the object-oriented data model. The logical schema is tailored to the programming language that is used for the actual implementation of the information system.

Physical level

At the physical level, the logical schema is transformed into a *physical schema*. This includes information about physical data storage and how the data is accessed within the system (e.g. using an index or file clustering).

External level

At the external level, the so-called *external schema* specifies what kind of facts can be read, added or removed by users and how they are displayed. The availability of these operations and general access rights can be customized for specific user groups. Furthermore, different user groups may be provided with interfaces to the information system that are tailored to their expertise levels.

Amongst all these levels, the conceptual level is the most stable. Each conceptual schema can be mapped to several different logical schemas and each logical schema itself can have several realizations on the physical level. Due to the profound impact the conceptual level has on the implementation and maintenance of an information system, it is of great importance that the conceptual schema be generated with great care.

1.5 Object Role Modeling

Object Role Modeling (*ORM*) is a semantic modeling approach. It has its origins in the early 1970s and exists in various variations. Since 2009, the second generation of ORM, ORM2, is available for use. In this thesis the term "ORM" refers to ORM2 if not stated otherwise.

ORM considers the UoD in terms of *objects* that play *roles*.

In order to construct a conceptual schema, the interactions between objects are stated in the form of *facts* which are statements that are assumed to be true in the UoD. For this reason, ORM is categorized as a *fact oriented* modeling method.

In ORM, a conceptual schema can be represented as a diagram or in textual form. The textual form is based on the ORM language FORML (Formal ORM Language). Depending on the complexity of the schema, the diagrammatic form may not be as expressive as the textual form and thus, diagrams can be complemented by text. The schemas used for this thesis were provided in a graphical representation generated in the ORM modeling software *NORMA* (*Natural ORM Architect for Visual Studio*)[2]. In the further course of this work, the term 'ORM schema' will refer to the diagrammatic form of a conceptual ORM schema which will be supported by text only when necessary.

The most profound difference between ORM and other modeling approaches such as UML, ER, IDEF1X or EXPRESS, is that ORM works without the use of attributes. While this causes bloating of ORM diagrams compared to diagrams from other languages, this renders ORM models more stable in the face of schema evolution, i.e. when a model undergoes changes. ORM has several advantages over the other languages: its diagrams can be verbalized in controlled natural language which makes them easier to understand for people without a modeling background and facilitates communication about the UoD. In addition, the circumstance that each relationship is explicitly formulated as a fact type makes it easier to validate the conceptual schema. Furthermore, ORM has a simple yet highly expressive constraint notation which will be introduced in section 1.5.2.

Compared to other modeling languages, ORM considers semantic domains rather than syntactic domains (e.g. string, integer): modeling a 'Person' in ER, it might have the attributes 'weight' and 'height' with values from the domain Integer. This allows a direct comparison between values for weight and height since they are from

[2] available at: `http://www.ormfoundation.org/files/folders/norma_the_software/default.aspx`

the same domain. This comparison, however, does not make much sense. In ORM, a ‘Person’ would be modeled as having a height which is recorded as a ‘Length’ (e.g. measured in meters) and as having a weight recorded as a ‘Mass’ (e.g. measured in kilograms). The semantic domains in this example are ‘Length’ and ‘Mass’. Since the domains are not the same, it is not possible to compare a Person’s height and weight if he is modeled in this way. Although comparing values from the domains ‘Length’ and ‘Mass’ is not permitted by the ORM model, their underlying representation is of the same data type (Integer).

Conceptual schemas formulated in ORM can be transformed into ER or UML class diagrams (see [HM08, ch. 8 and 9]). This can be useful in order to e.g. generate a more compact view of the conceptual schema or in database applications, where logical and physical schemas tend to be attribute-based.

An ORM conecptual schema displays *fact types*, *constraints*, *derivations* and *concept definitions*. The following sections should familiarize the reader with these concepts and their notations.

1.5.1 Basic Components

This section will begin by introducing the concept of facts which will later be abstracted to fact types which can be expressed in ORM conceptual schemas.

Facts

When building a conceptual schema in ORM, the modeler and domain expert generate examples of the kind of information that should later be represented in the information system. These examples are stated in the form of *facts*. A fact is a proposition that is taken to be true within the relevant UoD. A fact either declares that some individual *exhibits* a *property*, that one or more individuals take part in a *relationship*, or that an individual *exists* [HM08, ch. 3].

Assume the information model should represent information regarding students at a university. Consider the following example facts:

Fact 1 Sophie lives on campus.

Fact 2 Sophie lives in ’307’ .

These facts state that an *object* (Sophie) plays *roles* (living on campus, living in a room). Fact 1 expresses, that ‘Sophie’ has the property of living on campus while fact 2 expresses the relationship between ‘Sophie’ and her room ‘307’. In ORM, an object is an individual thing of interest that is of importance in the UoD. It can either be a *value* or an *entity* . In facts 1 and 2, ‘Sophie’ stands for “the student named ‘Sophie’” and ‘307’ stands for “the room with number ‘307’” which are *entities* while the strings ‘Sophie’ and ‘307’ are *values*.

Entities and Values

In general an entity is an object (e.g. a specific student, a specific room) which can unambiguously be referenced by its value (e.g. ‘Sophie’ or ‘307’) [HM08, p. 65 f]. In the example facts 1 and 2, the student ‘Sophie’ is a specific student that can unambiguously be referenced by her name (assuming, that in the UoD every possible first name occurs at most once). The same applies to the room with number ‘307’. Entities may change over time.

Predicates

In first order logic, a *predicate* is a relation over objects and either assigns a property to a single object or describes how two or more objects relate to each other. In ORM, a predicate is a proposition in which the objects of interest are replaced by placeholders [HM08, p. 67 ff]. Each predicate has a *reading*, in which the placeholder is an ellipsis (“...”).

In fact 1 the predicate reading is “... lives on campus”. This predicate assigns the property of living on campus to the entity ‘Sophie’. In fact 2, the predicate reading is “... lives in ...” and describes the relation between Sophie and the room she lives in.

The *arity* of a predicate is the number of objects that take part in a relation [HM08, p. 68]. In the example fact 1, only one object can take part in the relation. Thus, the arity of the predicate “.... lives on campus” is 1 and it is called a *unary* predicate.

In the case of fact 2, the predicate is “... lives in ...” and describes the relationship between the entity ‘Sophie’ and her room. Since two objects take part in this relationship, the arity of the predicate is 2 and it is also called a *binary* predicate. In ORM, predicates of any arity n ($n \in \mathbb{N}$) are allowed but usually relationships don’t span more than 3 objects which corresponds to a *ternary* predicate.

Elementary Facts

An ORM schema is generated from example facts. It is important to note that these need to be *elementary* facts, meaning that the information conveyed by such facts cannot be represented by using a set of smaller fragments of each fact (with the same objects) [HM08, p. 64]. An example of a non-elementary fact instance is shown in the following example:

Fact 3 Sophie lives on campus and Robert lives on campus.

In this case, the fact can be split into two individual elementary facts (Sophie lives on campus. Robert lives on campus.) which is indicated by the word “and”. Further words that indicate that a fact may not be elementary are “not”, “or”, “if”.

Example facts 1 and 2 are concrete facts about concrete objects. Since a conceptual schema displays information about concepts rather than concrete objects, these example facts are abstracted to *fact types* in order to model them in an ORM schema.

Fact Types

Fact types describe properties and relationships of *object types* [HM08, ch. 3]. Facts 1 and 2 are of the following fact types:

Fact type 1 Student (.name) lives on campus.

Fact type 2 Student (.name) lives in Room (.nr).

In order to get from the facts to the fact types, the entities “student named ‘Sophie”’ and “room with number ‘307”’ are abstracted to the *entity types* Student and Room. The mode in which specific instances of these entity types are referenced is defined by the parenthesized string appended to the entity type names.

Entity Types

In an information model, the *entity type* specifies what kind of entity is recorded. The entity type denotes the set of all instances of entities of this type that may be relevant throughout the lifetime of the information system. In our example, the entity type Student is the set of all people that are identified by their name and ‘Sophie’ is an instance [HM08, p. 66].

In an ORM schema, entity types appear as *named, soft rectangles containing the name of the entity type.* Figure 1.1 shows how a Student identified by its name would be represented in an ORM schema.

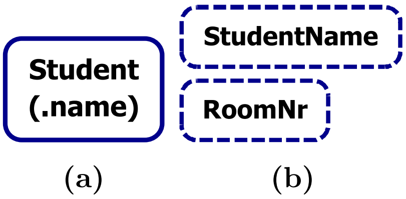


Figure 1.1: Figure a) displays how the entity type "Student" that can be identified by its StudentName is represented in ORM. Figure b) shows the representation of two value types, here of StudentName and RoomNr.

Reference Mode and Value Types

An entity type's *reference mode* describes how a single value refers to an entity [HM08, p. 67] and is placed in parentheses after the entity type's name. The entity type and reference mode "Student (.name)" explicitly means "Student has StudentName" where StudentName is a *value type* [HM08, p. 75] and the string 'Sophie' is a value type instance. Value types appear as *named, soft dashed rectangles containing the name of the entity type* in an ORM schema. Figure 1.1 shows the value types StudentName and RoomNr represented in ORM notation.

Object Types

In line with the definition of objects at the instance level, at the conceptual level an object type is either an entity type (e.g. Student, Room) or a value type (e.g. StudentName, RoomNr). Note that value types are mapped to data types, so in the example fact types StudentName and RoomNr are represented as strings.

Fact types can be populated by inserting concrete instances in place of the object types (thus creating a fact instance).

Population

A conceptual schema can be populated with object instances that play the roles defined by the ORM schema. For any given state of the resulting database, the population of an object type T, $pop(T)$, is the set of all instances of T in that given state. It is also possible to define the population of a role. For any given state of a database, the population of a role r, $pop(r)$ is the set of object type instances that play this role r [HM08, p. 161 ff].

Using the additional information contained in fact types 1 and 2 (the entity types, reference modes and value types), the example facts 1 and 2 could be reformulated as:

Fact 1 (extended) The Student with StudentName ‘Sophie’ lives on campus.

Fact 2 (extended) The Student with StudentName ‘Sophie’ lives in the Room with RoomNr ‘307’.

The fact blueprint represented by fact types 1 and 2 clearly defines what kind of objects play roles in facts 1 and 2 and by what means they can be identified. This allows storing the information about ‘Sophie’ and other students in an organized manner and allows computer processing without loss of information.

Summary

Table 1.1 gives an overview of the most important concepts covered in this section by analyzing an example fact. Note that the predicate in this example is binary.

Table 1.1: The most important basic ORM concepts extracted from an example fact.

fact	The Student named ‘Sophie’ lives in Room number ‘307’.
entities	The Student with StudentName ‘Sophie’, The Room with RoomNr ‘307’
entity types	Student, Room
values	‘Sophie’, ‘307’
value types	StudentName, RoomNr
reference schemes	Student(.name), Room(.nr)
predicate	lives in
fact type	Student(.name) lives in Room(.nr)

ORM is used to generate conceptual schemas which capture the semantics of data, but not the data itself. Although example facts are important for the generation of such schemas and help validating their correctness, they are not part of them. ORM schemas merely represent concept definitions, fact types, constraints that may apply to them and derivations that can be made from these fact types.

Figure 1.2 shows how the example fact type from table 1.1 would be represented in an ORM schema. Note that this figure shows a simplified version of an ORM schema, since it is missing uniqueness constraints which will be introduced in 1.5.2. Without uniqueness constraints it is an invalid schema.

N-ary predicates are displayed as a *named sequence of* n *adjacent squares* or *role boxes*. Each role box is connected to exactly one object type which indicates that only objects of this type can play this specific role. A predicate must be provided with at least one reading, which indicates how the objects playing the roles relate to one another.

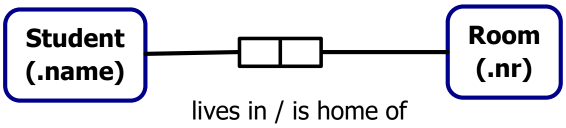


Figure 1.2: The fact type "Student(.name) lives in Room(.nr)" is displayed in a simplified ORM diagram. The reference mode is abbreviated by placing it in parentheses after the entity type name.

An overview of the most important ORM diagram elements is provided in appendix A.

1.5.2 Constraints

The previous section introduced the basic building blocks for ORM schemas and their notation. With diagrams like figure 1.2 it is possible to model a relationship between two entity types (Student, Room) and define their value types but it is not possible to express any rules that apply to their relationship. For example, it may be desirable to express that a Student can only live in one Room at any given point in time. Or that for each Student a Room must be recorded in the database. This is handled by constraints. In general, constraints are used to restrict the possible states and state transitions within the database [HM08, p. 110]. This section will introduce the various types of constraints that can be expressed graphically in ORM.

Intrapredicate Uniqueness Constraints

A basic kind of constraint that applies to every fact type in ORM is the *intrapredicate uniqueness constraint* or *internal uniqueness constraint* which specifies, for a given predicate, which roles or combinations of roles are played by unique objects or unique combinations of objects [HM08, p. 111 ff]. An intrapredicate uniqueness constraint is indicated by placing a bar above or below the fact role(s) it applies to.

Figure 1.3 shows all possible intrapredicate uniqueness constraints for binary predicates. Note that Students are now identified by their StudentIDs rather than their StudentNames, because the university allows students with identical names to enroll.

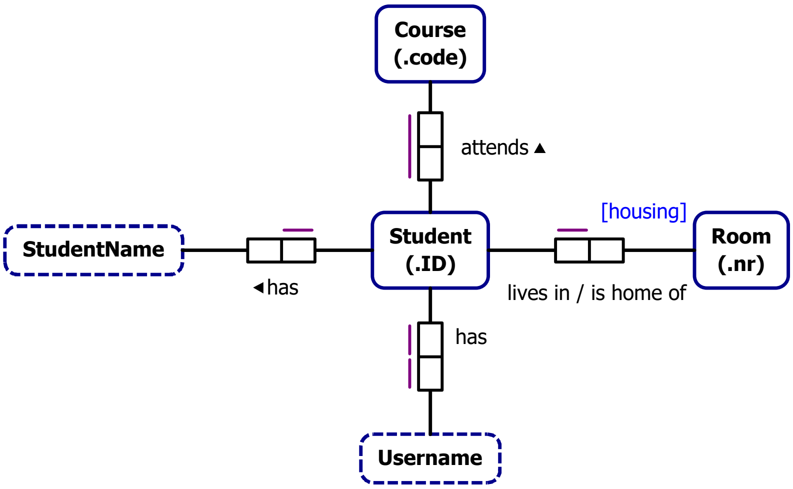


Figure 1.3: ORM schema showing (counter-clockwise beginning with the left predicate) n:1, 1:1, n:1 and m:n intrapredicate uniqueness constraints on binary predicates. This schema combines four fact types about Student(.ID) into one diagram.

Populating a fact type produces a so-called *fact table* [HM08, p. 85]. A fact table has a number of columns equal to the arity of the fact type's predicate. Each column stands for a role of this predicate and each row is filled with a combination of objects (entities or values) playing these roles.

The **1:1 uniqueness constraint** means that for each role of the predicate the entries in the corresponding column of the fact table need to be unique. In figure 1.3 this uniqueness constraint is applied to the predicate of the fact type "Student(.ID) has Username". It indicates that each Student is given exactly one unique Username which corresponds to a 1:1 mapping. Thus, the Username could also be used to uniquely identify a Student but in this case the domain expert and/or modeler preferred referencing the entity type Student by their StudentID.

From the **1:n uniqueness** constraint it follows that the entries in the fact table column corresponding to the left role need to be unique while the right column may contain duplicates. In figure 1.3 this constraint on the fact type "Student(.ID) has StudentName" means that each Student can have at most one (1) StudentName but multiple (n) Students can have the same StudentName.

With the **n:1 uniqueness constraint** the opposite is true: the entries in the fact table column corresponding to the right role need to be unique while the left column may contain duplicates. In the example schema from figure 1.3, this constraint on the fact type "Student(.ID) lives in Room(.nr)" means that each Student lives in at most one Room but multiple Students (n) may live in the same (1) Room.

The **m:n uniqueness constraint** indicates that the combination of entries in a row of the fact table must be unique, i.e. no two rows of the fact table shall be identical. The fact type "Student(.ID) attends Course(.code)" in figure 1.3 is subject to an m:n uniqueness constraint which specifies that a Student can attend several Courses (m) but it is also possible that several Students (n) attend the same Course.

The $m : n$ uniqueness constraint is always assumed and must be stated explicitly if none of the other intrapredicate uniqueness constraints apply to the predicate in question.

Intrapredicate uniqueness constraints can be applied to predicates of any arity and for a given predicate of arity n must span at least $n - 1$ of its roles. It is allowed to place multiple uniqueness constraints on the same predicate and they may overlap (but not completely). If no uniqueness constraints of length $n - 1$ apply to an n-ary predicate, then the uniqueness constraint spanning all n roles is required. Further examples for uniqueness constraints are shown in appendix A). Since verbalization is an integral part of ORM which facilitates the understanding of ORM schemas, NORMA provides verbalization of fact types (including constraints applying to them) in controlled natural language. For example, the m:n uniqueness constraint on the fact type "Student(.ID) attends Course(.code)." in figure 1.3 verbalizes in the following manner:

It is possible that some Student attends **more than one** Course **and that for some** Course, **more than one** Student attends **that** Course. **In each population of** Student attends Course, **each** Student, Course **combination occurs at most once.**

Interpredicate Uniqueness Constraints

Like intrapredicate uniqueness constraints, *interpredicate* or *external uniqueness constraints* make requirements to the uniqueness of objects populating roles, but now these roles are from *different* predicates [HM08, p. 128 ff]. The constraint is indicated by a circle containing a horizontal line which is connected to the role boxes it applies to by a dashed line. An example for this kind of constraint is shown in figure 1.4. In this case, the external uniqueness constraint expresses that it is not

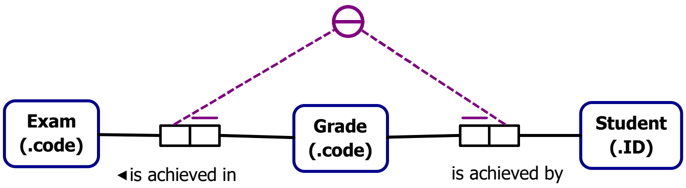


Figure 1.4: ORM schema showing an interpredicate uniqueness constraint roles from different predicates. It conveys that at most one Grade can be achieved by a certain Student in a certain Exam.

possible that two Grades are achieved by the same Student in the same Exam which verbalizes as:

For each Student **and** Exam, **at most one** Grade is achieved by **that** Student **and** is achieved in **that** Exam.

The interpredicate uniqueness constraint can connect more than two predicate roles.

Mandatory Role Constraint

The *mandatory role constraint* is used to specify which object types must play a certain role [HM08, p. 162 ff]. Each constraint applies to exactly one role (but may apply to multiple roles in one predicate) and entails that for each state of the database, each instance of an object type playing this role must play it. So, for a given object type o and its role r, a mandatory role constraint on r means that for each state of the database the following holds:

$$pop(r) = pop(o) \tag{1.1}$$

When a role is mandatory for an object type, this is symbolized by a filled circle that is either attached to the role box or to the object type that plays this role.

Figure 1.5 contains a mandatory role constraint on the role “was born on” which is connected to Student(.ID). This means that every instance of Student that is recorded in the database, must have a birthdate but does not need to have information on its Room or Courses stored.

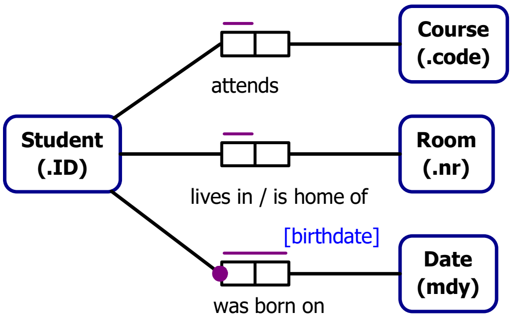


Figure 1.5: An ORM schema showing a mandatory role constraint. Whenever an instance of the entity type Student is created, the Date on which they were born must be recorded since the role “was born on” is mandatory.

As a general rule, any primitive object type that plays only one role in a global schema and any primitive entity type that plays only one fact role has a mandatory role constraint on this role. Due to this rule, the mandatory role constraint notation, i.e. the dot, can be omitted in these cases.

Disjunctive Mandatory Role Constraint

The *disjunctive mandatory role constraint* or *inclusive-or constraint* is applied to two or more roles and requires that at least one of these roles is played by instances of the associated object type [HM08, p. 168 ff]. It is represented as a circle containing a violet dot which is connected to the constrained roles through dashed lines.

Figure 1.6 shows an example for a disjunctive mandatory role constraint.

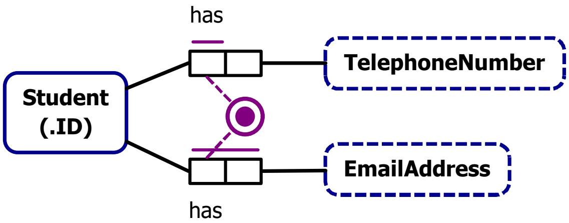


Figure 1.6: ORM schema showing a disjunctive mandatory role constraint. A Student must provide a TelephoneNumber, EmailAddress or both.

This schema specifies that a Student must provide TelephoneNumber or EmailAddress or both.

Value Constraint

A *value constraint* applies to a value type or a role and limits the values that these can take on [HM08, p. 216 ff]. The set of valid values is displayed in curly brackets which either contain an enumeration or the range of valid values.

In an enumeration the valid values are listed within brackets as in $\{'M','F'\}$ or $\{2, 4, 8, 16\}$. Ranges have an upper or lower bound or both and may contain ordinal values, such as positive or negative integers, real numbers or strings. For integer and string values ranges may take on the following forms: $\{1..10\}$, $\{1..\}$, $\{..-9\}$, $\{'June'..'December'\}$. When using real values, a square bracket next to the value indicates its exclusion from the range, e.g. $\{[1.05..5]\}$. It is also possible to insert ranges into an enumeration, e.g. $\{-9..4, 10..16, 24..86\}$ or $\{[-1..5], [6..9]\}$.

A value constraint on a value type is displayed by placing the set of permitted values either next to the constrained value type or next to the entity type that is referenced by this value type. A value constraint for a role is placed next to the role in question and provides the set of values that must be used to reference the entity type that plays this role.

Subset Constraint

The *subset constraint* is applied between two roles that are played by the same object type [HM08, p. 225]. It is used to indicate that the population of one role is a subset of another role. For two roles r_1 and r_2 and for each database state this means that $pop(r_1) \subseteq pop(r_2)$.

In an ORM schema this constraint is presented as a circle containing a subset symbol ($\subseteq$) which is connected to the subset role by a dashed line and to the superset role by a dashed line with an arrow. An example is shown in figure 1.7 where the instances of Students passing a Course are a subset of the Students attending the Course.

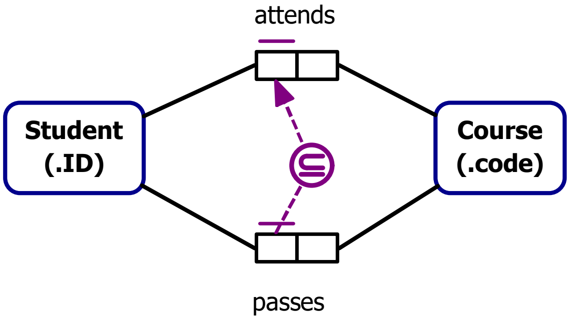


Figure 1.7: ORM schema showing a subset constraint between two roles. The instances of Students passing a Course are a subset of those attending it.

Equality Constraint

The *equality constraint* is used between two or more roles played by the same object type to ensure that the population of these roles is equal [HM08, p. 226 f]. If an equality constraint applies to n roles $\{r_1, r_2, ..., r_n\}$ and $n \in \mathbb{N}$, then the following must hold for each database state : $pop(r_1) = pop(r_2) = ... = pop(r_n)$.

This constraint is indicated by a circle containing an equality symbol (=) which is connected to the constrained roles through dashed lines. Figure 1.8 shows an example of this constraint. The instances of Students that have a Username (e.g. for the university's web platform) also have an EmailAddress (e.g. provided by the university after registering for the web platform).

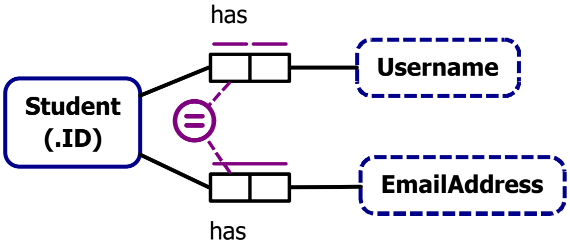


Figure 1.8: ORM schema showing an equality constraint between roles from different predicates. The instances of Students that have a Username are exactly the same that have an EmailAddress.

Exclusion Constraint

The *exclusion constraint* is used between two or more roles played by the same object type to ensure that the populations of these roles are mutually exclusive [HM08, p. 228]. If an exclusion constraint applies to n roles $\{r_1, r_2, ..., r_n\}$ and $n \in \mathbb{N}$, then the following must hold for each database state : $pop(r_1) \cap pop(r_2) \cap ... \cap pop(r_n) = \emptyset$.

In terms of notation, this constraint appears as a circle with a cross symbol (‘X’) which is connected to the constrained roles by dashed lines. An example for this constraint is displayed in figure 1.9 which shows that a Student can either live in a (dorm) Room or arrive at university by Car.

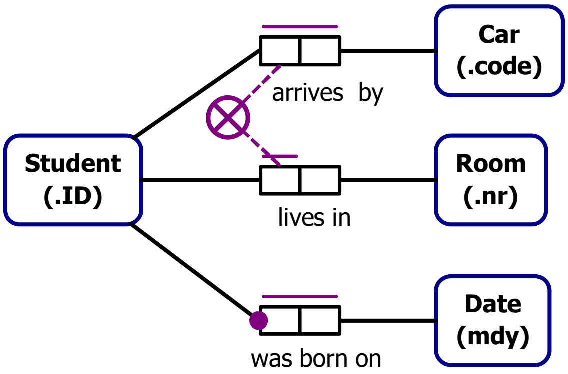


Figure 1.9: ORM schema showing an exclusion constraint between roles from different predicates. A Student may either live in a (dorm) Room or arrive (at university) by Car.

Exclusive-or Constraint

The exclusive-or constraint is a combination of the inclusive-or and exclusion constraint [HM08, p. 229]. Like these, it constrains two or more roles which are hosted by the same object type. It indicates that the object type’s instances play exactly one of the constrained roles.

In an ORM diagram this constraint appears as the overlay of the inclusive-or and exclusion constraint which can be seen in figure 1.10. This models that Students must either eat vegetarian food or non-vegetarian food when visiting university.

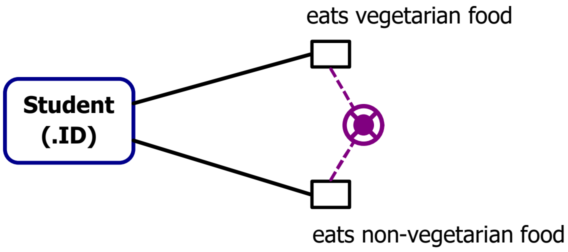


Figure 1.10: ORM schema showing an exclusive-or constraint between roles from different predicates. A Student must either eat vegetarian or non-vegetarian food.

The subset, equality, exclusion and exclusive-or constraints introduced previously restrict the populations of multiple roles played by the same object type. It is possible to extend the population restriction to a sequence of two or more roles using *tuple-subset*, *pair-equality* and *pair-exclusion constraints* [HM08, p. 232 f].

Frequency Constraint

ORM distinguished two kinds of frequency constraints on fact roles: *internal* and *external frequency constraints* [HM08, p. 272 ff]. Both are used to indicate that, for each state of the database, the population of the constrained fact role or set of fact roles must contain an object or tuple of objects a certain number of times. While internal frequency constraints apply to one or more roles within one predicate, the external frequency constraints applies to one role within two or more predicates. The requirement can either be in the form of a specific number (e.g. 5) or a range with an upper or lower bound or both ($\leq 5, \geq 5, 3..7$).

The internal frequency constraints can be further divided into *simple* and *compound frequency constraints*. The simple internal frequency constraint is used to specify how many times a single fact role should be played by an object. The required frequency is noted next to the constrained role.

This constraint can be extended to more than one role and is then called compound frequency constraint. It is used to specify how many times tuples of objects must play the constrained roles.

The external frequency constraint is used to constrain single fact roles from multiple predicates. In an ORM diagram this is noted by connecting the circled frequency or frequency range to the constrained roles by a dashed line.

Ring Constraints

In ORM, a *ring* is a binary relation between compatible object types. Constraints on this kind of relation are called *ring constraints* and define properties of the relation, such as *reflexivity*, *symmetry*, *asymmetry*, *transitivity*, and more. For a comprehensive list of all ring constraints, see appendix A or refer to [HM08, p. 278 ff]

Cardinality Constraints

Cardinality constraints can be applied to objects or roles in order to restrict the cardinality of their respective population [HM08, p. 289]. The constraint is noted as a "#" , followed by an expression indicating the allowed number or range (e.g. $\# = 2$, $\# \geq 2$).

Value Comparison Constraints

The *value comparison constraint* is used to compare the values of role instances using $<$, $\leq$, $>$, or $\geq$ operators [HM08, p. 290]. The values of the role instances must be compatible in order to allow comparisons.

Alethic and Deontic Constraints

In general, ORM distinguishes between two different kinds of constraints. The constraints introduced up to this point were *alethic* constraints [HM08, p. 408]. This means that these constraints must be met in every state of the information model. Any attempt to update an information model with data that violates alethic constraints is denied.

The second kind of constraints are so-called *deontic* constraints [HM08, p. 408 ff]. These constraints are not enforced and may be violated. If an update to an information model causes a conflict with this kind of constraint, the update is accepted and ideally the user should be informed that he violated a deontic constraint and should avoid this behavior in the future. Deontic constraints appear in blue rather than violet and mostly include an "o" which stands for "obligatory". Deontic ring constraints have a dashed line instead of a solid line.

If a constraint cannot be modeled using the concepts and notation introduced in this section, ORM allows the specification of textual rules in controlled natural language. The diagram elements involved in the constraint are marked with a footnote and the constraint rule is provided in a text box that appears in the diagram [HM08, p. 290 f].

This concludes the introduction of ORM constraints. Sections 1.5.3 through 1.5.5 introduce further concepts that can be used for concise modeling of the UoD, while section 1.5.6 will go into more detail on how entity types may be identified using *reference schemes*. Finally, section 1.5.7 will introduce the idea of *derivations* and conclude the introduction to ORM conceptual schemas.

1.5.3 Independent Object Types

An *independent object type* is a primitive object type which either hosts no fact roles or only optional fact roles [HM08, p. 219 ff]. This means that instances of these objects can exist which play no fact roles at all. This term is not used for value types that do not play any fact roles, or for subtypes. In an ORM diagram, independent object types are indicated by appending an exclamation mark ("!") to their name.

1.5.4 Objectifications

Objectification or *reification* describes the process of turning a relationship into an object [HM08, p. 88 f]. The type of object formed by this process is an *objectified association*. In an ORM schema, this is depicted as a named, soft rectangle placed around the predicate that is objectified. The name of the objectified association is stated in double quotes. The objectified association can participate in fact types in the role of an object type.

1.5.5 Subtyping

When an object type is further classified into more specific object types, this is called *subtyping* and the resulting specialized object types are called *subtypes* [HM08, p. 238 ff]. The object type from which the subtype originated is referred to as the *supertype*.

In ORM it is possible for one subtype to have two or more supertypes, which is known as multiple inheritance. Furthermore, one supertype can have many subtypes and may have instances that are not instances of any of its subtypes. Given a supertype A and its subtype B, the following expression must be satisfied for each state of the database:

$$A \neq B \wedge pop(B) \subseteq pop(A) \tag{1.2}$$

B is then called *proper subtype* of A and A is the *proper supertype* of B.

In an ORM schema, a line between two object types with an arrow at its end indicates that the object type at its origin is the subtype of the object type (the supertype) the arrow points to.

Figure 1.11 shows three constraints that can be applied to the population of subtypes of one supertype. Figure 1.11 a) shows an exclusion constraint between the subtypes which means that the intersection between Seminar and Lecture is empty (but other Courses may exist). In figure 1.11 b) there is a disjunctive mandatory role constraint between the subtypes, meaning that each Student must be a Bachelor Student, Master Student or PhD Student or any combination thereof. There can be no instances of Student that aren't instances of one or more of its subtypes. Figure 1.11 c) displays an exclusive-or constraint between the subtypes meaning that each instance of Student must be either a Commuting Student or a Campus-dwelling Student.

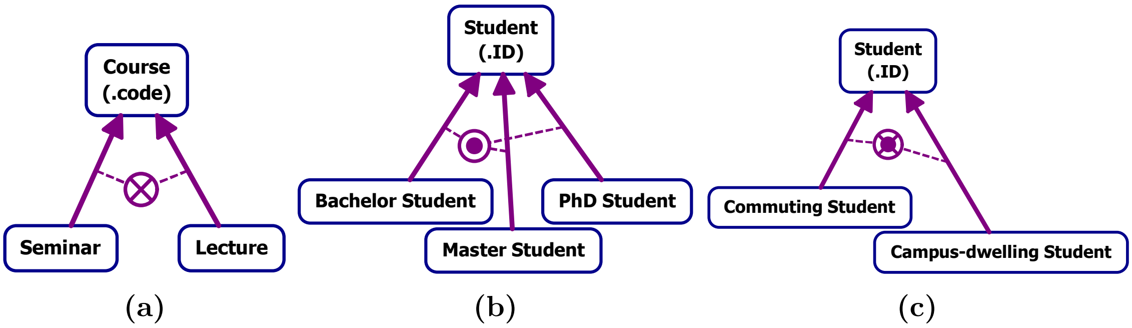


Figure 1.11: Figure a) displays exclusive subtypes, figure b) shows exhaustive subtypes and figure c) represents a partition of the supertype.

1.5.6 Reference Schemes

In ORM, each entity type must have a preferred reference scheme which indicates the manner in which entities can be referred to by values. In ORM, entities can be identified by values from both *semantic* and *syntactic* domains.

Two different reference schemes can be distinguished: in a *simple* or 1 : 1 *reference scheme*, each entity can be identified by exactly one value. In a *compound* reference scheme the identification of an entity takes two or more values [HM08, ch. 5.3].

Simple Reference Scheme

If an entity is identified by exactly one value, it has a simple 1 : 1 reference scheme. The reference mode provides information on how an entity is related to its value and can be abbreviated by placing the value type in parentheses after the entity type's name. A reference mode is always the *preferred* reference scheme for the entity type. This kind of reference scheme is shown in figure 1.3 where each Student is uniquely identified by his/her StudentName. This means that the UoD cannot contain more than one Student who is called 'Sophie'. Any other facts recorded about 'Sophie' in this UoD (e.g. that she owns a Car) must refer to the same Student 'Sophie'.

NORMA distinguishes three kinds of reference modes: *popular reference modes*, *unit-based reference modes* and *general reference modes.*

The most often used reference modes are listed as **popular reference modes** in NORMA and include *name*, *code*, *id*, *title*, *#*. If popular reference modes are parenthesized, they are preceded by a dot, e.g. Student(.name). The value type which is represented by the reference mode has a name which is a combination of the entity type name followed by the reference mode name starting with a capital letter. So, "Student(.name)" has the value type "StudentName".

Unit-based reference modes are also included in NORMA in the form of a list which contains physical and monetary units. If unit-based reference modes are parenthesized, the unit is followed by a colon, e.g. Weight(kg:). The value type is derived from only the reference mode by appending the string 'Value', i.e. Weight(kg:) has the value type kgValue.

General reference modes don't have additional punctuation when parenthesized and their value type is identical with their names, e.g. Book(ISBN) has the value type ISBN.

Each entity type has a *preferred* reference scheme. If a simple 1 : 1 reference mode exists, this is the preferred reference scheme. The reference mode can also be formulated explicitly as seen in figure 1.12. The double uniqueness constraint is used to indicate that this is the preferred reference mode for the entity types Student and Room.

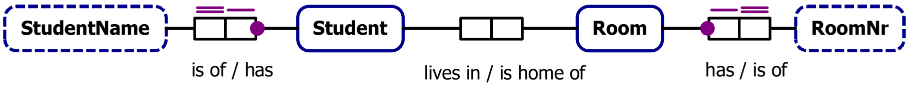


Figure 1.12: ORM schema explicitly showing the preferred reference mode for the entity types Student and Room. The double uniqueness constraint is used to indicate that this is the preferred reference mode which necessitates the mandatory role constraints.

Compound Reference Scheme

If the definite identification of an entity requires the use of two or more values, this is called a *compound reference scheme.* If it is the primary source of identification, it is called a *preferred* compound reference scheme. When an entity type is identified by two or more of its roles, these roles must be included in an interpredicate uniqueness constraint. If the thus defined reference scheme is the preferred reference scheme for the entity type in question, the circle indicating the constraint has two horizontal lines. An example for an interpredicate uniqueness constraint defining a preferred reference scheme for an entity type is displayed in figure 1.13.

In the modeled UoD, a Student can be uniquely identified by the combination of his StudentName and the Date he was born on. By placing the double bar in the interpredicate uniqueness constraint, it is clear that this is the preferred reference scheme for this entity type. Note that the interpredicate uniqueness constraint defining Student's preferred reference scheme affects two different kinds of roles. One is played by the value type StudentName but the second is played by the entity type Date. Instances of StudentName are values while instances of Date are entities - in turn identified by their reference mode "mdy" which represents a specific date format and is most likely a string value. In conclusion, Student's preferred identification is through the value of StudentName and the value of Date's preferred identifier.

Additionally, note that Student has a second, simple reference scheme in which a Student is identified by its StudentID.

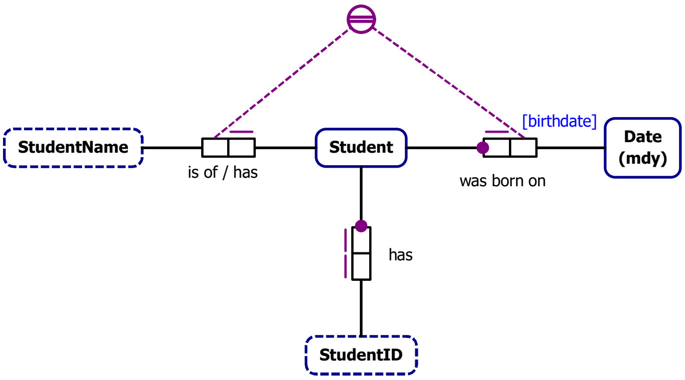


Figure 1.13: ORM schema showing a preferred compound reference scheme for the entity type Tournament. The interpredicate uniqueness constraint with the double bar implies that Tournament can be uniquely identified by the host City and the Tournament's Date and that this is the preferred way to reference Tournament.

1.5.7 Derivations and semiderivations

In ORM, fact types are either *asserted*, *derived* or *semiderived* [HM08, p. 98 ff]. A fact type is asserted or *primitive* if it is not defined in terms of other fact types. Its population consists of asserted facts.

A fact type is derived, if it is defined in terms of other fact types (asserted or derived). A derived fact type is marked with an asterisk “*” appended to its reading. The derivation rule can be supplied in textual form within the diagram and is preceded by an asterisk. Whenever the derived fact type is queried, the derived information is computed (derived-on-query). An example is shown in figure 1.14. Using this schema, the value for the value type “NrCourses” is deduced by counting the number of courses a student attends instead of requiring the user to explicitly state this number for each student. This reduces the likelihood of errors and inconsistencies.

A double asterisk “**” indicates that a fact type is derived and the derived information is stored in the information system which reduces response time (derived-on-update).

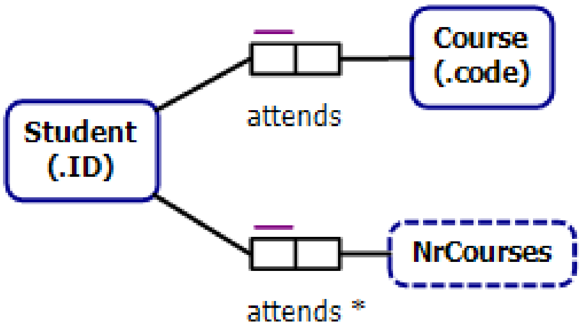


Figure 1.14: This ORM schema shows the derived fact type “Student attends NrCourses”. This is indicated by the asterisk appended to the predicate “attends”. The derivation text isn’t displayed in the ORM schema, but reads as *Student attends NrCourses **if and only if that** Student attends **some** Course **where** NrCourses = count(**each** Course **for that** Student)., i.e. it counts the number of Courses each Student attends and stores the result in “NrCourses”.

A fact type is semiderived, if its population contains asserted and derived facts. Semiderived fact types are marked with a “+” appended to their reading. The derivation rule can be stated in textual form within the diagram with a leading “+” symbol. An example is shown in figure 1.15. Here the fact type “Person is grandparent of Person” is a semiderived fact. One the one hand this information can be derive. In cases where a person A is a parent of a person B and this person B is a parent of a person C, it is derived that person A is a grandparent of person C. But on the other hand it is possible to explicitly state that some person is a grandparent of another person without necessarily constructing the parenting hierarchy.

Just as fact types, subtypes can also be asserted, derived or semiderived. A derivation is denoted by “*” and a semiderivation is denoted by “+” following the sub-

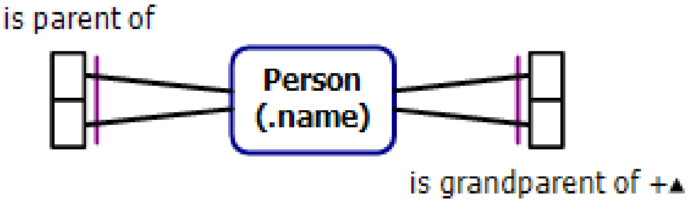


Figure 1.15: This ORM schema displays the semiderived fact type "Person is grandparent of Person". This is indicated by the "+" appended to the predicate reading of this fact type. The derivation rule isn't displayed in the ORM schema, but reads as +Person1 is grandparent of Person3 **if and only if** that Person1 is parent of **some** Person2 **and that** Person2 is parent of **that** Person3.

type's name. The derivation rule can be supplied in the diagram as with (semi)derived fact types.

1.6 TGraphs

TGraphs were introduced by Ebert et al. in [EF95] as a very general class of graphs. These graphs can be used as conceptual models, formal mathematical structures and efficient data structures.

The basic elements of TGraphs are *vertices* and *edges*. TGraphs are

directed, i.e. each edge has a start and an end vertex also referred to as `alpha` and `omega` of the edge,

typed, i.e. vertices and edges can be divided into distinct classes,

attributed, i.e. both vertices and edges can have attribute-value pairs attached to convey additional information,

ordered, i.e. edges, vertices and edges incident to a vertex have a consistent ordering,

graphs.

Figure 1.16 shows an example of a TGraph which shows two vertices of type `Student` and one vertex each of the type `Seminar` and `Lecture`. The vertices are connected via edges of the type `StudentAttendsCourse`. These edges are directed, starting at instances of `Student` and ending at instances of `Seminar` or `Lecture`.

The vertices are attributed: `Seminar` and `Lecture` have an attribute `courseCode` and `Student` has the attributes `name` and `studentID` amongst others.

The edges and vertices are ordered. `e1` is the first and `e3` is the last edge. The vertices are ordered beginning with `v1` and ending with `v4`. The ordering if the edges incident to a vertex class is indicated by the numbers at the beginning and end of edges: for e.g. `v3` `e1` is the first and `e2` is the second incidence.

Which types of vertexes and edges and which attributes are allowed in a certain class of TGraphs can be defined in a TGraph schema. This schema can be provided using the modeling language *graph UML*, short *grUML* [BHR$^+$10], which is a profiled version of UML 2 class diagrams.

1.7 grUML

Graph UML (*grUML*) is a profiled version of UML 2 class diagrams and can be used to define TGraph classes.

A grUML diagram, also called a *grUML schema*, defines the structure and constraints that apply to instances of the thus defined TGraph class.

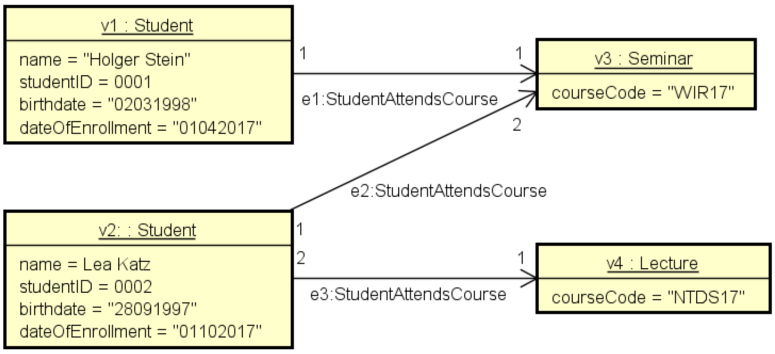


Figure 1.16: An example TGraph showing two vertices of type `Student` (`v1`,`v2`) and one each of the type `Seminar` (`v3`) and `Lecture` (`v4`). The vertices are attributed and ordered. The edges are directed, ordered and of type `StudentAttendsCourse`. Edges incident to a vertex are also ordered.

Unlike UML 2 class diagrams, grUML classes may not contain method declarations. The main components of a grUML schema and an example grUML schema, which defines the TGraph in figure 1.16, will be introduced in the following sections.

1.7.1 Main components

The main components of a grUML diagram and their meaning for the definition of a TGraph class are as follows:

classes Each class (with the exception of association classes) within a grUML diagram represents a vertex class that may be used in the defined TGraph.

The attributes of a grUML class will map to attributes of the graph class in the TGraph. Classes in grUML diagrams don't have method declarations.

Generalizations between classes of the grUML diagram translate into generalizations between the defined TGraph classes.

Abstract classes in the grUML diagram (indicated by classifier's name written in italic or the use of the stereotype ⟨⟨abstract⟩⟩) mean that the set of instances of this class in the TGraph will be empty.

Multiple inheritance is allowed in grUML diagrams.

associations An association in a grUML diagram represents an edge class in the TGraph class. The source and target of an association define the direction of the edge class in the TGraph class. Aggregations (composite or shared) define composition or aggregation classes at the TGraph level.

association classes An association class in a grUML diagram also represents an edge class within a TGraph. Abstract association classes map to empty edge class instances in a TGraph. Generalizations between association classes in the grUML diagram are represented as associations between edge classes in a TGraph.

attributes Attributes of classes in a grUML diagram are mapped to attributes in the corresponding classes of a TGraph. There is a predefined set of attribute domains in grUML.

domains attributes can be from the following domains: boolean, integer, long, double or string values; lists or sets of any of the previous value types; maps where keys and values can be from an arbitrary domain; enumerations of values defined by the schema designer; records with values from arbitrary domains.

stereotype Four stereotypes are used in grUML schemas:

1. ⟨⟨**abstract**⟩⟩ The stereotype ⟨⟨abstract⟩⟩ is used to specify that a vertex or edge class is abstract. The stereotype can be used instead of the italicized class name to indicate this circumstance.
2. ⟨⟨**enumeration**⟩⟩ The stereotype ⟨⟨enumeration⟩⟩ is used in the same way it is in UML 2. In grUML diagrams its purpose is the definition of an enumeration domain. The values listed in the enumeration type need to adhere to the rules for Java enumeration identifiers, i.e. may only consist of uppercase letters, numbers and underscores.
3. ⟨⟨**record**⟩⟩ The stereotype ⟨⟨record⟩⟩ is used in classes in grUML diagrams to define a record domain. The attributes of this class define the components of the record. The components can be from any valid grUML domain, including nested records, but these may not contain any cyclic dependencies.
4. ⟨⟨**graphclass**⟩⟩ a grUML diagram requires the existence of exactly one class with the ⟨⟨graphclass⟩⟩ stereotype. This class contains the name and the attributes of the TGraph class defined by the schema. The stereotyped class may not have any associations to other classes within the schema.

packages Each grUML schema contains exactly one default package which has an empty name. If vertex or edge classes or the valid grUML domains are not contained in any other package, they are automatically in the default package.

constraint Since UML only provides limited support for a diagrammatic representation of constraints, grUML uses GReQL, the *Graph Repository Query Language* [KK01], to specify them. The following section will provide a brief introduction to GReQL. Within grUML schemas, the constraints can be attached to vertex classes, edge classes or the graph class itself. A constraint consists of three parts: a textual description of the constraint in natural language; a GReQL expression that evaluates to the boolean `true` if the expression holds and `false` if not; an optional GReQL query that can be used to retrieve a set of elements that conflict with the constraint.

comments Comments can be attached to the graph class, vertex classes, associations, domains and packages. Comments that don't have a specific link are automatically assigned to the graph class.

1.7.2 grUML metamodel

The grUML metamodel, which formally defines the elements of a grUML schema and their interactions, is shown in Figures 1.17 and 1.18.

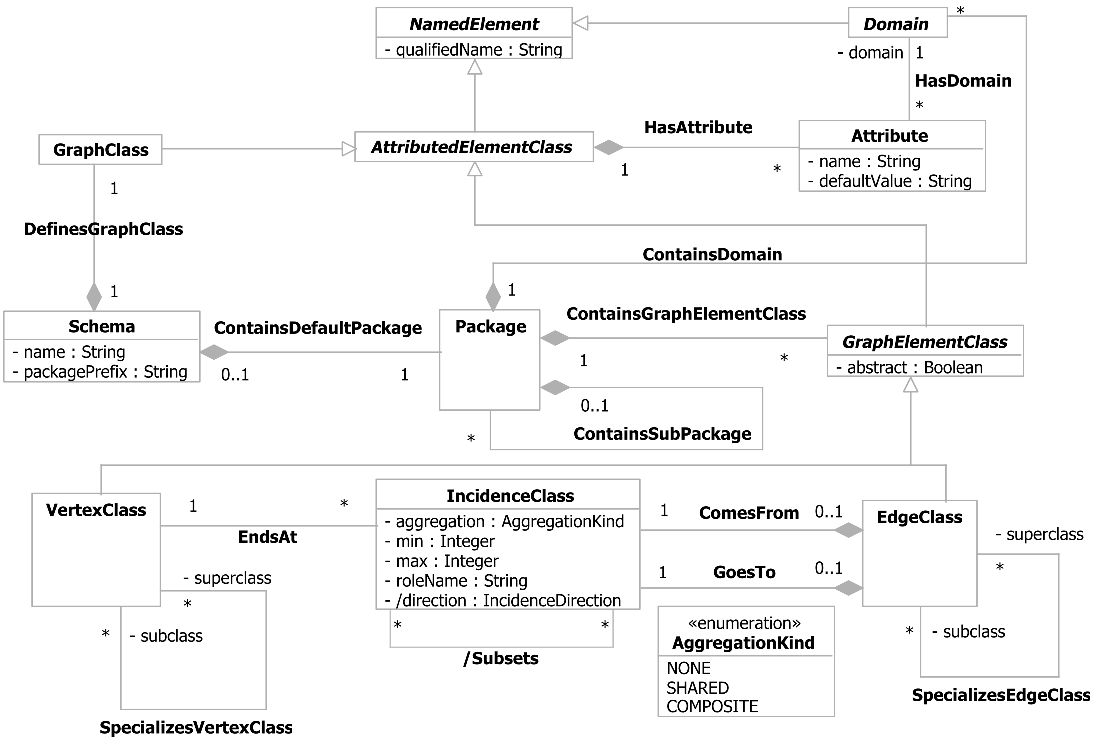


Figure 1.17: The simplified grUML metamodel.

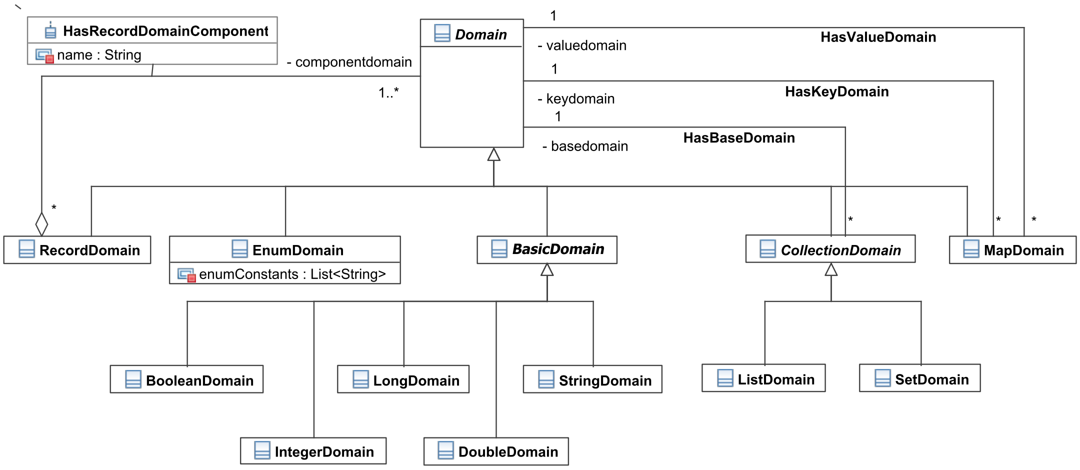


Figure 1.18: The grUML domain metamodel.

1.7.3 Example grUML schema

Figure 1.19 shows the grUML schema that defines the TGraph in figure 1.16 on page 30. The vertex class `Student` has four attributes from the grUML domains Integer and String. The edge class `StudentAttendsCourse` is directed from `Student` to the vertex class `Course`. `Course` has an attribute `courseCode` and is the superclass of the two vertex classes `Seminar` and `Lecture`. As subclasses of `Course`, they inherit the attribute `courseCode` and the ability to form connections to `Student` via the edge class `StudentAttendsCourse`. The multiplicity 1..* at the source of the edge class relates that an instance of `Course` must be associated with at least one instance of `Student` and the multiplicity * at the end of the edge class indicates that an instance of `Student` does not have to have a connection to an instance of `Course`.

Looking at the TGraph in figure 1.16 (p. 30), there are no violations in terms of multiplicity. Each instance of `Course` is connected to an instance of `Student` by an edge. The instances of `Seminar` and `Lecture`, `v3` and `v4`, inherit the attribute `courseCode` from their parent vertex class `Course`.

1.7.4 Validating grUML constraints

For this work, generating grUML constraints that correspond to the constraints defined in ORM will be one of the most important and most challenging tasks. As mentioned at the beginning of this section on grUML, it is possible to define grUML constraints within grUML schemas. Although they are added at the schema level,

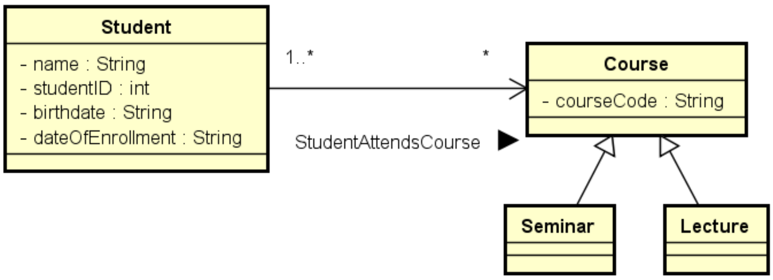


Figure 1.19: An example grUML schema. It defines the vertex class `Student` with various attributes. It also defines the vertex class `Course` with the attribute `courseCode` and its child vertex classes `Seminar` and `Lecture`. Lastly it connects the two vertex classes via the directed edge class `StudentAttendsCourse` and further defines the number of vertices that can be linked through the multiplicity values at each end (1..* and *).

they actually apply to TGraphs, i.e. instance level. In order to check whether a TGraph is in accordance with the constraints defined in its schema, JGraLab provides the class `GraphValidator.java`. Its constructor takes a TGraph as argument and subsequently applying the `validate` method will result in the evaluation of the constraints applying to this TGraph. Additionally, this method also checks that there are no violations of the multiplicity values defined in the schema. The `validate` method returns a sorted set of `ConstraintViolations` which provide further information on the kind of violation and the elements which are in violation of the constraint.

1.8 GReQL

After defining a TGraph schema using grUML and generating a TGraph instance, the graph querying language GReQL [KK01][EB10] can be used to query this graph and store the results. GReQL (currently in version 2) is designed to extract information from TGraphs but doesn't allow graph manipulation. Each GReQL query can contain expressions of various kinds: variable declarations and definitions, GReQL functions, the selection of edges and vertices of certain types, and many more.

For this work, GReQL expressions are necessary in order to define grUML constraints. In this context, the most important expressions are *quantified*, *conditional* and *FWR* (*from-with-report*) expressions. These will be introduced in the following sections.

1.8.1 Quantified expressions

In GReQL, the quantified expression is used to check whether a certain amount of element(s) within a defined set of elements satisfy an expression or not. The general structure of a quantified expression is as follows:

```
<quantifier> <variable declaration> @ <expression>
```

GReQL provides three different quantifiers to use in quantified expressions: `forall`, `exists` and `exists!`. Depending on these quantifiers, these kinds of expressions check whether all (`forall`), at least one (`exists`) or exactly one (`exists!`) element(s) fulfill the expression stated after the `@`. Quantified expressions return boolean values as result.

Listing 1.1 shows an example for a quantified expression. This expression can be applied to any TGraph instantiating the grUML schema in figure 1.19. Following the quantifier `forall`, the variable declaration `s:V{Student}` defines a variable `s` which represents an instance of a vertex class (`V`) of type `Student`. Finally the expression after the `@` symbol checks, whether for the variable `s` the value of the attribute `birthdate` is smaller than the value of the attribute `dateOfEnrollment`. In conclusion, the quantified expression in listing 1.1 evaluates, whether for each instance of the vertex class `Student` the value of `birthdate` is smaller than the value of`dateOfEnrollment`. If this query were evaluated on the TGraph from figure 1.16, the return value would be `true` ("02031998" < "01042017" and "28091997" < "01102017").

Listing 1.1: A quantified GReQL expression.

```
forall s:V{Student} @ s.birthdate < s.dateOfEnrollment
```

Throughout this thesis, quantified expressions are often used in grUML constraints to guarantee that all instances (`forall`) of a specific vertex or edge class satisfy a certain condition.

1.8.2 Conditional expressions

Conditional expressions are structured in the following manner:

```
<expression> ? <expression 1> : <expression 2>
```

If the `expression` before the question mark (`?`) returns `true`, `expression 1` is evaluated next; if it returns `false`, `expression 2` is processed. The return value for the entire expression is `true` or `false`.

Listing 1.2 shows an example for a conditional expression. In this case, the conditional expression is nested within a quantified `forall` expression. The `forall` expression validates whether, for every instance `e` of an edge class (`E`) of type `StudentAttendsCourse`, the following conditional expression holds true.

The conditional expression starts with `omega(e).courseCode = ``WIR17''`. This checks whether an instance `e` of the edge class `StudentAttendsCourse` has a target vertex (`omega`) which has the attribute `courseCode` with the value “WIR17”. If this is the case, i.e. the condition evaluates to true, it continues by evaluating whether this edge `e`’s source vertex has the attribute `dateOfEnrollment` with a value which is smaller than “01102017”. If the expression preceding the question mark returns `false`, the conditional expression will return `true`.

If the expression from listing 1.2 is evaluated on the TGraph from figure 1.16, it will return `false`, since `e2`’s target vertex (`v3`) has the attribute `courseCode` with the value “WIR17” but its source vertex’ (`v2`) attribute `dateOfEnrollment` is “01102017” which is not smaller than “01102017”.

Listing 1.2: A conditional GReQL expression nested in a forall expression.

```
forall e:E{StudentAttendsCourse} @
omega(e).courseCode = ``WIR17'' ?
alpha(e).dateOfEnrollment < ``01102017'' : true
```

During this work, conditional expressions are used in grUML constraints if a constraint has a precondition that needs to be met. In the example above, the precondition is that the target vertex of the edge defined in `e` has the attribute `courseCode` with the value “WIR17”.

1.8.3 FWR expressions

Within a grUML constraint, FWR expressions can be used to retrieve elements which are conflicting with the GReQL expression defined in the constraint. FWR have the following general structure:

```
from <variable declaration>
with <expression>
report <output definition>
end
```

Listing 1.3 shows an example for an FWR expression. It can be used to extract the elements which violate the condition formulated in listing 1.2.

Listing 1.3: A from-with-report GReQL expression.

```
1 from e:E{StudentAttendsCourse}
2 with omega(e).courseCode = ``WIR17'' and
3 not(alpha(e).dateOfEnrollment < ``01102017'')
4 report e
5 end
```

`from` clause

The `from` clause is the first part of an FWR expression and is used to declare variables and their domains for use in the `with` and `report` clauses. Variable names are typically stated in camel case. Multiple variables can be declared in a single step if provided as a comma separated list. The variables can be from various domains: string, boolean, signed integer or float values, or edges and vertices of a certain type. The type is appended to the domain name in curly brackets.

During the evaluation of the FWR expression, the variables declared in the `from` clause will be bound to all possible values, forming all possible combinations of values.

In listing 1.3, line 1 is used to define the variable `e` representing an edge (`E`) of type `StudentAttendsCourse`.

`with` clause

The `with` clause is an optional part of the FWR expression. It is used to define constraints on variable combinations generated in the `from` clause. These constraints appear as predicates and can take on various forms which must evaluate to `true`, `false` or `null`.

In listing 1.3, lines 2-3 define an expression which is evaluated for all values of `e`. So for each `e` it evaluates whether the edge's target has the attribute `courseCode` with value “WIR17” and the source vertex' attribute `dateOfEnrollment` is not smaller “01102017”.

`report` clause

The final part of the FWR expression is the `report` clause. This part is used to set up the output format for the query. It is possible to fill tables, sets, maps or bags with the elements which violate the expression defined in the `with` clause.

In listing 1.3, line 4 defines which values to return (the instances of `e`) and in which form (all instances are returned). If this GReQL expression were applied to the TGraph from figure 1.16, it would return the edge `e2:StudentAttendsCourse`.

During this work, the elements returned from FWR queries are typically collected in a set by using the keyword `reportSet` instead of `report`.

The FWR expression is completed by the statement `end`.

1.9 Event-Condition-Action (ECA) rules

JGraLab provides the possibility of defining *Event-Condition-Action rules* which can be added to TGraphs.

These rules define an *action* that is performed if a certain *event* takes place. Additionally, the rule may contain a *condition* which is checked if the defined event has taken place and the action is only performed if the outcome of the check is positive. Examples for predefined events are changing, creating or deleting edges or vertices in a TGraph or changing an attribute value. The three components - event, condition, action - are combined in `ECARule`s which are collected in an `ECARuleManager`. This manager listens for changes in the TGraph and fires events in response to the kinds of changes specified above.

In comparison to grUML constraints, `ECARule`s are added to a TGraph, so are added at instance level rather than schema level. They provide the possibility of reacting to changes in the TGraph directly and thus could be used to realize derivations and semiderivations in TGraphs. They can also be used to check whether an updated attribute value is valid, e.g. if it is required to be unique.

1.10 Summary

Figure 1.20 provides an overview of the concepts introduced in this chapter. Both grUML and ORM are schema languages. While ORM already includes a large number of constraints, grUML provides the possibility of defining grUML constraints and adding them to the schema. These grUML constraints are formulated in GReQL rendering them very flexible. ORM instances are commonly represented in the form of relational tables (not pictured in the overview figure). The instances of grUML schemas are TGraphs - ordered, attributed, typed and directed graphs. ECA rules can be added to TGraphs in order to respond to graph changes with user specified actions.

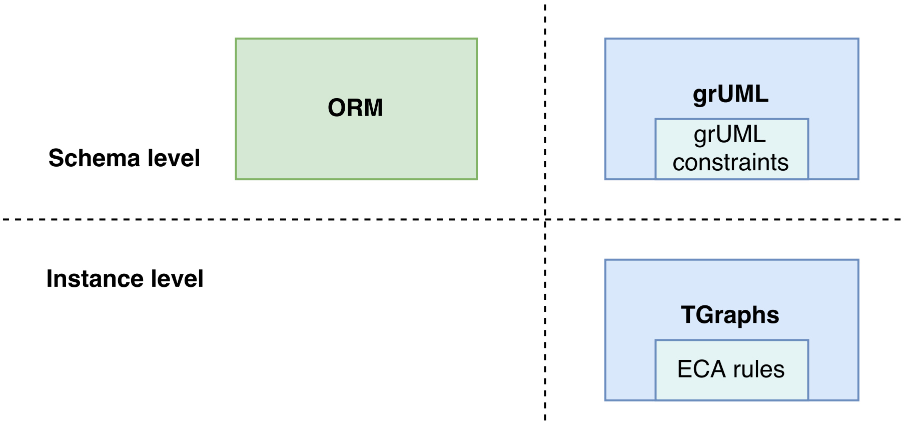


Figure 1.20: An overview of the main concepts used in this work: ORM, TGraphs and grUML. The schema language grUML can be extended by using gruML constraints formulated in the TGraph query language GReQL. GrUML schema instances are TGraphs, which can contain ECA rules to react to graph changes. The ORM schema language containt numerous constraints.

2 Mapping ORM to grUML

This chapter introduces the reader to the conceptual mapping that was defined in order to transform an ORM schema into a grUML schema. Initially, the reader will be familiarized with some general considerations about mapping schemas from ORM to grUML. This is followed by a step-by-step approach in which for each ORM element or group of elements, its correspondence in grUML is presented along with the considerations that led to the mapping. Finally this chapter will conclude with a summary of the mapping in form of a table that lists the ORM elements and their proposed representation in grUML.

Although the naming of the mapped elements is actually an *implementation* concern and doesn't play a role at the conceptual mapping level, the grUML naming conventions and the details on how names for grUML elements were constructed, will be introduced within this chapter in order to have a consistent naming scheme for the mapping examples. This mapping attempts to transfer all of ORM's constructs to grUML while largely maintaining their semantics by choosing adequate counterparts in grUML. The goal is to define a complete mapping by considering all possible combinations of ORM components. Details on the implementation of this mapping can be found in the following chapter 3.

2.1 Prerequisites

In order for the mapping from ORM to grUML to work, the *global*, i.e. full, ORM schema must be available. Mapping separate parts of one ORM schema to grUML and adding these parts together may lead to a different result compared to mapping the entire ORM schema. It could also cause conflicts, since role, vertex class and edge class names must be unique within a grUML schema.

Furthermore, it is assumed that the ORM schemas provided for the mapping procedure are semantically correct.

2.2 Naming conventions in grUML

The names of the elements mapped from an ORM schema to a grUML schema must be compliant with grUML's naming conventions. The conventions relevant to the mapping procedure presented in this work are detailed below.

Edge class names While predicate readings in ORM schemas are generally written in lower case and may consist of multiple words separated by whitespaces, grUML expects the association or edge class names to begin with an upper case letter (A-Z) and contain only alphanumeric characters[1]. Each edge class must have a name.

Vertex class names The naming conventions applying to the edge class names also apply to vertex class names. Each vertex class must have a name.

Attribute names Attribute names in grUML are required to begin with a lower case letter (a-z) and consist of only alphanumeric characters [1]. Attribute names may not be empty.

In addition, edge class and vertex class names need to be unique within a grUML schema and attribute names must be unique within the vertex or edge class they are associated with.

2.3 Entity Type mapping

ORM element: Entity type

Corresponding grUML element: Vertex class

Description: ORM entity types are the abstraction from *entities* which are concrete objects that are uniquely identifiable through their relations to other objects. Entity types provide a way of grouping concrete objects based on a commonality, so "Spain" and "Italy" may be instances of an entity type "Country" or "EUMember". Due to these semantics the grUML counterpart for ORM's entity types are *vertex classes*. Vertex classes in grUML represent concepts. Therefore, ORM entity types are always mapped to vertex classes in a grUML schema.

[1] (a-z,A-Z,_,0-9)

Name: The name of the vertex class is generated from the name of the ORM entity type. ORM entity type names must be unique within an ORM schema, so this requirement for vertex class names is met automatically. All characters which are not allowed in a grUML vertex class name are removed from the entity type name and its first letter is transformed to upper case to create the name of the vertex class.

Reference scheme: Every entity type in ORM must be uniquely identifiable by one or multiple values which are its *preferred identifiers*. The reference scheme defines the connection between the entity type and the value type(s) whose instances identify the entity. The mapping of 1:1 preferred reference schemes or *reference modes* specifically and reference schemes in general is detailed in sections 2.4 and 2.11 respectively. In brief, each vertex class receives an attribute “preferredIdentifiers” which stands for a record. Within this record there are components that represent the value types necessary to uniquely identify an entity.

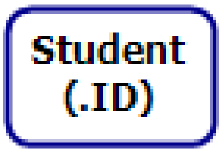


Figure 2.1: An entity type from an ORM schema.

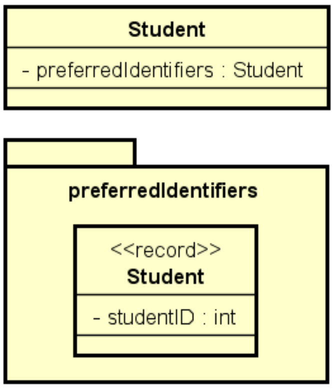


Figure 2.2: The mapping of the ORM entity type “Student” on the left to a grUML vertex class “Student”.

Example: The mapping of the ORM entity type “Student” to a grUML vertex class “Student” is shown in figures 2.1 and 2.2. The reference mode “ID” of the entity type “Student” maps to the component “studentID” from the domain Integer. This component is stored as an attribute in a class with the stereotype ⟨⟨record⟩⟩ that has the same name as the vertex class. The vertex class “Student” has an attribute “preferredIdentifiers” which links to this record. The record is placed in a package “preferredIdentifiers” in order to avoid duplicate class names and better organize the grUML schema. The specifics on the mapping of reference modes will be introduced in the following section.

Exceptions -

2.4 Reference Mode mapping

ORM element: Reference mode of an entity type

Corresponding grUML element: Each grUML vertex class has an attribute "preferredIdentifiers" which is from the record domain and contains the reference mode name and domain from the ORM entity type which is represented by this vertex class.

Description: The reference mode is an example of a simple or 1:1 preferred reference scheme and is used to identify entities in ORM. The mapping of more complex reference schemes will be discussed in section 2.11. The reference mode of an entity type is an abbreviation for a mandatory 1:1 relation between the entity type and the preferred value type used to identify it (recall Figure 1.12 on page 24). The role of the entity type is mandatory, since each entity must be uniquely identifiable and the relation to a value is the only way to achieve this.

Each entity type must have at least one unique identifier, and since the identifier is a value, it will map to an attribute of the vertex class generated from the entity type. But how can an attribute that represents a unique identifier for a vertex class be distinguished from its other attributes?

This mapping proposes that each vertex class originating from an entity type's mapping receives an attribute "preferredIdentifiers" which is used to collect the unique identifier(s) for this entity type. The attribute "preferredIdentifiers" specifies a value from grUML's record domain. In the grUML schema, the record is represented as a class with the stereotype ⟨⟨record⟩⟩. It has the same name as the vertex class and is placed into the package "preferredIdentifiers" within the grUML schema for a clearer schema structure. As explained previously, each reference mode can be expanded to a relation between an entity type and its value type. The ⟨⟨record⟩⟩ class' attributes represent the components of the record. Each component's name is generated from the name of the value types that uniquely identify the mapped entity type and the component domains are derived from their domains.

Every reference mode, which actually resembles a value type, comes with a predefined domain. In appendix B table 4 displays ORM's reference modes, their value domains and the mapping of these domains to grUML domains. In the case of the reference mode "ID", which is from the ORM domain "Numeric: Auto Counter" the mapping is to the grUML domain "Integer".

Name: The name of a record domain component is generated from the reference mode name and the name of the entity type which is referenced by it. The prefix of the name is the entity type name (first letter is transformed to lower case), followed by the name of the reference mode (its first letter is transformed to upper case). So the reference mode "ID" of an entity type "Student" is mapped to a record domain component named "studentID".

Two reference modes in ORM require renaming because their names are not valid grUML names: "#" and "%". "#" is named "No" in grUML and "%" is named "Percent".

Example: Figures 2.1 and 2.2 show an example of an entity type "Student" with the reference mode "ID" which maps to the attribute "StudentID" from the domain "Integer". The preferred identifier is stored in the record domain "Student" in the package "preferredIdentifiers".

Note: The package "preferredIdentifiers" that contains the record domains for mapped entity types (and objectifications, see 2.6) will be omitted in most further example grUML schemas in this chapter in order to keep the schemas more compact. Keep in mind, that the attribute "preferredIdentifier" does however always map to this package even if it is not displayed.

Exceptions: -

2.5 Value Type mapping

ORM element: Value types

Corresponding grUML element: Attribute of a vertex class

Description: ORM value types are used to identify or further define entity types. This is realized through relations between value types and entity types. An entity type that has a relation to a value type in ORM maps to a vertex class (corresponding to the entity type) that contains an attribute (corresponding to the related value type).

The reason for this decision is that value types in ORM primarily define values and their domains. This corresponds to the concept of attributes in UML and concordantly in grUML.

Mapping value types to vertex classes is a bad practice since the class would hold just one attribute representing the actual value and no further information. Besides defining the domain of values, ORM value types are also intended to represent semantic domains, i.e. it should not be possible to compare values from the value types “Length” and “Height” because the values have different semantics. This cannot be represented when mapping value types to attributes in grUML. However, this property would equally be lost if value types mapped to vertex classes. Although semantic domains can be modeled on schema level, their correct handling is ultimately a task to be performed at the implementation level.

Name: The name of the attribute generated from the ORM value type varies, depending on whether the value type is a preferred identifier of the entity type (or objectification) it is related to or not.

If the value type is *not* a preferred identifier, the attribute name is generated as follows: all characters which are not allowed in a grUML attribute name are removed from the value type name and its first letter is transformed to lower case. If an ORM object type has multiple relations with the same value type, the attribute name generated from this value type is qualified with an integer value since duplicate attribute names aren’t allowed in grUML.

If the value type is a preferred identifier of the related entity type or objectification, the vertex class representing the latter receives an attribute “preferredIdentifiers” from the grUML domain record. In analogy to the reference mode mapping in the previous section the attribute name then consists of the entity type or objectification name with the value type name (reduced to only characters valid in grUML attribute names) as suffix.

Attribute domain: Value types in ORM must have a data type which defines their value domain. Value types can either be represented through reference modes (see previous section) or be user-defined. In the latter case, the user can select from various domains which are listed in appendix B table 5 along with their correspondences in grUML. The data type of an ORM value type can't be determined by looking at the ORM schema graph but is stored in the ORM schema file.

Example: Figures 2.3 and 2.4 show an example of an ORM schema with an entity type "Student" in a relation with a value type "StudentName" and its mapping to the vertex class "Student" with the attribute "studentName" in grUML. The data type of the value type "StudentName" was defined as "Text: Variable Length" in the ORM schema file. This maps to the domain "String" for the attribute "studentName" in grUML. The entity type "Student" has the reference mode "ID" which is its preferred identifier, so the value type name "StudentName" is merely adjusted to meet the grUML naming conventions without prefixing it with the entity type's name. The preferred identifier of "Student" is stored in its attribute "preferredIdentifiers" while the attribute "studentName" which is no preferred identifier is simply added to the class.

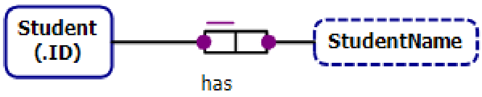


Figure 2.3: An ORM schema of an entity type and a value type taking part in a binary fact type.

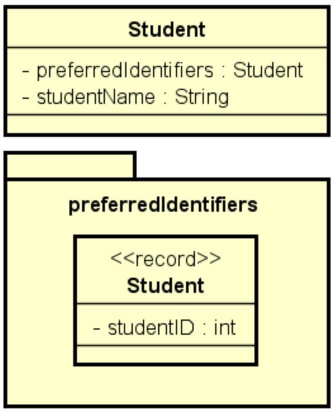


Figure 2.4: The mapping of the ORM schema on the left to a grUML schema. The value type "StudentName" maps to the attribute "studentName" of the vertex class "Student". Preferred identifiers can be clearly distinguished from attributes generated from relations with value types.

Exceptions: Not all value types can be mapped to attributes since an attribute requires the existence of a vertex class. Value types which either have no relations at all or have relations that are exclusively with other value types can't be mapped to attributes for this reason. ORM schemas containing these constellations are rejected.

2.5.1 Mapping of value type domains

Not all ORM value domains could be mapped to a precise equivalent in grUML and ORM's raw value types cannot be represented in grUML at all. This is due to the fact that ORM supports a wider range of domains compared to grUML.

Some of the ORM value type domains are highly specific. For example, the domain "Numeric: Float (Custom Precision)" allows the definition of the precision of the float value. In a grUML schema, this domain is represented by the "Double" domain with the additional requirement of cropping attribute values inserted into a TGraph to match the defined precision. Other numeric ORM value domains don't have an exact equivalent in grUML: e.g. "Numeric: Unsigned Integer" (32 bit unsigned integer) is represented as a "Long" (64 bit signed integer) in grUML, but the "Long" domain needs to be restricted to values ≥ 0.

For this thesis, the mapping of value type domains was implemented as defined in appendix B table 5 without taking into account the additional demands regarding precision or value ranges.

2.6 Objectification mapping

ORM element: Objectification

Corresponding grUML element: Vertex class

Description: ORM allows the objectification of predicates and the participation of the resulting objectified association in further fact types. This behavior is mapped to grUML by generating a new vertex class, that represents the objectification. This mapping both reflects the conceptual character of an ORM objectification and includes the possibility for it to participate in further relations. Objectifications, just as entity types, have preferred identifiers, which are stored in the attribute "preferredIdentifiers" of the vertex class generated from the objectification. The preferred identifiers of objectifications are generated automatically from the objectified fact type and the uniqueness constraints applying to it. These preferred identifiers cannot be defined explicitly.

Name: The name of the vertex class representing the objectification is based on the name of the objectification within the ORM schema. If necessary, this name is modified in the same way as entity type names (see section 2.3) to match grUML convention. .

Example: The mapping of an ORM objectification to a grUML vertex class is shown in figures 2.5 and 2.6.

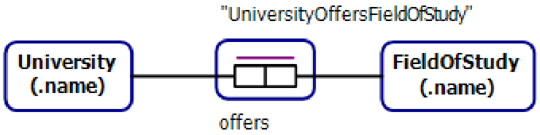


Figure 2.5: An example of an objectification. The fact type “University offers FieldOfStudy” is objectified and given the name “UniversityOffersFieldOfStudy”.

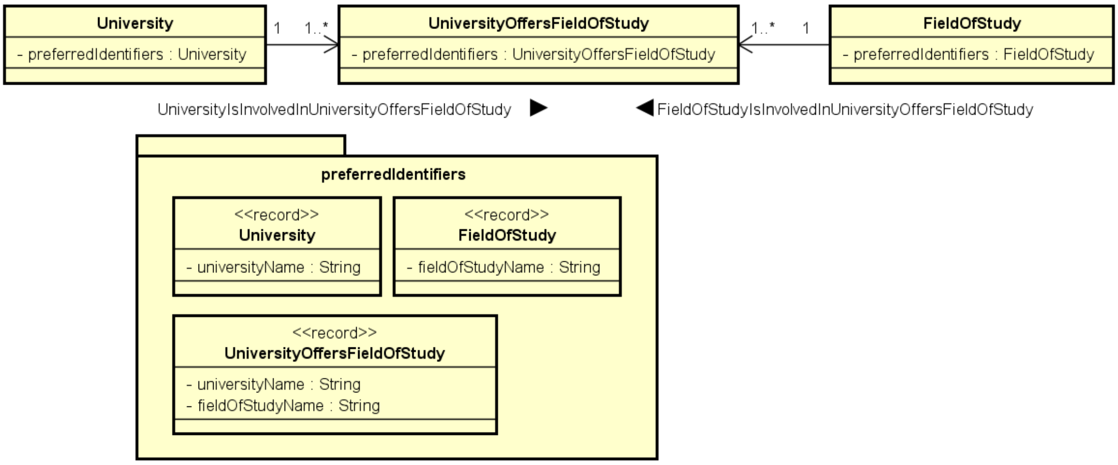

University
- preferredIdentifiers : University
1
1..*
UniversityOffersFieldOfStudy
- preferredIdentifiers : UniversityOffersFieldOfStudy
1..*
1
FieldOfStudy
- preferredIdentifiers : FieldOfStudy
UniversityIsInvolvedInUniversityOffersFieldOfStudy ▶
◀ FieldOfStudyIsInvolvedInUniversityOffersFieldOfStudy
preferredIdentifiers
<<record>>
University
- universityName : String
<<record>>
FieldOfStudy
- fieldOfStudyName : String
<<record>>
UniversityOffersFieldOfStudy
- universityName : String
- fieldOfStudyName : String

Figure 2.6: The mapping of the ORM schema containing an objectification. The objectification “UniversityOffersFieldOfStudy” has become a vertex class with its own preferred identifiers.

Exceptions: -

2.7 Fact Type mapping

ORM element: Fact type

Corresponding grUML element: Edge class connecting vertex classes (mapping of relations between entity types) *or* vertex class containing an attribute (mapping of relations between entity types and value types)

Description: Fact types in ORM describe relationships between entity types, objectified facts and/or value types. The equivalent for relations between entity types or objectified facts in ORM are associations between vertex classes in grUML.

Relations between entity types and value types have a different semantic since they form the link between an entity and the value which is required for the identification of this entity or which is used to further describe it. In grUML this is mirrored by mapping the value type to an attribute of the vertex class representing the related entity type or objectification.

Associations in grUML schemas must be binary. Thus, ORM unary as well as ternary and higher arity relations require special handling in order to transform them into binary associations in grUML. Concordantly, the mapping of ORM fact types will first be introduced based on their arity in the following sections.

Name: If an edge class is created as a result of mapping a fact type, the edge class name is generated from the fact name stored in the ORM schema file. The fact name is the predicate reading of a fact type with the object type names inserted into the "holes". For one fact type, multiple predicates can be provided in a single ORM schema. The mapping presented in this work makes use of the first name listed in the ORM schema file. From this name all invalid characters are removed, the first letter is transformed to upper case and if necessary the name is qualified with an integer value since edge class names must be unique throughout the grUML schema.

If the fact type is defined between entity types and value types, each value type becomes an attribute of each of the involved entity types. The attribute name is created as described in section 2.5.

Exceptions: In this mapping ORM schemas containing fact types which consist only of value types are rejected. The reasons for this were presented in 2.5 but can be summarized as the inability to map such value types to attributes.

2.7.1 Mapping unary Relations

ORM element: Unary relation involving an entity type

Corresponding grUML element: vertex class with a boolean attribute which represents the unary relation. If an entity plays a unary role, the attribute has the value "true" and otherwise "false".

Description: Unary relations bring up the question of what happens if an object type hosting a unary relation doesn't play this role. In ORM and modeling in general, there are two different concepts to answer this question.

With the *"closed world" assumption* [HM08, ch. 3.3, p. 65], the absence of an information, i.e. an object type does not play its unary role, implies the negative of this information. So for a fact type "Person drinks alcohol." and an instance of Person "Penny" which does not play the role "drinks alcohol" it is implied that Penny does not drink alcohol.

With the *"open world" assumption* [HM08, ch. 3.3, p. 65], negative information can be modeled explicitly. If this is not the case and "Penny" does not play her role "drinks alcohol", then it is not possible to conclude that Penny does not drink alcohol.

In this work, the "closed world" assumption is used. The information whether the entity type participates in a unary fact type can be represented by adding a boolean attribute to the vertex class representing the entity type.

Example: Figures 2.7 and 2.8 show an example of this mapping. The predicate of the unary relation "paid tuition fees" is transformed into the boolean attribute "paidTuitionFees" of the vertex class "Student". Again, "Student" has an attribute "preferredIdentifiers" but the boolean attribute is not included in the record since it does not identify instances of "Student".

Exceptions: -

2.7.2 Mapping binary relations containing only entity types or objectifications

ORM element: Binary relation between entity types or objectifications

Corresponding grUML element: Edge class connecting vertex classes that represent the entity types or objectifications

Description: A binary relation between entity types or objectifications maps to two vertex classes connected by an edge class.

In analogy to UML associations, edge classes in grUML are directed, have a name, and each end has an optional role name and multiplicity value. The direction of the edge class depends on the predicate reading which is selected.

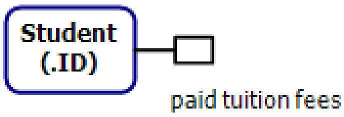


Figure 2.7: An ORM schema displaying a unary relation hosted by the entity type "Student".

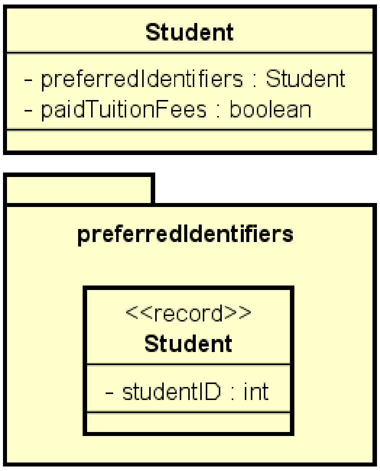


Figure 2.8: The grUML schema as a result of mapping the ORM schema on the left. The unary relation is transformed into the boolean attribute "paidTuitionFees".

Name: The edge class name is generated from the fact name stored in the ORM schema file. The fact name is the predicate reading of a fact type with the object type names inserted into its "holes". For one fact type, multiple predicates can be provided in a single ORM schema. The mapping presented in this work uses the first name listed in the ORM schema file. From this name all invalid characters are removed, the first letter is transformed to upper case and if necessary the name is qualified with an integer value since edge class names must be unique throughout the grUML schema.

Role name: The ORM schema can optionally contain a role name for a fact type role. This role name can be mapped directly to a grUML edge class role name because no naming conventions apply to it except that a role name needs to be unique throughout the entire grUML schema. So it may be necessary to qualify a role name with an integer value to ensure this.

Multiplicity: The multiplicity of the edge class is determined through *internal mandatory* and *internal uniqueness constraints* applying to the relation formulated in ORM. Table 2.1 shows how binary ORM relations between entity types are mapped.

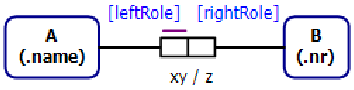
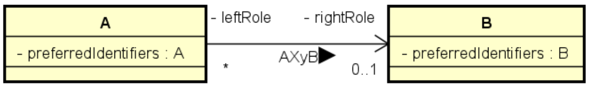
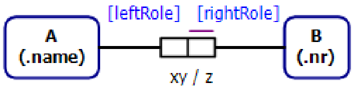
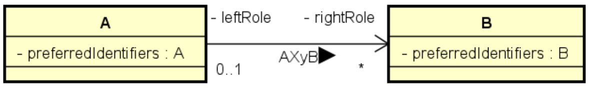
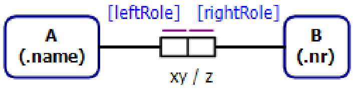
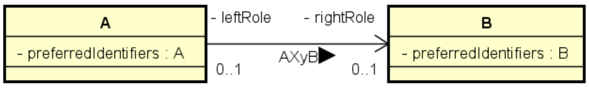
Consider the first ORM schema: here A's role has a uniqueness constraint. This means that over all relations between instances of A and B, the instances of A need to be unique with respect to their unique identifier. This means, that any instance of A can participate in at most one relation with an instance of B. This is represented in the corresponding grUML schema by placing the multiplicity of 0..1 on the end opposite A. If A's role in this fact type were mandatory, the multiplicity would be 1. In this fact type, B has no uniqueness constraint and thus no restrictions as to how many times a single instance of B can participate in a relation with an instance of A. This translates into a multiplicity of *. If B's role in this fact type were mandatory, its multiplicity would be 1..*.

Determining the multiplicity values for fact types with other combinations of single-role uniqueness constraints and mandatory constraints requires the same considerations. The results can be seen in table 2.1.

A spanning uniqueness constraint (as seen e.g. in the fourth entry of the table) means that all combinations of instances of A and B must be unique with respect to their unique identifiers. This provides information on the multiplicity: A and B both must have multiplicity * in this relation, since the only requirement is that their combinations be unique. However, multiplicity alone would allow duplicates of A-B instance pairs, so the grUML schema shown in the table contains a note which demands that the combinations are unique. During mapping this requirement is turned into a grUML constraint. Section 2.8 will go into more details about the mapping of ORM constraints.

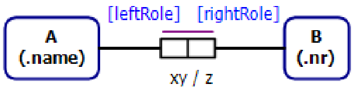
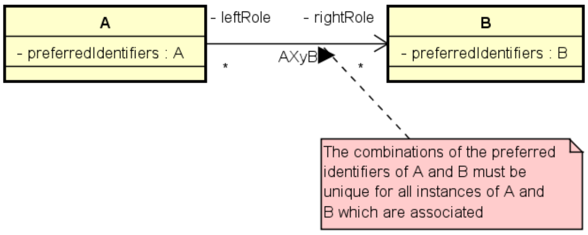
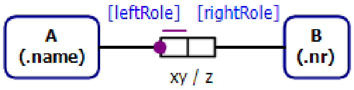
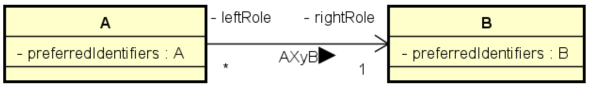
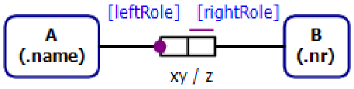
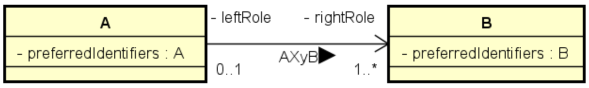
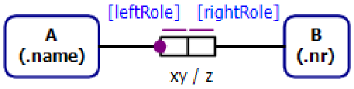
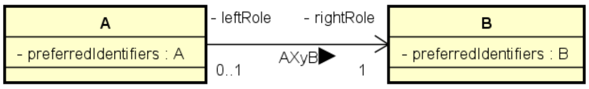
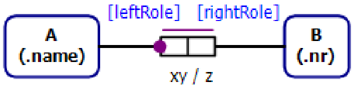
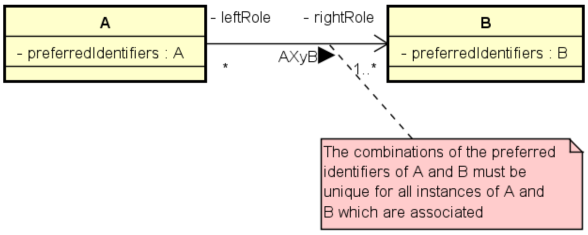
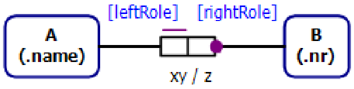
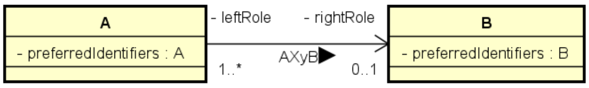
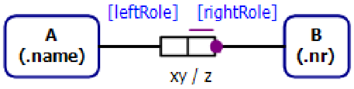
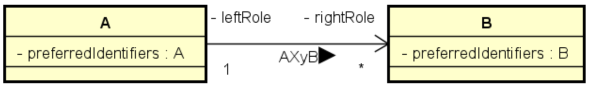
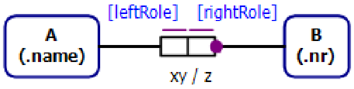
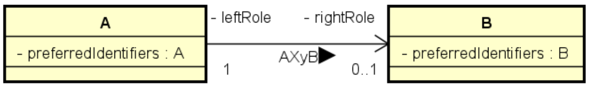
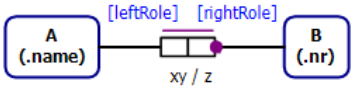
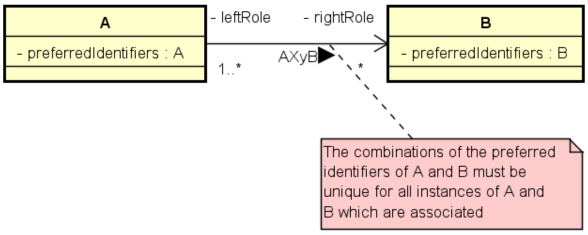
Example: Table 2.1 shows the mapping of binary relations in ORM in which only entity types participate.

Table 2.1: The mapping of ORM binary relations to grUML binary associations.

ORM schema	Mapping to grUML
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB * 0..1 B - preferredIdentifiers : B
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 0..1 * B - preferredIdentifiers : B
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 0..1 0..1 B - preferredIdentifiers : B

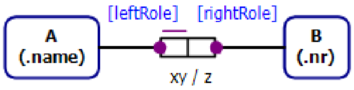
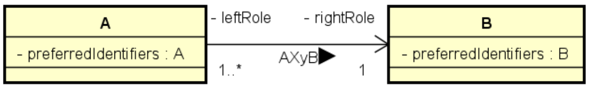
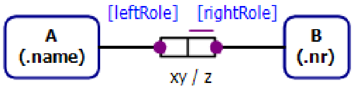
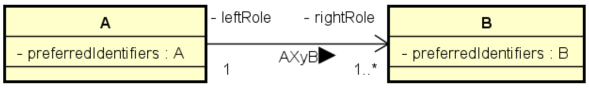
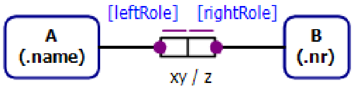
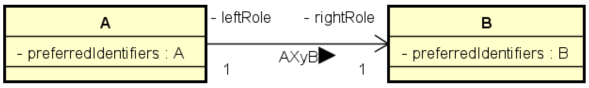
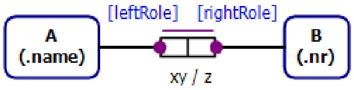
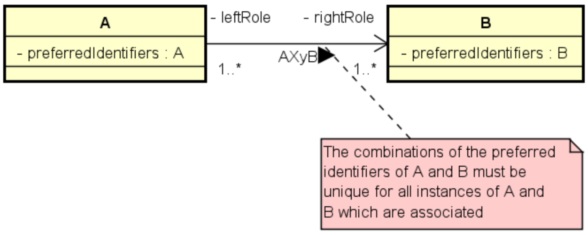
Continued on next page

Table 2.1 – continued from previous page

ORM schema	Mapping to grUML
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B * AXyB * The combinations of the preferred identifiers of A and B must be unique for all instances of A and B which are associated
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B * AXyB 1
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 0..1 AXyB 1..*
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 0..1 AXyB 1
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B * AXyB 1..* The combinations of the preferred identifiers of A and B must be unique for all instances of A and B which are associated
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 1..* AXyB 0..1
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 1 AXyB *
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 1 AXyB 0..1
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole B - preferredIdentifiers : B 1..* AXyB * The combinations of the preferred identifiers of A and B must be unique for all instances of A and B which are associated

Continued on next page

Table 2.1 – continued from previous page

ORM schema	Mapping to grUML
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 1..* 1 B - preferredIdentifiers : B
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 1 1..* B - preferredIdentifiers : B
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 1 1 B - preferredIdentifiers : B
 [leftRole] [rightRole] A (.name) B (.nr) xy / z	 A - preferredIdentifiers : A - leftRole - rightRole AXyB 1..* 1..* B - preferredIdentifiers : B The combinations of the preferred identifiers of A and B must be unique for all instances of A and B which are associated

Exception: -

2.7.3 Mapping binary relations containing value types

ORM element: Binary relation containing one value type and one entity type or objectification

Corresponding grUML element: The entity type or objectification maps to a vertex class which contains an attribute representing the related value type. The attribute is either single-valued or multi-valued (in form of a set).

Description: As with binary relations between entity types/objectifications, the mapping of binary relations between entity types/objectifications and value types depends on the internal uniqueness constraints and internal mandatory constraints defined within them. In this situation however, the mapping requires more considerations, since the constraints can't be represented as multiplicity values.

Firstly, the mapping of value types to attributes always depends on the existence of the vertex class they belong to. This is a problem if the value type doesn't play a role with a mandatory constraint because this means that a value type instance may exist without the related entity type or objectification instance. This is of course not possible using a grUML schema that maps the value type to an attribute since a vertex class needs to be generated before an attribute can be set. But it is also debatable how often such a situation will come up during modeling and of what importance it is. From the author's point of view the generation of values without directly linking them to entities or objectifications is a very rare use case. For this reason, table 2.2 only shows the mapping of binary relations between entity types and value types, where the value types play mandatory roles. If the mapping procedure encounters an ORM schema with a binary relation containing a value type that plays a role which is not mandatory, it will map it as if it were mandatory and provide the user with a warning of this behavior.

Secondly, the uniqueness and mandatory constraints influence the attribute's domain. Consider the first example in table 2.2: The instances of A must be unique with respect to A's preferred identifier and each instance of A can be related to at most one instance of Value. This means, that the attribute "value" is single-valued (and in this case modeled to be from the domain Integer) and is optional. This requirement doesn't need to be added to the grUML schema as a grUML constraint.

Now consider the second example in the table: An instance of A can be related to any number of instances of Value but Value's instances must be unique over all instances of A. Since one instance of A can be related to multiple instances of Value, A's attribute "value" is now multivalued and from the domain Set<Integer>. The choice of a set instead of a list to store multiple values was made because the instances of Value must be unique over all instances of A which implies the uniqueness within each instance of A. This guaranteed when using a set. The requirement that A.value be unique over all instances of A needs to be added to the grUML schema as a grUML constraint (details can be found in section 2.8).

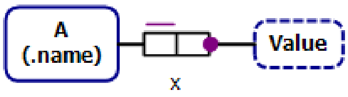
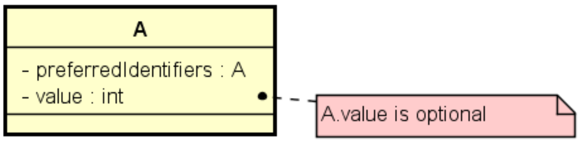
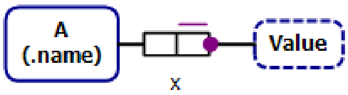
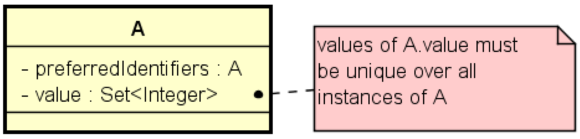
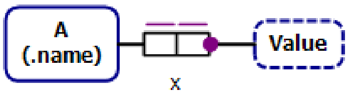
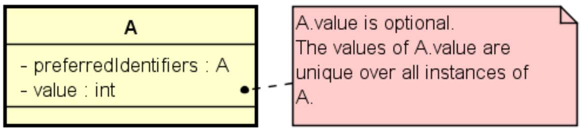
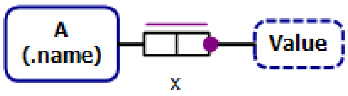
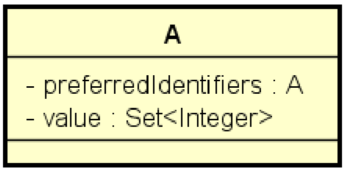
The third example in table 2.2 means that each instance of A can have at most one Value instance attached to it but each instance of Value must be related to an instance of A. In addition, an instance of A can exist without an instance of Value, so A.value is optional. Since each instance of A can have at most one instance of Value which is related to it, the attribute A.value is optional and single-valued. The requirement that A.value be unique over all instances of A needs to be added to the grUML schema as a grUML constraint (details can be found in section 2.8).

Finally, the fourth example in the table contains a spanning uniqueness constraint. This indicates that the combination of instances of A and Value must be unique. So one instance of A could have any number of instances of Value related to it and thus, the attribute "value" is multivalued and defined to be from the domain Set<Integer>. The requirement, that each instance of A has unique combinations with instances of Value is guaranteed by storing instances of Value in a set. This mapping doesn't require a grUML constraint.

The other cases listed in table 2.2 are analogous to the four previous ones, but the attribute "value" is no longer optional due to A's role now being mandatory. All mappings for these cases require additional grUML constraints to ensure the requirements stated in the notes.

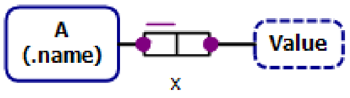
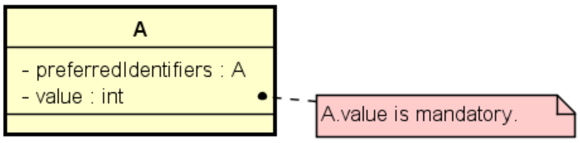
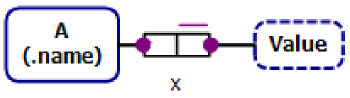
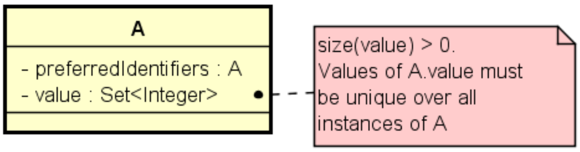
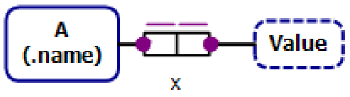
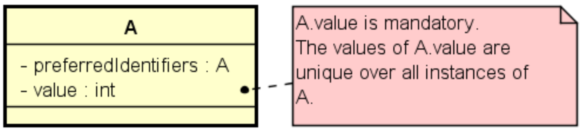
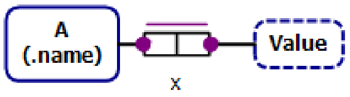
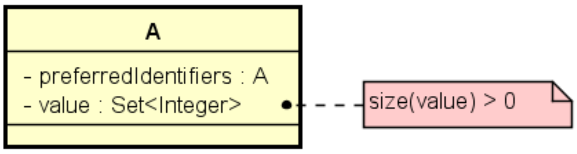
Example: Table 2.2 shows the mapping of binary relations in which an entity type and a value type participates.

Table 2.2: The mapping of ORM binary relations between an entity type and a value type.

ORM schema	Mapping to grUML
	 A - preferredIdentifiers : A - value : int A.value is optional
	 A - preferredIdentifiers : A - value : Set<Integer> values of A.value must be unique over all instances of A
	 A - preferredIdentifiers : A - value : int A.value is optional. The values of A.value are unique over all instances of A.
	

Continued on next page

Table 2.2 – continued from previous page

ORM schema	Mapping to grUML
	 A - preferredIdentifiers : A - value : int A.value is mandatory.
	 A - preferredIdentifiers : A - value : Set<Integer> size(value) > 0. Values of A.value must be unique over all instances of A
	 A - preferredIdentifiers : A - value : int A.value is mandatory. The values of A.value are unique over all instances of A.
	 A - preferredIdentifiers : A - value : Set<Integer> size(value) > 0

2.7.4 Mapping ternary and higher arity relations

ORM element: Ternary or higher arity relations containing only entity types or objectified types

Corresponding grUML element: An *additional vertex class* is introduced to represent the ternary (or higher arity) fact type. The other vertex classes (generated from mapping of the entity types or objectified types participating in the fact type) have binary associations with this additional vertex class.

Description: In contrast to UML, grUML only supports binary associations. In order to map ternary and higher arity relations, an *additional* vertex class is introduced and each vertex class participating in the ternary (or higher arity) relation has an association with this additional vertex class. The additional vertex class' name is generated from the ternary fact's name. As all other vertex classes that result from this mapping, the additional vertex class has the "preferredIdentifiers" attribute. The names of the edge classes leading from the vertex classes participating in the relation to the additional vertex class are generated from the vertex class' name, followed by "InvolvedIn" and the additional vertex class' name. Each instance of the additional vertex class stands for a single ternary relation between three object types in ORM.

Since each object type must participate in the relation for it to exist and at most one object of each type can participate in each relation, the multiplicity of the association at the end of the participating vertex classes is 1. As each vertex class, i.e. object type in ORM, can participate in multiple ternary relations of the same type, the multiplicity of the association at the end of the additional vertex class is * and may be 1..* if the role is declared mandatory in the schema.

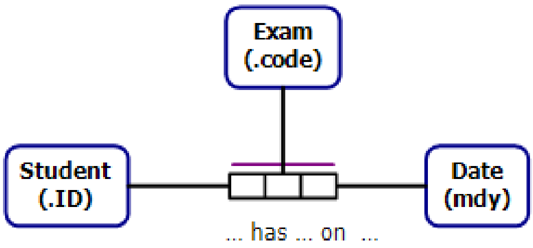


Figure 2.9: An ORM schema displaying a ternary relation hosted by three entity types.

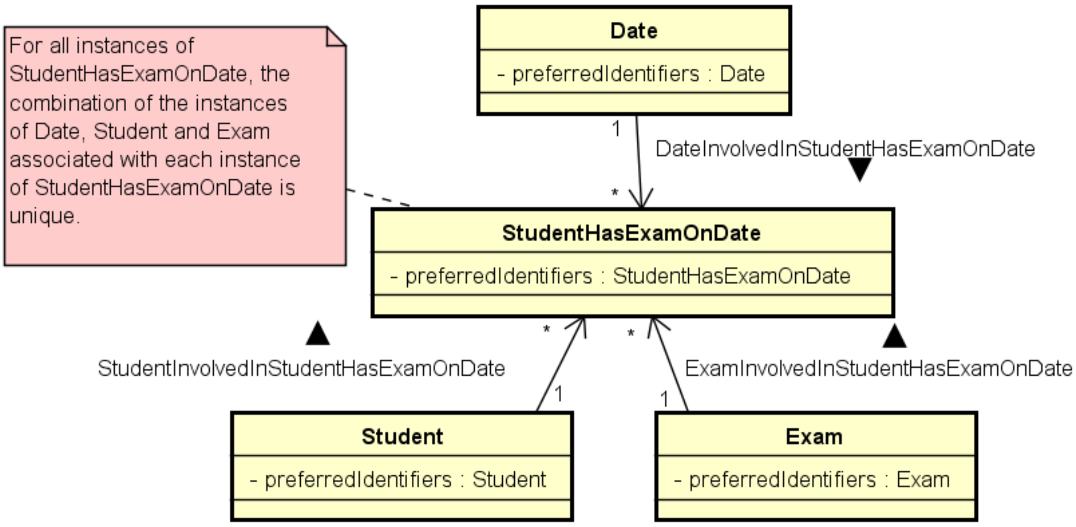

For all instances of StudentHasExamOnDate, the combination of the instances of Date, Student and Exam associated with each instance of StudentHasExamOnDate is unique.
Date
- preferredIdentifiers : Date
1
DateInvolvedInStudentHasExamOnDate
*
StudentHasExamOnDate
- preferredIdentifiers : StudentHasExamOnDate
*
*
StudentInvolvedInStudentHasExamOnDate
ExamInvolvedInStudentHasExamOnDate
1
1
Student
- preferredIdentifiers : Student
Exam
- preferredIdentifiers : Exam

Figure 2.10: The grUML schema as a result of mapping the ORM schema above (figure 2.9). The ternary fact type "Student has Exam on Date" is transformed into the additional vertex class "StudentHasExamOnDate". The package containing the records for the preferred identifiers of the involved vertex classes was omitted in favor of compactness.

Example: Figures 2.9 and 2.10 show the mapping of a ternary relation containing only entity types. In order to ensure that the uniqueness constraint in figure 2.9

is met, the additional vertex class “StudentHasExamOnDate” needs a grUML constraint which is informally stated in the note attached to it in figure 2.10. During the mapping process the uniqueness constraint is transformed into a grUML constraint and added to the grUML schema (details can be found in section 2.8).

Exceptions: -

Variation: Ternary and higher arity relations which contain value types are mapped by creating the additional vertex class which represents the relation but each value type is mapped to an attribute of the vertex classes representing each of the entity types or objectifications participating in the fact type.

Variation example: Figure 2.11 is a variation of the previous ORM schema. Now “Date” is modeled as a value type instead of an entity type because “Date” doesn’t take part in any other relations and only serves as a storage container for a value. Figure 2.12 shows the mapping of this modified ternary relation. The uniqueness constraint verbalized in the note in this schema needs to be translated into a grUML constraint during mapping (details can be found in section 2.8). Furthermore, it needs to be ensured that instances of “Student” and “Exam” which are related via a ternary relation receive the same values for “date”.

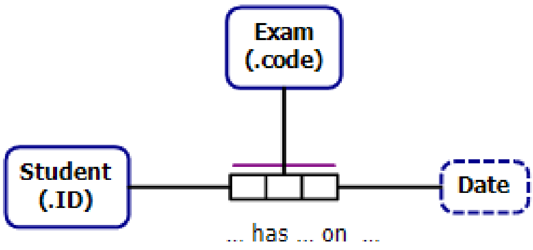


Figure 2.11: An ORM schema displaying a ternary relation hosted by both entity types and value types.

2.8 Constraint mapping

ORM element: ORM constraint

Corresponding grUML element: GrUML constraint

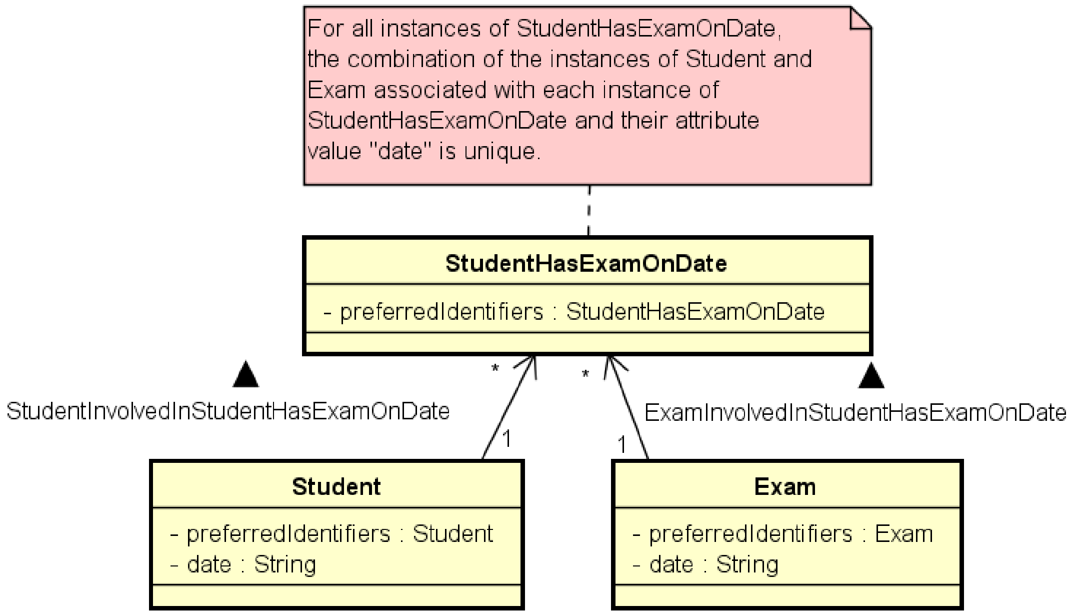

For all instances of StudentHasExamOnDate, the combination of the instances of Student and Exam associated with each instance of StudentHasExamOnDate and their attribute value "date" is unique.
StudentHasExamOnDate
- preferredIdentifiers : StudentHasExamOnDate
StudentInvolvedInStudentHasExamOnDate
ExamInvolvedInStudentHasExamOnDate
*
*
1
1
Student
- preferredIdentifiers : Student
- date : String
Exam
- preferredIdentifiers : Exam
- date : String

Figure 2.12: The grUML schema as a result of mapping the ORM schema shown in figure 2.11. The value type "Date" is now an attribute of the two vertex classes created from the entity types which "Date" was related to. The uniqueness constraint still involves "Date" but required modification compared to figure 2.10 since "Date" is no longer a vertex class.

Description: The ORM constraints that have a graphical notation were introduced in chapter 1.5.2. The constraints will be transferred to grUML in form of grUML constraints which are formulated in GReQL syntax.

ORM offers a large assortment of constraints and one constraint may find application in various different schemas. For example, an exclusion constraint may apply to roles played by only entity types in one schema but constrain roles played by value types and entity types in another schema. Since these schemas will map in different ways to grUML (the value types will become attributes and not vertex classes), each constraint needs to be adapted to the individual situation.

To map all the constraints that ORM has to offer to grUML, it is necessary to consider all possible scenarios in which they may be used. Then each constraint can be formulated as a grUML constraint. Though grUML constraints consist of three parts, this section will be limited to the second part: creating the GReQL expression that represents the constraint which can be used to check whether a given TGraph corresponds to this constraint or not. The GReQL expression is based on the semantics of the ORM constraint and the schema it applies to.

The generation of the first and third part of a grUML constraint - the verbalization and the GReQL expression to retrieve violating elements - are addressed in chapter 3. The finished grUML constraint is added to the grUML schema and can be validated at the implementation level.

Due to the large number of constraints and the even larger number of schema constellations in which they could be used, it isn't possible to introduce all the GReQL constraints designed for each constraint and each scenario during this thesis. Instead, this section will give an example mapping for ORM's exclusion constraint applied to a specific schema.

In this chapter about mapping, the ORM constraint modalities "alethic" and "deontic" are not addressed. They take effect at the implementation level and don't play a role for the manner in which the constraints are mapped, so they can be omitted at this point but will be mentioned in chapter 3.

Example: see section 2.8.1

Exceptions: -

2.8.1 Example mapping of an exclusion constraint

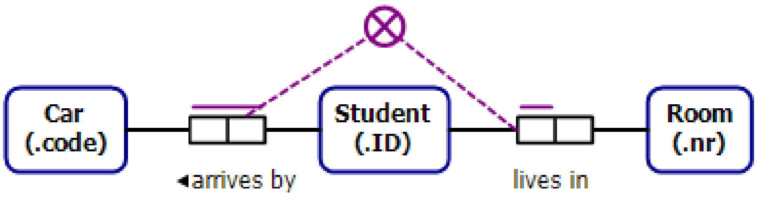


Figure 2.13: ORM schema containing an exclusion constraint.

Figure 2.13 shows an ORM schema with an exclusion constraint. Figure 2.14 shows the mapping of this ORM schema to grUML. Table 2.3 gives an example of a grUML constraint that expresses the ORM exclusion constraint from figure 2.14. It contains the GReQL expression which is applied to instances of the grUML schema from figure 2.14 to ensure they don't violate the constraint. It also contains the GReQL query that can optionally be added to a grUML constraint to retrieve those TGraph elements which are in violation of the constraint defined in the GReQL

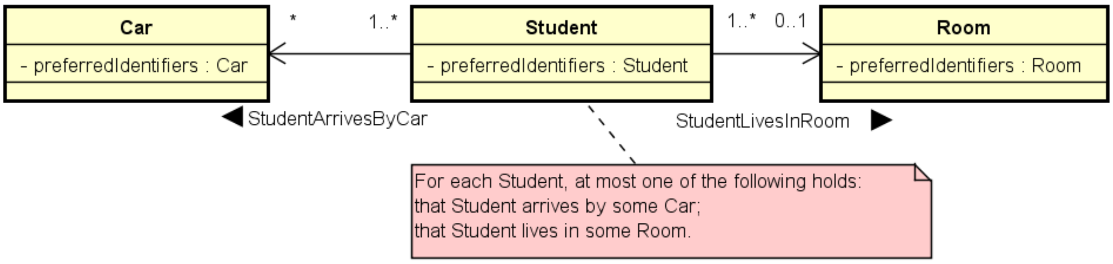

Car
- preferredIdentifiers : Car
*
1..*
Student
- preferredIdentifiers : Student
1..*
0..1
Room
- preferredIdentifiers : Room
◀ StudentArrivesByCar
StudentLivesInRoom ▶
For each Student, at most one of the following holds:
that Student arrives by some Car;
that Student lives in some Room.

Figure 2.14: The grUML schema as a result of mapping the ORM schema above (figure 2.13). The note verbalizes the constraint that needs to be added to the grUML schema.

Table 2.3: grUML constraint corresponding to the exclusion constraint in figure 2.13. All three components of the grUML constraint are displayed.

Verbalization	For each Student, at most one of the following holds: that Student arrives by some Car; that Student lives in some Room.

GReQL expression	`forall s:V{Student} @`
	`degree{StudentArrivesByCar}(s) < 1`
	`or degree{StudentLivesInRoom}(s) < 1`

GReQL query	`from s:V{Student} with not(`
	`degree{StudentArrivesByCar}(s) < 1`
	`or degree{StudentLivesInRoom}(s) < 1) report s end`

expression. And lastly, it contains the verbalization of the constraint in natural language.

2.9 Independent Object Type mapping

ORM element: Independent object type

Corresponding grUML element: Vertex class (possibly requires adjustment of association multiplicities to a lower bound of 0). Independent vertex classes receive a note “independent” within the grUML schema

Description: In ORM, entity types, objectifications and value types can be declared independent. Independent object types can exist without participating in fact types. In UML and in grUML, classes can exist without participating in associations - either because they aren’t attached to an association or because the multiplicity of the association allows this. Concordantly, the mapping of entity types and objectified types to vertex classes also works if these elements are declared independent in ORM. It may only necessary to adjust the multiplicity (lower bound must be 0) of the associations in which these vertex classes take part. To clarify for the user that the generated vertex classes actually stem from an entity type or objectification that was declared independent, the note “independent” is added to these elements in the grUML schema.

Exceptions: Using the proposed mapping, the independence of value types cannot be represented in grUML and ORM schemas containing independent value types are rejected. In this mapping procedure value types map to attributes so their existence requires the existence of the vertex class they belong to. But, although it is not possible to create an attribute that can exist without a class in grUML, it is questionable what role independent value types might play in modeling in general.

2.10 Subtype mapping

As introduced in section 1.5.5, ORM allows subtyping with multiple inheritance. Additionally it allows an object type instance to be a member of multiple subtypes at once. It is also possible to define constraints on subtype relations which may not affect all subtypes of one supertype.

The following sections will first portrait how subtype relations with and without constraints are mapped when they refer to entity types or objectifications. Finally, one section will discuss the mapping of subtype relations between value types.

2.10.1 Subtype relations without constraints

ORM element: Subtype definition (between entity types or objectifications) without constraints

Corresponding grUML element: The subtype hierarchy is copied to grUML. The user is given a warning that a vertex class instance in grUML can’t be a member of multiple subclasses at once which is possible in ORM.

Description: A subtype definition between entity types (or objectifications) in ORM without additional constraints allows multiple realizations at the instance level. A supertype instance can be a member of (multiple) subtypes or of none. The subtypes may be exhaustive for their supertype, meaning that their union makes up the supertype, or not.

The first circumstance (one instance belonging to multiple subtypes) cannot be represented in grUML because it isn’t possible that a vertex class instance is an instance of multiple vertex classes at once. When mapping schemas with non-constrained subtype relations the user is given a warning, that vertex class instances cannot be members of two subclasses at once and thus that this mapping reduces ORM's expressiveness.

The second case (subtypes can be exhaustive of their supertype or not) is represented in grUML by simply mapping the type hierarchy to a vertex class hierarchy. This allows the instantiation of both the superclass and the subclasses but doesn't prevent instantiation of only subclasses (which makes them exhaustive for their superclass).

Note: At this point it should be noted that top-level object types, i.e. object types which aren't subtypes, are always mutually exclusive in ORM. The difficulty that an object type instance can belong to multiple subtypes only arises at the subtype level.

Exceptions: -

2.10.2 Exclusive subtypes

ORM element: Exclusion constraint between subtype relations

Corresponding grUML element: The subtype hierarchy is copied to grUML.

Description: An exclusion constraint between subtypes indicates that instances of the supertype can only be instances of at most one of the subtypes.

This behavior can be transferred to grUML by simply mapping the subtype relations, i.e. creating subclass relations between the vertex classes that result from mapping the entity types to grUML. This suffices, since vertex class instances in grUML are disjoint by definition.

Example: Figure 2.15 shows the ORM schema with exclusive subtypes first introduced in the chapter about ORM, 23. Figure 2.16 next to it shows how the subtype definition is simply transferred to grUML.

Exceptions: -

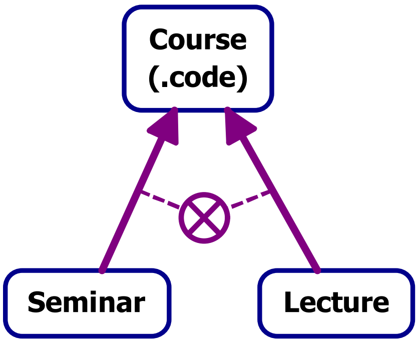


Figure 2.15: ORM schema of a subtype relation with an exclusion constraint (exclusive subtypes).

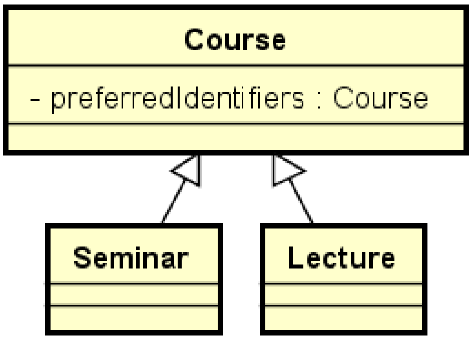


Figure 2.16: The grUML schema as a result of mapping the ORM schema on the left.

2.10.3 Exhaustive subtypes

ORM element: Disjunctive mandatory constraint between subtype relations

Corresponding grUML element: The subtype hierarchy is copied to grUML. The superclass is declared abstract. The user is warned that a vertex class instance in grUML can't be a member of multiple subclasses at once which is possible in ORM.

Description: An inclusive-or constraint between subtypes indicates that each supertype instance must be an instance of at least one of its subtypes. This implies that the subtypes are exhaustive for their supertype.

These semantics can't be fully transferred to grUML. The reason being that ORM allows an instance to belong to multiple subtypes which isn't possible for vertex class instances grUML. In this case, the subtype relation is copied to grUML and the superclass is declared abstract to account for the supertypes being exhaustive for their supertype. The user receives a warning that a subtype relation with this constraint will behave in the same manner as a subtype relation with an exclusive-or constraint (see next section).

Example: Figure 2.17 shows an ORM schema with exhaustive subtypes which was also introduced on page 23. Figure 2.18 next to it the mapping to grUML, where the subtype hierarchy remains unchanged but the superclass is defined as abstract.

Exceptions: -

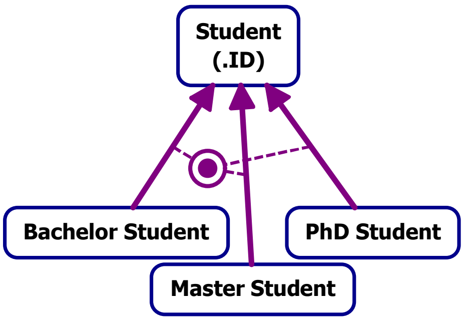


Figure 2.17: ORM schema of a subtype relation with a disjunctive mandatory role constraint (exhaustive subtypes).

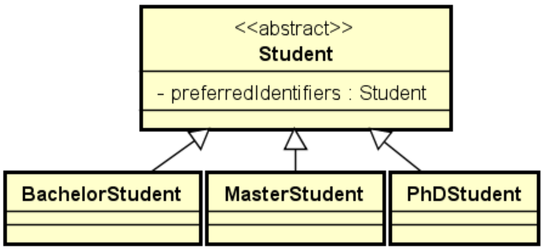


Figure 2.18: The grUML schema as a result of mapping the ORM schema on the left.

2.10.4 Partition of subtypes

ORM element: Exclusive-or constraint between subtype relations

Corresponding grUML element: The subtype hierarchy is copied to grUML. The superclass is declared abstract.

Description: An exclusive-or constraint between subtypes means that these subtypes are mutually exclusive and exhaustive for their supertype, creating a partition of the supertype. This is easily transferred to grUML by maintaining the subtype relations as defined in ORM and making the superclass abstract. By defining the superclass as abstract, it cannot be instantiated so all instances of the superclass' type must be instances of its subclasses. This reflects the aspect of this constraint, that the union of the subtypes in ORM should equal their supertype. The second aspect, the mutual exclusiveness, is automatically met in grUML, since an instance can only be a member of one class.

Example: For an example ORM schema with a subtype partition, please see figure 1.11 on page 23. The mapping of this schema works in analogy to the mapping for exhaustive subtypes (see the previous section) with the exception that it does not require a warning for the user.

Exceptions: -

2.10.5 Further subtype definitions

The cases of subtype definitions containing constraints as introduced in the last sections discussed constraints that applied to all of the subtype relations. In ORM it is possible that a constraint only applies to a subset of the subtype relations for one supertype.

An example for this behavior can be seen in 2.19. Here a Course can either be a Lecture or a Seminar but not both. However, in addition it can also be an OnlineCourse. Given such a schema, there is the possibility that subtype instances belong to multiple types and this situation can not be represented in grUML. In these probably quite rare cases, the subtype hierarchy is mapped to grUML based on the constraints that apply to it and the user receives a warning that vertex class instances cannot be members of two subclasses at once and thus that this mapping reduces ORM's expressiveness.

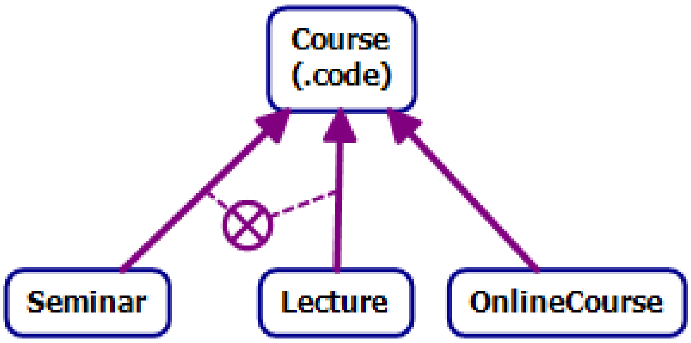


Figure 2.19: A subtype definition where the constraint (here an exclusion constraint) does not affect all subtypes.

2.10.6 Subtype definitions between value types

ORM also allows the definition of hierarchies between value types. This makes sense seeing that value types can be used to further describe the domain of the actual values. Adding hierarchies to value types increases their information content.

Since value types are treated as placeholders for values and thus are transformed to attributes in the course of this mapping, there is no way to represent any subtype relations that may be stated in ORM. For this reasons, ORM schemas that contain hierarchy information for value types are rejected during the mapping process.

2.11 Reference Scheme mapping

In ORM, each entity type must be uniquely identifiable and may have multiple sources for identification. One way of unique identification must be declared as the *preferred* mode.

2.11.1 Preferred reference schemes

In ORM, entity types can have a reference mode which allows their unique identification through a single value. The reference mode is an abbreviation for a 1:1 preferred reference scheme. The mapping of reference modes was discussed in section 2.4.

If an entity type doesn't have a reference mode or an explicitly stated preferred 1:1 reference scheme, it must be uniquely identified by multiple values which is then referred to as a compound reference scheme. If no other reference schemes exist, it must be the preferred reference scheme.

In ORM this is indicated through an interpredicate uniqueness constraint with a double bar that affects roles neighboring the roles played by the entity type in question. The constrained roles are either played directly by value types whose values are used for identification or by entity types (or objectifications). In the latter case the preferred identifiers of these neighboring entity types become part of the set of preferred identifiers for the entity type in question. This concept was introduced along with an example in section 1.5.6.

The value types that are preferably used to uniquely identify the entity type (or even an objectification) are stored in the attribute “preferredIdentifiers” in the vertex class generated from mapping the entity type/objectification. The record “preferredIdentifiers” receives a new component for each of the value types that is part of the preferred compound reference scheme.

Note that only preferred identifiers are stored in the record “preferredIdentifiers”.

2.11.2 Other reference schemes

Other, non-preferred reference schemes (indicated by an interpredicate uniqueness constraint with a single bar) don't have an explicit representation as such in grUML. The interpredicate uniqueness constraint is formulated as a grUML constraint and added to the schema which ensures the uniqueness of the values at instance level.

2.12 Derivation and semiderivation mapping

This far, the mapping of subtypes and fact types was introduced for their *asserted* versions. As explained in section 1.5.7, fact types and subtypes can also be semiderived or derived.

The mapping procedure does not yet include a transformation for derivations and semi-derivations to grUML. There are mainly two reasons for this. Firstly, the generation of derivations in NORMA is not very intuitive and requires a deep knowledge of both ORM and the software to create a derivation which has the intended semantics. Although the outcome can be checked by viewing NORMA's verbalization of the derivation rule, this quickly becomes tedious due to non-transparent generation process and the sometimes unexpected behavior of the software. And secondly, NORMA allows the definition of arbitrarily complex derivations. As an example, derivation rules contain variables which represent object types and NORMA provides 28 functions to manipulate or combine these defined variables.

These two aspects increased the complexity to an extent that the mapping of derivations had to be dismissed as not to overstretch the scope of this work.

2.13 Summary

This section provides an overview of the ORM elements and their mapping to grUML. A summary of the mappings described to this point can be found in table 2.4.

Table 2.4: This table contains a brief summary of the mapping of ORM elements to grUML.

ORM element	**grUML correspondence**
entity type	**vertex class**
objectification	**vertex class**
value type	**attribute** of a vertex class
reference scheme (incl. reference mode)	each vertex class has an **attribute "preferredIdentifiers"** which links to a **record** containing **components** that represent those **values** necessary to uniquely identify the vertex class instances
unary relations	the unary relation is represented as a **boolean** attribute of the vertex class participating the relation ("closed world" assumption)
binary relations	binary relations between entity types become **associations**. Binary relations between an entity type and a value type map to a **vertex class** with an **attribute**.
ternary and higher arity relations	the **ternary** or **higher arity fact type** is represented as a **vertex class** in grUML. This vertex class has **associations** with all **participants** of the fact type.
independent object types	independent value types **can't be mapped**. Independent entity types and objectifications map in the **same way** as **non-independent** elements but might require multiplicity adjustments.
subtyping	the **subtype hierarchy** is **maintained** in grUML. In some cases the **possibilities** for creating subtypes from a grUML schema are **reduced** compared to ORM because instances of a grUML vertex class can't be members of multiple vertex class at once.
derivations	-

3 Implementation

The mapping described in the previous chapter presented the theoretical groundwork necessary to implement a mapping from ORM schema files to grUML schema files. Using the resulting grUML schema files, it should be possible to create TGraph instances that are in concordance with the ORM schema.

At the beginning of this chapter you will find a broad overview of the implementation tasks attempted in this thesis. It will then continue to go into more detail concerning the concrete implementation of the mapping and the TGraph instance generation. It will also touch one some of the difficulties encountered during implementation.

3.1 Overview

The following enumeration will give an overview of the intended scope of the implementation section of this work.

1. **Mapping ORM schemas to grUML schemas**

   (a) Input: valid ORM schema in NORMA's ".orm" file format

   (b) Parse this file to retrieve all information contained in it

   (c) From the parsed data, dismiss unnecessary information and compute additional information as needed

   (d) Create a grUML schema using the parsed data by performing the transformations described in chapter 2

   (e) The resulting grUML schema should convey the same information as the ORM schema or, if this is not possible in certain cases, warn the user that the mapping has reduced the expressiveness or changes the semantics of the ORM schema

   (f) Output: valid grUML schema in ".tg" format

2. **Generating and processing TGraphs generated from mapped grUML schemas**

   (a) Provide a framework which automatically adds ECAs (see section 1.9) to TGraphs and performs validity checks

   (b) The validity of TGraph's should be checked using JGraLab's `GraphValidator` (see section 1.7.4)

3.2 Mapping ORM schemas to grUML schemas

The key steps for implementating the mapping defined in chapter 2 are visualized in figure 3.1.

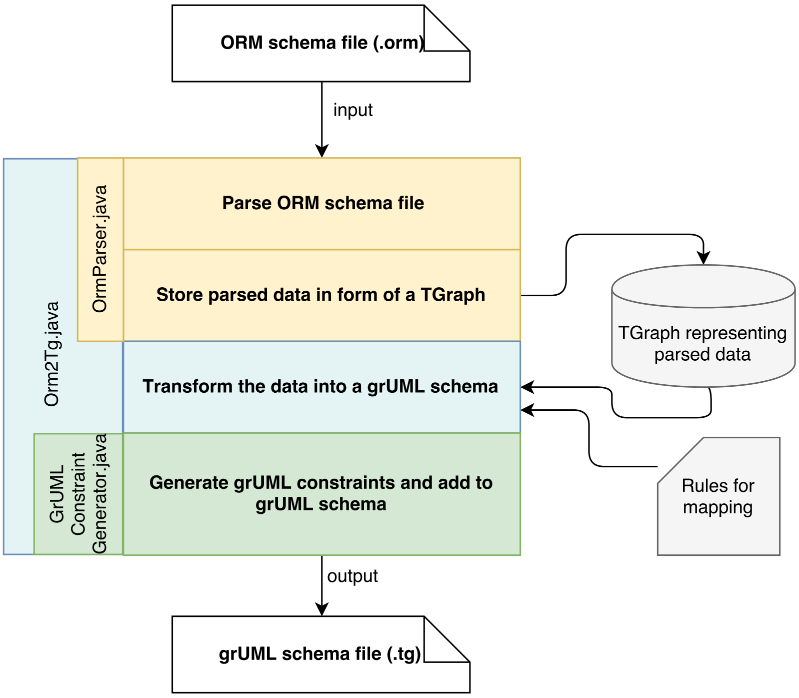

ORM schema file (.orm)
input
Orm2Tg.java
OrmParser.java
Parse ORM schema file
Store parsed data in form of a TGraph
TGraph representing parsed data
Transform the data into a grUML schema
GrUML Constraint Generator.java
Generate grUML constraints and add to grUML schema
Rules for mapping
output
grUML schema file (.tg)

Figure 3.1: A diagram to give an overview of the implementation of the mapping defined in this thesis. The class `ORMParser.java` is responsible for parsing an ORM schema file in ".orm" format. Using the parsed data which is stored in the form of a TGraph, `Orm2Tg.java` performs the actual mapping step. GrUML constraints are generated by classes specializing `GrUMLConstraintGenerator.java` and added to the grUML schema file that is stored in ".tg" format.

3.2.1 ORM schema file

The starting point for the mapping procedure is a file of an ORM schema which is assumed to be semantically correct. For this work, these schema files need to be in ".orm" format which is the file format utilized by NORMA.

The ".orm" schema files are XML-based. Unfortunately, the schema files do not contain a reference to an *XSD* (XML Schema Definition) or *DTD* (Document Type Definition) to help understand the files' structure for the upcoming parsing step.

As a result, the file structure had to be decoded *empirically*. On that account several ORM schemas displaying several possible combinations of object types, fact types, subtype definitions, constraints and objectifications were created. Using these many example schemas and the knowledge of ORM terminology, it was possible to quite reliably predict the structure of ORM schema files generated in NORMA.

Core elements

At the top level, ORM schema files are divided into the following key elements: `Objects`, `Facts`, `Constraints` and `DataTypes`. This section will give a very brief overview of these elements and their roles in the file.

Objects This element can contain any number of the three types of objects, that can appear in an ORM schema diagram: `EntityType`, `ValueType` and `ObjectifiedType`. Within these elements lies most of the information about them: e.g. what roles they play, whether they are part of a type hierarchy, but also type specific data like reference mode names, preferred identifiers or conceptual data type definitions.

For each schema, the `Objects` element lists all objects which can participate in relations. Besides the objects defined by the schema's modeler, the schema may contain additional objects which NORMA generates as auxiliary structures.

Facts ORM fact types are referred to as "facts" in the ORM schema file. Within the `Facts` element, three different kinds of facts can be distinguished: `Fact`, `ImpliedFact` and `SubtypeFact`. They each contain an element `FactRoles` that holds the list of roles which participate in the fact represented by the element.

`Fact` ORM relations which are not objectified and are not subtype relations are represented by this element.

`ImpliedFact` Some facts, e.g. with arity three or above, are not only stored as a `Fact` in the ORM schema file but are also stored in an *objectified* form. In such cases, the relation in question is transformed into an `ObjectifiedType` bearing the name of the original relation. This element is connected to the participants of the transformed relation through an `ImpliedFact` (this process is analogous to the proposed mapping of ternary and higher arity relations in section 2.7.4).

`SubtypeFact` This element stands for a subtype definition between a supertype and its subtype. It consists of `SubtypeMetaRole`s and `SupertypeMetaRole`s. These roles can be subject to disjunctive mandatory constraints, exclusion constraints and exclusive-or constraints which represent the subtype constraints introduced in section 1.5.5.

The `Facts` element lists all relations between the objects defined in the `Objects` element. These relations can be user-defined or implied, e.g. when ternary or higher arity relations are objectified for representation in the schema file.

Constraints This element can contain any of ORM's constraints which apply to fact roles. This leaves the exception of `CardinalityConstraint`s that apply to `ObjectifiedType`s or `EntityType`s and `ValueConstraint`s that apply directly to `ValueType`s.

The remaining constraints contain at least one element called `RoleSequence` which holds the references to those `Role`s which are affected by the constraint. Some constraint elements will contain additional data, e.g. a `FrequencyConstraint` has the attributes `MinFrequency` and `MaxFrequency` which define the lower and upper bound of the frequency range.

For a given ORM schema, the `Constraints` element contains all ORM constraints which apply to roles within this schema. This also includes implied constraints.

DataTypes This element is an enumeration of all the data types which can be assigned to value types in NORMA. A `ValueType` defined in `Objects` will refer to these elements to specify its `ConceptualDataType`.

3.2.2 ORM schema file parser

After gaining a better understanding of the general structure of ORM schema files, the implementation of a parser ensued.

The parser, implemented in `OrmParser.java`, takes a ".orm" file as input and reads and processes the information contained in it. `OrmParser.java` extends JGraLab's class `Xml2Tg.java`. This class reads an XML file and generates a specific TGraph, an `XMLGraph`, from it which represents the file in a fashion similar to a *DOM* (Document Object Model) tree. Besides various general methods giving access to TGraph components, JGraLab also provides the class `XmlGraphUtilities.java` that contains methods specific to retrieving the elements of an `XMLGraph`.

The information contained in the ORM schema file is gathered using these tools and is finally stored in form of a TGraph in an `ORMGraph`. Doing this has the advantage of not needing to implement a data structure to hold the information and allows access to the data through methods provided by JGraLab. Defining the structure of an `ORMGraph` was a part of this work an will be explained in section 3.2.2.

Before or during parsing the ORM schemas are checked for constellations which cannot be mapped using the previously described transformation rules, e.g. schemas containing independent value types.

The parser does not only read the information contained in an ORM schema file but will also compute additional data. For example, it computes multiplicity values based on mandatory, uniqueness and internal frequency constraints to simplify the task of the mapper (see section 3.2.3).

ORM schema file model

This section introduces the model which defines the grUML schema for TGraphs of the type `ORMGraph`, which are used to hold the information contained in an ORM schema file.

Figure 3.2 shows a simplified version of the model designed for this thesis. For the most part, the model represents the key components of the ORM schema file as introduced in section 3.2.1. It is however a simplified version of the model actually used to define the structure of `ORMGraph`s. This model was generated with IBM Rational® Software Architect.

The elements in an ORM schema file have the attribute `id` which is used to uniquely reference them within the file. Concordantly, most vertex classes in figure

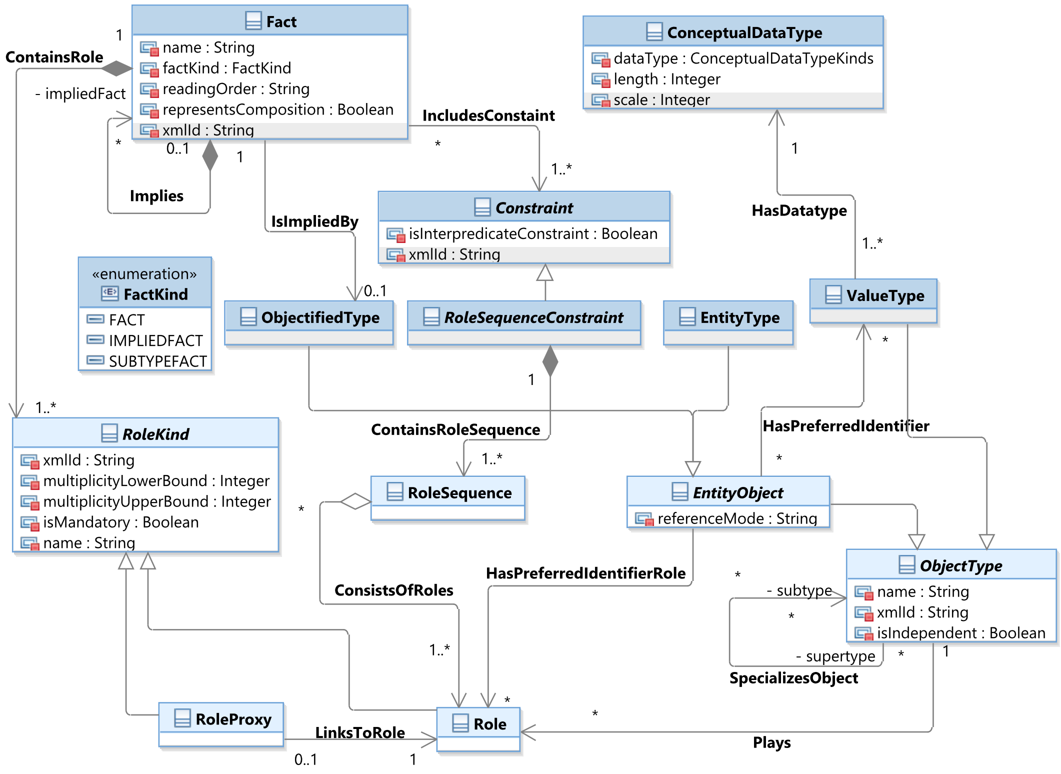


Figure 3.2: A simplified model representing the grUML schema for `ORMGraphs`. It represents the main object types, facts and constraints used in ORM.

3.2 have an attribute `xmlId` which holds this identifier in order to maintain the link between the `ORMGraph` elements and the schema file elements.

Object Type The model defines the three object types that can appear in ORM schemas: `ValueTypes`, `EntityTypes` and `ObjectifiedTypes`.

The last two specialize the class `EntityObject` which represents elements that may have a reference mode or a role that uniquely identifies them.

Each `ValueType` has an association with the vertex class `ConceptualDataType` which represents the ORM value type domains (see appendix B table 5) in its attribute `dataType` but also stores further information regarding the length and scale of values. `ConceptualDataTypeKinds` is an enumeration which contains all possible value type domains but it could not be included in the model due to its large size.

The instances of the classes specializing `ObjectType` can participate in the `SpecializesObject` relation which represents the relation between sub- and supertypes.

Fact ORM relations are represented by the vertex class `Fact`. Each `Fact` has an attribute `factKind` that defines the type of fact it represents. The attribute `representsComposition` can be used to indicate whether a modeler specified a relation within the ORM schema to be a composition or not. Furthermore, a `Fact` has the attribute `readingOrder` which holds a string that contains the predicate reading with placeholders for the involved object types. This is important for the verbalization of the constraints applying to the fact.

Facts can imply each other: As mentioned previously, a ternary fact type is stored in the ORM schema file but additional *implied* facts are generated to represent the relations between the fact's participants and the objectified type created from the fact.

Role and RoleProxy Each `Fact` must contain at least one `RoleKind`. The vertex class `RoleKind` was introduced as a generalization of the two kind of roles that are used in an ORM schema file: the `RoleProxy` and the `Role`.

The vertex class `Role` represents an ORM fact role. The vertex class `RoleProxy` on the other hand also represents a fact role but within a relation between an object type and an objectified fact (represented as an `ObjectifiedType`). If an ORM schema contains an objectified fact, the ORM schema file will include both the objectified fact and the same fact without its objectification. Each `RoleProxy` links to the corresponding `Role` in the fact that is not objectified. Using this link is the only way of finding the player of the `RoleProxy`.

The order of the `RoleKind` elements in the `Fact` later defines the direction of the edge class generated from it. The `readingOrder` corresponds to this order too.

Constraint Finally, `Facts` include at least one `Constraint` each. `Constraint` is an abstract vertex class that represents all constraints that may appear in an ORM schema. Its attribute `isInterpredicateConstraint` indicates whether a constraint is an interpredicate constraint, applying to multiple fact types at once, or not.

This vertex class is specialized by the abstract class `RoleSequenceConstraint`. This class subsumes all the ORM constraints which apply to at least one `RoleSequence`. The `RoleSequence` in turn consists of at least one `Role` which is played by exactly one `ObjectType`. For simplicity reasons, the model is missing all specializations of `Constraint` and `RoleSequenceConstraint` and further aspects that were included in preparation for the mapping of derivations.

Using JGraLab's `Rsa2TG.java` class, the model introduced in this section can be transformed into a TGraph which serves as a grUML schema to define the TGraph type `ORMGraph`.

3.2.3 Mapping the data to grUML

The data extracted from the ORM schema is transformed to grUML step-by-step. The mapping process is implemented in `Orm2Tg.java`. Its method `transformOrm2TG` takes the file path of the ORM schema that should be transformed as input, initializes the parser (`OrmParser`) and processes the data contained in the resulting `ORMGraph`. Here it starts by generating vertex classes from entity types and objectifications, then proceeds to generate attributes from preferred reference schemes and relations with value types and finally generates grUML constraint where necessary. Once the mapping is completed, the grUML schema file can be stored in ".tg" format.

`Orm2TG` has two maps which link the XML IDs of ORM elements in the ORM schema file to the names these elements receive after mapping and vice versa: `xmlIdTogrUMLNameMap` and `grUMLNameToXmlIdMap`. These maps are important when generating grUML constraints, because they create the link between the `ORMGraph` elements affected by a constraint and the names of these elements after mapping them to a grUML schema.

Generating grUML constraints

ORM constraints need to be transformed into grUML constraints. This task is performed by subclasses of the abstract class `GrUMLConstraintGenerator.java`. This class is specialized by constraint-specific classes such as `AcyclicRingConstraintGenerator.java` or `MandatoryConstraintGenerator.java`. `GrUMLConstraintGenerator.java` has three key methods: `createGReQLMessage`, `createGReQLExpression` and `createGReQLOffendingElements`. These generate the three components of a grUML constraint (recall section 1.7). Another important method is `createGReQLConstraintVariables` which creates the variables for use in GReQL expressions.

Verbalizing ORM constraints A grUML constraint requires a verbalization to return to the user if a TGraph is in violation of it. The best way to approach the task of generating a constraint verbalization was to base it off the texts provided by NORMA. Since the verbalization depends on the structure of the schema, it was once again necessary to consider numerous applications of the same constraint to different schemas in order to extract a blueprint for the verbalization. The text also contains the name of the ORM constraint. Assuming that people who use this mapping tool will have a good understanding of ORM, the constraint name should help pinpoint the TGraph elements causing a violation. The generation of these verbalization strings is handled by the class `GrUMLConstraintMessageGenerator.java` which was written in the context of this work.

Representing variables in GReQL expressions As explained in section 1.8, most GReQL expressions require the definition of variables which represent edges or vertices of specified types in a TGraph. When creating GReQL expressions that check a TGraph's correspondence to an ORM constraint, the variables that are defined often refer to the edges or vertices that are directly affected by the constraint.

Within a GReQL expression it may be necessary to not only access the variable itself, but its type, attributes it may have etc. In order to collect all information surrounding these variables, the class `GReQLConstraintVariable.java` (see figure 3.3) was implemented and instances of this class are created during constraint generation.

On the grUML side, a `GReQLConstraintVariable` has a name (`variableName`), knows the name of the grUML type it instantiates (`grUMLTypeName`) and can be set to mute (`isMute`) meaning that it won't show up in a GReQL variable declaration.

On the ORM side, a `GReQLConstraintVariable` has access to the object type (`involvedObjectType`) and the role (`involvedRole`) affected by the constraint. Furthermore, the position of this role within the fact type it participates in is recorded in `positionInFact`.

Finally, the attribute `variableKind` indicates the kind of grUML element the variable represents, i.e. an edge or vertex class element.

The methods are mostly getters but include `declareType` and `printObjectType` which return strings in GReQL syntax providing access to the variable's type or the player of the role from which they were created.

Creating GReQL expressions The `createGReQLConstraintVariables` method needs to be implemented for each class specializing `GrUMLConstraintGenerator.java`. Since each constraint requires a different GReQL expression for

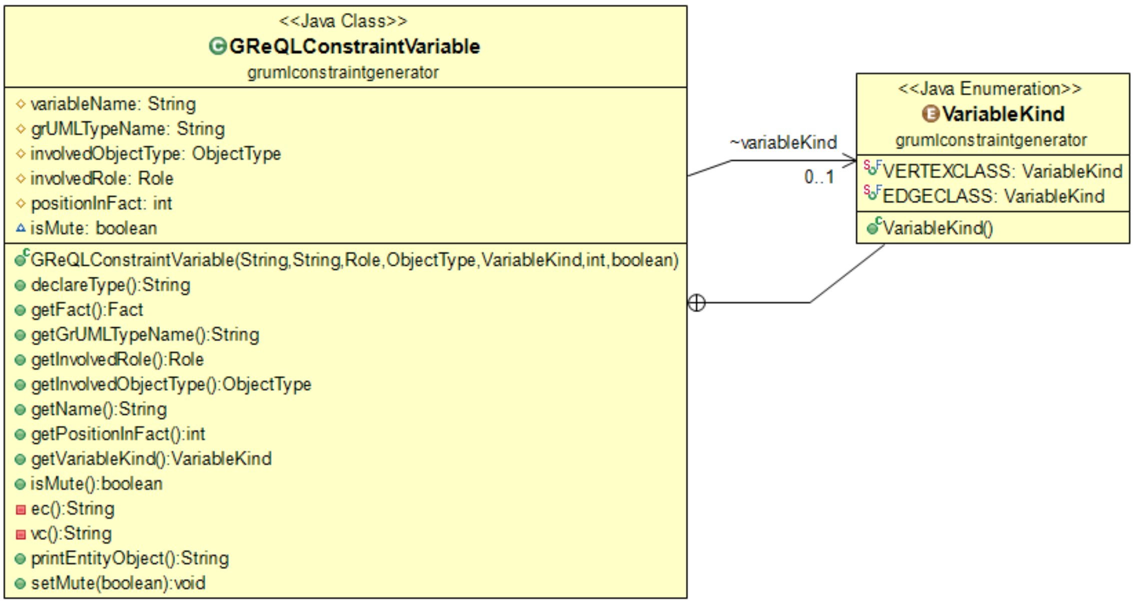


Figure 3.3: The class `GReQLConstraintVariable.java` used to represent variables for use in GReQL expressions. Each variable holds its name, the name of the grUML type it instantiates and its type, so whether it represents an edge class or a vertex class. Furthermore, each variable knows the role from which it was generated, the object type which plays this role and this role's position within the fact type it belongs to. `GReQLConstraintVariables` can also be muted so they do not appear in GReQL expressions. The methods are mostly getters but also provide access to the variables properties using GReQL syntax.

validation and most constraints can apply to multiple scenarios, these two factors greatly influence which variables need to be declared.

The classes generating specific GReQL expressions use a template expression which was established in the theoretical mapping step and adjust this template by inserting the appropriate variables and types.

Listing 3.1 shows an example for such a template. This is a template for the following scenario: an ORM exclusion constraint which applies to two roles played by the same entity type or objectification where the neighboring roles too are played by entity types or objectifications.

Listing 3.1: A template for a GReQL expression expressing an ORM exclusion constraint between two roles played by an entity type where the neighboring roles are played by entity types or objectifications too. <e1> through <e5> are placeholders for variable declarations and GReQL expressions.

```
forall <e1> @ degree{<e2>}(<e3>) < 1 or
degree{<e4>}(<e5>) < 1
```

The ORM schema from figure 2.13, page 62, is an example for such a scenario. A class to generate this exclusion constraint displayed in this example would create variables from the two constrained roles - one mute and one non-mute. The first placeholder `<e1>` is replaced with the GReQL syntax for declaring the non-mute variable: `s:V{Student}`. The remaining placeholders are replaced with the names of the grUML edge class generated from the constrained fact types (`StudentArrivesByCar` and `StudentLivesInRoom`) and the variable's name (`s`) to finally produce the GReQL constraint in listing 3.2.

Listing 3.2: GReQL expression as a result of mapping the ORM exclusion constraint from figure 2.13 (page 60) using the previously defined template.

```
forall s:V{Student} @ degree{StudentArrivesByCar}(s) < 1
or degree{StudentLivesInRoom}(s) < 1
```

Expression for retrieval of violating elements The third, optional part of a grUML constraint, is an expression which retrieves the elements violating the GReQL expression defined in part two. To avoid long explanations, the term "GReQL expression" will refer to the second part of the grUML constraint until the end of this section.

Generating an expression to retrieve violating grUML elements is the final step of generating a grUML constraint, i.e. the GReQL expression is already available at this point. The GReQL expressions used in this work are designed in a way, that this third expression can be created using the template shown in 3.3.

Listing 3.3: A template for an expression to retrieve elements violating the GReQL expression that replaces <e2>.

```
from <e1> with not(<e2>) reportSet <e3> end
```

The first placeholder, `<e1>`, is filled with the variable declaration from the GReQL expression created previously. Then `<e2>` is replaced with the part of the GReQL expression that follows the `@` symbol. Finally, `<e3>` is an enumeration of the variable names declared in `<e1>`.

Listing 3.4 shows an example which uses the GReQL expression introduced in listing 3.2. This expression is added to the grUML constraint representing the exclusion constraint to retrieve elements violating this constraint.

Listing 3.4: The third part of a grUML constraint: an expression to retrieve elements violating the specified constraint (here an exclusion constraint).

```
from s:V{Student} with
not(degree{StudentArrivesByCar}(s) < 1
or degree{StudentLivesInRoom}(s) < 1)
reportSet s end
```

3.3 Generating TGraphs

Besides defining a mapping from ORM to grUML schemas, another very important goal for this work was to provide the possibility of creating instances from the mapped grUML schemas (in form of TGraphs) and to ensure their correspondence with the constraints defined in the ORM schema. This section will talk about the programming performed to take steps towards this goal.

3.3.1 Adding ECA rules

Besides constraints, there are other requirements which ORM schema instances must meet. Consider the ORM schema shown in figure 3.4 which maps to the grUML schema in figure 3.5. If the value of the preferred identifier “dateMdy” of “Date” changes, this also has an influence on the vertex class “Student” since “dateMdy” is one of its preferred identifiers too. The old value of “dateMdy” in the preferred identifiers of “Student” must be replaced with the new value defined for “Date”.

This scenario cannot be solved using grUML constraints, since these do not manipulate TGraphs but only query them. This situation can be dealt with using ECA rules. To address this kind of problem, each vertex class instance (irrespective of the original ORM schema file) will receive an ECA rule that reacts to changes of the “preferredIdentifiers” attribute. An event is generally defined in the form

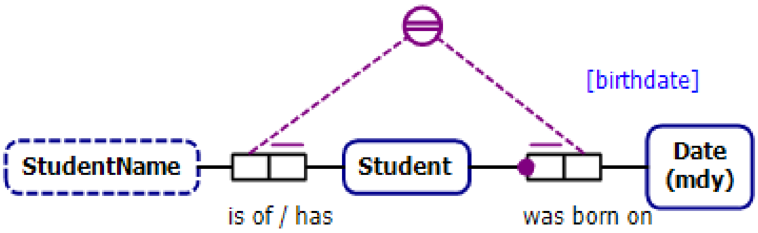


Figure 3.4: An ORM schema defining a preferred compound reference scheme for the entity type “Student”.

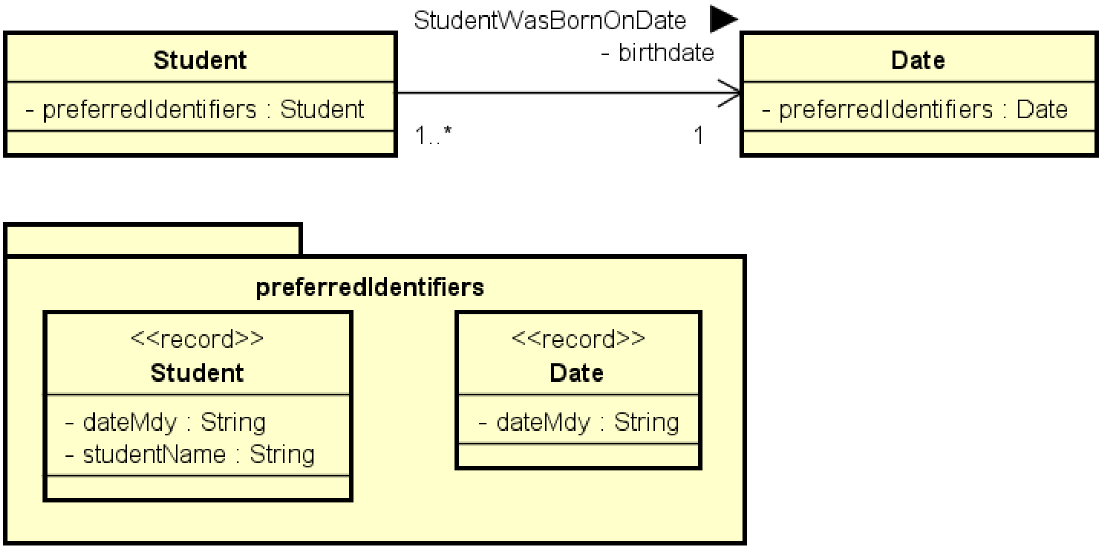


Figure 3.5: The grUML schema that results from mapping the ORM schema from figure 3.4.

of an `EventDescription`, which in this case is a more specific `ChangeAttributeEventDescription` that monitors a specified attribute of a defined vertex class for changes. If this attribute is changed, the event is fired and the `PreferredIdentifierAttributeChangedAction` is performed. Note that there is no condition that is checked before the action is performed. The reason being that there are multiple conditions that need to be checked, so they are included into the `PreferredIdentifierAttributeChangedAction`.

`PreferredIdentifierAttributeChangedAction` which implements JGraLab's `Action` interface, does one of two things: it either accepts the change within a "preferredIdentifiers" record and may propagate this change to further records that contain the same component (compare to "dateMdy" whose change needs to be propagated to the "preferredIdentifiers" record of "Student") or it might reject the change. In order to decide on one of the two actions, `PreferredIdentifierAttributeChangedAction` first checks two conditions: whether the graph is valid at the moment the change occurs (implemented in `GraphValidityCondition.java`) and whether the change is valid with respect to the uniqueness of the value of the updated preferred identifier component (implemented in `PreferredIdentifierAttributeChangedCondition.java`). If both conditions are met, the component within the "preferredIdentifier" record is updated and the change is propagated if necessary. Whether a propagation is necessary or not is determined during construction of `PreferredIdentifierAttributeChangedAction`. The constructor allows the definition of a neighboring vertex class and the edge class connecting it to the vertex class of the vertex in question. Using this information, it is possible to access the neighboring instances of this defined vertex class and to update their "preferredIdentifiers" records.

If a change is invalid, the "preferredIdentifier" record is restored to the state before the change.

Though the code for the ECA rules is available, there is no framework to automatically add them to the vertices of growing TGraphs or TGraphs which are loaded from file. Furthermore, to this point there is no method in place to automatically extract the vertex and edge class that will be needed for the correct propagation of "preferredIdentifiers" components.

3.3.2 Validating TGraphs

TGraphs' correctness with respect to grUML constraints and edge class multiplicity can be validated using JGraLab's `GraphValidator` and its method `validate`. Since there is no framework for TGraph generation in place, this too needs to be added manually by anyone generating instances from grUML schemas created by the ORM to grUML mapper `Orm2Tg`.

It is also necessary to briefly consider the way in which TGraphs are generated at the moment. JGraLab provides two APIs for graph generation. Both involve Java code that defines a TGraph and subsequently stores it in ".tg" format. The second option involves specifying the TGraph's components, i.e. edges and vertices, in `csv` files and generating a TGraph from this using JGraLab's `Csv2Tg.java`. This approach too will produce a ".tg" file containing the TGraph.

One question to address with respect to validating TGraphs, is *when* to validate the TGraph. This is also the point at which the differentiation between *deontic* and *alethic* constraints will come into play. The following sections will provide a few thoughts on both aspects.

When to validate a TGraph

There are two situations in which validation is warranted: when loading or saving a TGraph, it is important to know the status quo in order to be able to work with or store a valid TGraph. Since none of the approaches for generating TGraphs is interactive in nature, there is no way of validating the TGraph during its creation (apart from generating it in small increments).

If at some point there should be an interactive TGraph editor, ECA rules would provide a way of continually validating the graph. An ECA rule could respond to changes in the graph by running the `validate` method of `GraphValidator`. However, this may be time consuming if TGraphs grow large and constraint violations that are found cannot interfere with a TGraph editor's execution.

Alethic and deontic constraints

Alethic constraints are constraints which must be met. One way of implementing this behavior would be refusing to save a TGraph if the `validate` method of `GraphValidator` returns a set of size ≥ 0. The error message would have to be expansive enough so the user knows what to correct within the graph. If a TGraph contains violations after loading, this should have no influence on the ability to make changes to the graph. The user should however be informed of the violations.

Deontic constraints are suggestions which don't have to be followed. In this case, it would suffice to provide the user with a list of constraint or multiplicity violations upon loading and saving the TGraph.

3.4 Final result

`Orm2Tg` performs the mapping of ORM schema files created in NORMA to grUML schemas in ".tg" format. To achieve this, it uses an `OrmParser` which parses an ORM schema file and stores the data in form of a TGraph. This data can subsequently be used by `Orm2Tg` to generate a grUML schema by following the transformation steps detailed in chapter 2. The mapping procedure implemented during this thesis correctly maps entity types, objectifications, value types, subtype relations and fact types to grUML. To this point, the ORM constraints that can be successfully mapped to grUML constraints are the following: ring constraints, internal uniqueness constraints, internal mandatory constraints, value constraints and inclusive-or constraints.

At TGraph level, ECA rules which perform the actions defined in `PreferredIdentifierAttributeChangedAction` can be added manually to each vertex class. This ensures that changes in preferred identifiers maintain their uniqueness and are propagated if necessary.

Lastly, the constraints added to a grUML schema can be validated upon loading or storing a TGraph created from this schema by using JGraLab's `GraphValidator`.

4 Discussion

This chapter will discuss the results presented in the chapters 2 and 3. The discussion will begin with the theoretical mapping and then move on to the implementation.

4.1 Theoretical mapping from ORM to grUML

Defining the mapping from ORM to grUML on a theoretical basis was approached one ORM element at a time. This ensured that each element would have a correspondence in grUML. The only requirement for this approach was a good understanding of the semantics of both the ORM elements and the candidate grUML elements. However, as the mapping evolved, it became apparent that this path alone was misleading. During the generation of example schemas to better understand the ORM schema file format, more and more possibilities of combining ORM elements were discovered, which required additions and changes to the first drafts of the mapping.

In [HM08], chapter 9.8, Halpin describes a mapping from ORM schemas to UML class diagrams called *UMLmap*. Since UML provides notation to express some of ORM's constraints (e.g. exclusive-or constraint) which grUML does not, this procedure could only serve as inspiration for the mapping defined in chapter 2.

Intriguingly Halpin's mapping will turn entity types into attributes and value types into classes in specific cases. The first will happen if an entity type A has an n:1 or 1:1 relation with an entity type B (that plays no further fact roles), then A maps to a class and B becomes A's attribute. The second case arises if a value type plays an explicitly mandatory role, is independent or takes part in a n:1 relation.

The following two sections will expand on these two cases and show examples for each of them. Further examples for Halpin's mapping process from ORM to UML can also be found in [HM08], chapter 9.8. In this chapter, the process is not discussed in great detail, so the mapping for certain situations, e.g. relations containing only value types or subtype definitions between value types, remain unspecified.

4.1.1 Mapping entity types to attributes

If an entity type has an n:1 or 1:1 relation with another entity type which plays no further fact roles, then Halpin suggests turning the latter into an attribute of the class created from the former.

In the mapping proposed in this work, entity types never map to attributes. Although it makes sense in the cases presented in Halpin's book, there are two reasons why these exceptions were not introduced in this thesis.

Firstly, the semantics of entity types are different from the semantics of an attribute. An entity type represents a group of concrete objects that are subsumed by the entity type. Entity types may participate in several relations apart from those that are required to uniquely identify them. An attribute on the other hand only serves the purpose of defining a value by giving it a name and a domain and binding this information to a class. Attributes cannot participate in any other relations.

The second reason for consistently mapping entity types to vertex classes is the aim to keep the mapping a simple as possible. This has several advantages. Understanding a mapping in which each rule has at least one exception makes it considerably more complex and thus harder to understand, implement and finally also to maintain (both at the theoretical and the implementation level).

Figure 4.1 shows an adaptation of a situation in which Halpin proposes mapping an entity type to an attribute. In this case, he proposes mapping both "Date" and "Gender" to attributes of the related entity type "Student". For this specific scenario, mapping these entity types to attributes does seem to be the preferred solution. But possibly it would be a better choice to remodel these entity types as value types, since in this schema, they only serve the purpose of carrying information in form of

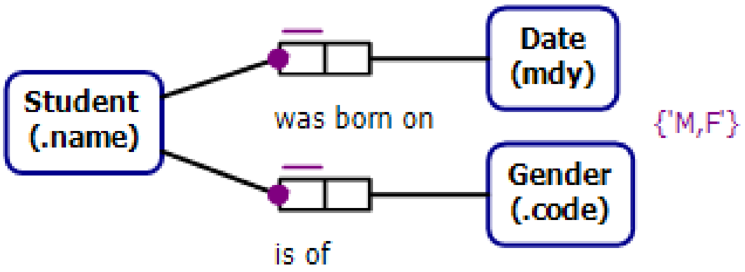


Figure 4.1: This ORM schema depicts a situation in which Halpin proposes to map "Date" and "Gender" to attribute of the class generated from the entity type "Student". The schema is based on the schema in figure 9.56 from [HM08].

values (the gender code or the date in a specific format). Maybe it would be a good idea to generally revise schemas in which entity types only play a single fact role and remodel these as value types. But in the end, it must be assumed that the ORM schemas going into the mapping process are generated carefully and intentionally in order for the mapping process defined in this work to make sense.

4.1.2 Mapping value types to vertex classes

Halpin proposes turning a value type into a vertex class in specific situations, e.g. when it participates in an n:1 relationship or it plays an explicitly mandatory role. Unfortunately the author doesn't expand on the reasoning for this mapping.

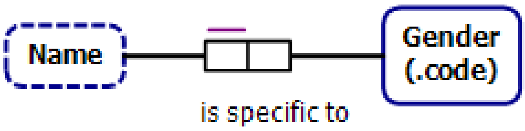


Figure 4.2: An ORM schema with a value type that plays a role in a n:1 relationship with an entity type. In this case Halpin proposes transforming the value type into a class and the entity type is mapped to an attribute of this class. This example is based on the schema in figure 9.57 from [HM08].

Figure 4.2 is a variation on one of the examples Halpin gives in his book in which the value type "Name" takes part in an n:1 relationship. The mapping he suggests is to create the class "Name" from the value type which holds an optional attribute "specificToGender" that is generated from the entity type "Gender". Figure 4.3 shows the way this schema is mapped using the mapping procedure suggested in this work. The outcome is very similar to Halpin's transformation but without requiring an exception to the rule of always mapping value types to attributes.

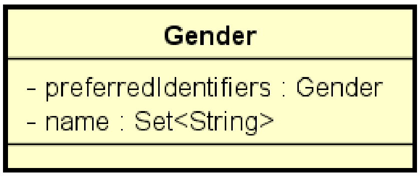


Figure 4.3: The mapping of the schema presented in figure 4.2 according to the procedure described in this work. The value type "Name" becomes an attribute of the vertex class "Gender" which results from mapping the entity type "Gender". The package containing the records for the preferred identifiers is not shown for simplicity reasons.

The third scenario in which Halpin maps a value type to a vertex class is a case with which the mapping defined in this thesis struggles: independent value types.

However, as mentioned in section 2.9 this is considered to be a case which rarely occurs. Seeing that schemas with independent value types need to be rejected at this point, it may nevertheless be desirable for the future to make an exception for these ORM elements. This would still align with the general requirement for a simple mapping since this situation is most likely very rare.

4.1.3 Constraints

ORM is a highly flexible modeling language and concordantly, each constraint has a variety of applications - e.g. in one schema it might apply to roles of a binary fact type and in another it might constrain roles of a ternary fact type. The schemas resulting from mapping a fact type with arity 2 differ substantially from that of a schema with arity 3. Thus, the difficulty in mapping the ORM constraints lies in the question whether all possible application scenarios can be anticipated. Each scenario comes with its own mapping to grUML which might call for a modification or redesign of the GReQL expressions within the grUML constraint. Due to the complexity of this undertaking, only a subset of ORM's constraints were included in the implementation for this work.

With a growing understanding of the possibilities ORM has to offer, it is difficult to assess to what extent the possible scenarios for each constraint were considered. In addition, the fact that there is no fast or easy way of creating example TGraphs from grUML schemas made the process of defining GReQL constraints and testing them a very tedious endeavor.

Although it is unlikely that all situations were foreseen for each constraint, it is probably safe to say that the most common situations were taken into account. Certainly all relevant schemas in [HM08] were taken into consideration.

Halpin doesn't go into details about the mapping of the constraints in his UMLmap procedure. He suggests adding the constraints to the UML class diagram in textual form and does not elaborate on how they could be realized. Chapter 11 of his book [HM08] is concerned with the mapping procedure from ORM to the relational schemas. Within this chapter, he mentions that outlining the process of mapping the ORM constraints to a relational schema would exceed the scope of the book. Furthermore, none of the tools that are publicly available (e.g. NORMA) provide support for more than the mapping of uniqueness and mandatory constraints. This supports the notion that mapping ORM constraints is a complex endeavor.

4.1.4 Derivations

The main arguments for dismissing the mapping of derivations and semiderivations are mentioned in section 2.12. In brief, the low user-friendliness of NORMA's interface for creating (semi-) derivations, along with the large number of possibilities ORM has to offer in this respect and the missing XSD to help interpret the ORM schema file, rendered this task too time-consuming.

Although this aspect of ORM could not be included in this thesis, it should easily be possible to extend the parser and mapping software that resulted from this work in order to include (semi-) derivations. A new approach could be to collect models created by professionals that work with ORM in order to get an overview of the most commonly used derivations and to categorize these. Like this it may still not be possible to cover all the possibilities ORM has to offer but at least the most commonly used cases.

4.1.5 Mapping to a composition relationship

GrUML offers three kinds of relationships between class instances: the association, the aggregation and the composition. In the mapping procedure described in chapter 2, fact types that did not exclusively contain value types were defined to be mapped to associations between vertex classes by default.

The association is the most general form of relation. Both aggregation and composition are used to define a whole/part relationship but the composition reflects a stronger form of ownership where deletion of the composite, i.e. the whole, will lead to the deletion of its parts. This is the theoretical understanding of a composition and it is a question of implementation whether or not the members of the composition behave in this way. As for grUML, a composition defined in a schema can be instantiated within a TGraph and deleting the composite from the graph has the effect of removing the components too.

ORM's notation does not provide a way for a modeler to explicitly define a whole/part relation. This information is implied by the uniqueness and mandatory constraints and the predicate reading of a fact type. Concordantly, automatic mapping to compositions or aggregations is not possible.

One possibility to still use these elements grUML provides is for the modeler to attach notes to a fact type within his model stating “composition” or “aggregation” and the direction of this relation. Of course, this only applies to binary fact types.

4.1.6 grUML elements that could not be included

This section briefly lists the grUML elements which could not be included as a result of mapping. Firstly, grUML's *packages* are only used to organize the records holding vertex class' preferred identifiers. Beyond this task, they could not be used since there was no way of categorizing ORM elements based on their semantics in an automated fashion.

As discussed in the previous section, aggregations and compositions are currently not included in the mapping but could be included in later versions.

Lastly, some of grUML's domains are not included in the mapping, like the map domain or the enumeration domain. While there is no obvious application for the map domain, the enumeration domain could be used as an alternative to a grUML constraint for value constraints which define a limitation to a small number of values. In these cases, it would be possible to restrict the values of an attribute generated from a constrained value type to the values defined within the enumeration.

4.1.7 What is lost during mapping

After discussing which grUML elements could not be used during the mapping process, it is also important to discuss what is lost during mapping. ORM is a semantic modeling approach and concordantly its schemas contain a lot of information. Although the mapping was carefully designed with the aim of being able to map as many schemas as possible and at the same time minimizing the amount of information that is lost during the mapping process, the result is a compromise.

Not all ORM schemas can be mapped because for some situations there is no way of transferring the semantics to grUML without making the mapping much more complicated - e.g. subtype definitions between value types.

ORM relations can have multiple predicates - one for each reading order or even multiple predicates for one reading order. Since grUML edges are directed, one of these predicate readings has to be selected as the name for the edge class (or vertex class) that results from mapping. Seeing that there is no simple way of establishing which predicate reading contains the most information, the mapping procedure just selects the first listed in the ORM schema file. If multiple predicate readings are provided, this will lead to loss of information. One idea to reduce this information loss might be to add a note to the grUML schema containing the additional predicate readings.

The relations between entity types and value types map to vertex classes that hold attributes. The relation's predicate is not used to generate the attribute's name, so this information is lost too. This also holds true for role names that might apply to such a relation. In terms of the predicate reading, please consider the following relation "Student(.ID) has Name" where "Name" is a value type. Including the predicate reading in the attribute name would lead to the attribute "hasName" for the vertex class "Student". This raises the expectation that this attribute represents a boolean value, although it is actually a string. Since relations between value types and entity types often express the ownership or the connection between the entity type and the value type, and this is expressed using a rather fixed set of predicates, it was considered acceptable to drop the predicate reading.

Although this may appear to be a rather big number of losses, it is important to question how often these situations arise and how grievous the consequences really are.

Not being able to map certain schemas and loosing additional predicate readings are considered to be the worst of the losses discussed above. However, extensions to the mapping definition provided in this work can increase the amount of accepted ORM schemas. And the additional predicate readings can be stored in the grUML schema in form of a note - possibly granting the user a choice of which predicate reading he deems most meaningful.

4.1.8 Completeness and Correctness

The requirements for the mapping from ORM to grUML were *completeness* and *correctness.*

Completeness

The mapping defined in chapter 2 is incomplete. It is missing derivations and semiderivations for reasons discussed previously. It might also be incomplete with respect to ORM constraints and the many scenarios they can apply to.

Correctness

From a theoretical viewpoint, the mapping as defined in chapter 2 is correct. Each ORM element was analyzed with respect to its semantics and it was attempted to find an equivalent for this in grUML. Finding a match was not always possible (see e.g. subtype constraints) but these exceptions are few and they were made deliberately. Additionally, the user receives a warning, whenever the expressiveness of grUML does not suffice.

However, as discussed in the previous section, it could be that some scenarios were not considered rendering the mapping incomplete and thus incorrect.

4.2 Implementation

4.2.1 ORM schema parser

One result of this work is an ORM schema parser which consistently parses ORM schema files and sets the foundation for the actual mapping procedure. It is able to read all the information contained in an ORM schema with the exclusion of derivation rules. If the parser encounters a derivation rule, it rejects the schema.

Seeing that the parser is based on a possibly incomplete understanding of the structure of ORM schema files, there is no guarantee that it will function for every possible ORM schema without derivations. It has however been tested using over a hundred schema files and for these, it worked without problems.

A major difficulty with writing the ORM schema parser was the fact that there was no XSD file available. And even after understanding the file format through the generation and analysis of numerous example files, the ORM schema files still showed some unexpected behavior. A few examples of unexpected behavior are listed below.

- Binary relations with a spanning uniqueness constraint were represented by objectifying the binary relation. But binary relations with uniqueness constraints that applied to only single roles were represented as binary relations without an objectification.

- Unary relations are represented as binary relations between the player of the unary role and a second, pseudo value type that stands for a boolean value to indicate whether the role is played or not.

Ultimately, this work and the task of writing an ORM schema parser would definitely have profited from a XML schema definition for the ".orm" file format but the most value would have been added through a *good documentation* of the file format.

4.2.2 Mapping the data to grUML

The mapping process generates grUML schemas using the data parsed from an ORM schema file. Due to the very simple nature of the mapping proposed in this thesis, the main components like object types, fact types and subtype relations can be mapped reliably. Problems may arise when mapping constraints.

Generating grUML constraints

In this section, various aspects that posed problems during the generation of grUML constraints will be discussed.

Verbalization The first part of grUML constraints is a message which can be returned to the user whenever a TGraph is in violation of this constraint.

For the grUML constraints generated in this work, the wording was extracted from NORMA by creating constraints in various different schema constellations and finding rules by which the verbalization was generated. This means that the verbalization of constraints will work for many situations (namely those which were tested) but in other cases it may fail to adequately express the meaning of the constraint at hand. For these cases the verbalization always includes the name of the ORM constraint which still can help the user find the source of the violation.

Going forward, it will be difficult to create a flexible framework for generating constraint verbalizations in the way NORMA does without knowing more about their verbalization implementation.

GReQL expression As mentioned in the chapter on mapping, the GReQL expressions for a specific kind of constraint may change substantially if the context of the constraint is changed.

Which constraints need to be explicated? The final task regarding grUML constraints was identifying which constraints even required an explicit grUML constraint. In order to do this, it is necessary to establish the context in which the ORM constraints can be applied. Considering the way in which these contexts are mapped, it has to be determined whether the constraint needs to be explicated or is already implicitly handled through e.g. multiplicity values or the use of sets for multivalued attributes.

4.3 Generating instances

After being able to map an ORM schema to a grUML schema, one of the main goals for this work was to be able to generate TGraphs from such a schema and to validate their correspondence to the rules provided in the ORM schema.

This goal is reached but still requires a lot of know-how from the user since there is no comprehensive framework which provides e.g. graph validation when loading or saving a graph or which automatically adds ECA rules where needed.

4.4 Outlook

There are several aspects that could be improved and some of them were already discussed throughout this chapter. This section should give a short outline of the most important steps that need to be taken moving forward.

The first step should be implementing the remaining grUML constraint generating classes. ORM's *set comparison constraints*, the *value comparison constraints* and the *interpredicate constraint* are not yet handled.

The second major step would be the establishment of a framework that automatically adds ECA rules to a TGraph an makes validity checks.

A minor point might be the extension of the domains supported by grUML in order to be able to handle raw data value types. In terms of value type domains, another minor improvement might be the implementation of grUML constraints that ensure a certain length or precision of values.

Finally, a new version of ORM was announced to be released later this year. In light of this development, it might be necessary to reevaluate the next steps once the extent of the changes is known. This change may be accompanied by an improved documentation or even the release of the ORM2 metamodel. The latter could help improve this project by clarifying the possible relationships between ORM schema elements, especially derivations and their components, but also by providing a vocabulary which is possibly used in the ORM schema files too.

5 Conclusion

The goal of this master thesis was to represent instances of schemas supplied in the modeling language ORM in the form of graphs and more precisely TGraphs. In a first step to achieve this goal, the ORM schemas needed to be mapped to the TGraph schema language grUML. Defining the mapping between ORM schema elements and the elements of grUML, a profiled version of UML 2 class diagrams, required a deep understanding of both schema languages.

Chapter 2 introduced the rules for mapping ORM schemas to grUML. Though this mapping is not complete - it does not handle derivations and semiderivations and some scenarios simply can not be represented in grUML - it still defines a way to map the most common ORM schemas to grUML.

Chapter 3 gives insight into the challenges and problems implementing this mapping and generating grUML schema instances. At the implementation level, not all ORM constraints could be translated into grUML constraints. The reason lies in ORM's flexibility: each constraint can apply to many different scenarios which results in different GReQL expressions and verbalizations.

The goal for this thesis of representing ORM schema instances in form of TGraphs was reached for a reduced set of ORM schema files.

In conclusion, this work shows that it is possible to map ORM schemas to grUML schemas and to generate TGraph instances that are in accordance with the original ORM schemas. Although this is momentarily only true for a subset of ORM schemas, the reason lies in the limited time frame for this work and not in the lack of feasibility of this approach.

A widely accepted vehicle for representing ORM schema instances is relational tables. Besides their static format, the major drawback of relational tables is the insufficient support for ORM constraints. Although Terry Halpin mentions a mapping for ORM constraints to relational schemas, he considered it too elaborate to include in his book ([HM08]) and it is not featured in NORMA, which provides a mapping from ORM to relational schemas.

Storing ORM instances in TGraphs allows the definition of grUML constraint using the TGraph query language GReQL. Although not all constraints could be implemented within the scope of this thesis, the reason lies in ORM's high complexity and not in an insufficient expressiveness of GReQL. Furthermore JGraLab provides methods to efficiently access TGraph elements.

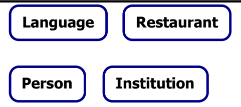
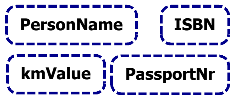
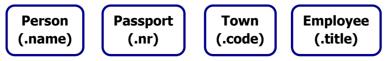
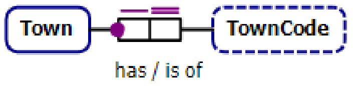
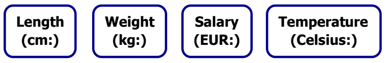
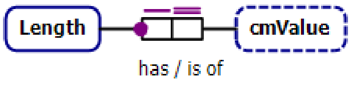
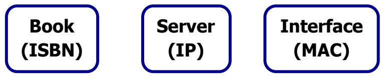
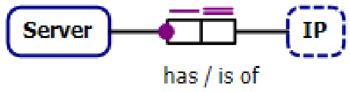
The results of this thesis provide support for an alternative, graph-based approach to storing ORM schema instances which also provides a powerful way of representing ORM constraints in the form of grUML constraints. Using TGraphs and their query language GReQL to realize these constraints, allows quick access to TGraph elements and a fast evaluation of constraints.

Appendices

A ORM notation

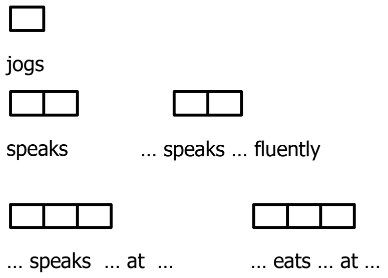
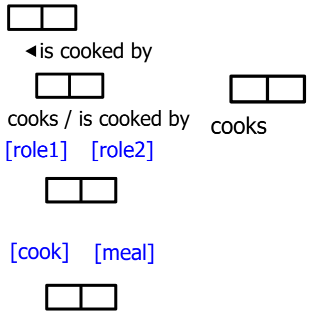
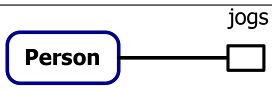
This section provides a summary of the graphical notation for the modeling elements introduced in sections 1.5.1 to 1.5.7. The notation shown here is the notation currently provided by the NORMA tool. Table ?? gives an overview of the basic notation elements as introduced in 1.5.1.

Table 1: Table of the basic elements of ORM's notation as used in NORMA.

Name	Example	Description
entity type		An entity type is shown as a named, soft rectangle
value type		A value type is shown as a named, soft, dashed rectangle
entity type with popular reference mode	 Person (.name) Passport (.nr) Town (.code) Employee (.title)	Abbreviated 1:1 reference scheme for frequently used value types. Explicitly looks like e.g. 
entity type with unit-based reference mode	 Length (cm:) Weight (kg:) Salary (EUR:) Temperature (Celsius:)	Abbreviated 1:1 reference scheme for unit value types. Explicitly looks like e.g. 
entity type with general reference mode		Abbreviated 1:1 reference scheme for general value types. Explicitly looks like e.g. 

Continued on next page

Table 1 – continued from previous page

Name	Example	Description
n-ary predicates Here: $n \in \{1, 2, 3\}$	 jogs speaks … speaks … fluently … speaks … at … … eats … at …	A predicate is a sequence of role boxes that correspond to roles which are played by object types. It must have at least one reading which can be in mixfix notation. If there is no object type placeholder "…", unary predicates are in prefix and binary predicates in infix notation.
predicate readings	 ◂is cooked by cooks / is cooked by cooks [role1] [role2] [cook] [meal]	Predicates can have a forward and a backward reading. At least one reading needs to be provided. If both are provided, they are separated by a '/'. The reading direction is typically left to right or top to bottom. If this doesn't apply, an arrow symbol indicates the reading direction. Roles can be annotated with role names in blue color and square brackets.
unary fact type		Displays the fact type 'Person jogs'

Continued on next page

Table 1 – continued from previous page

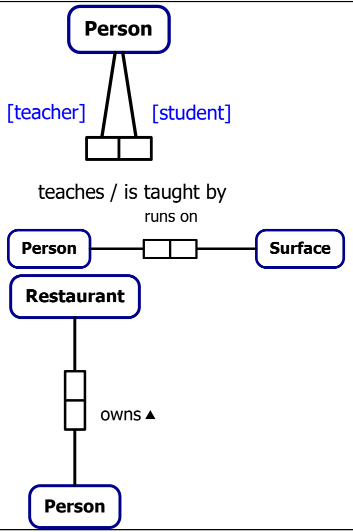
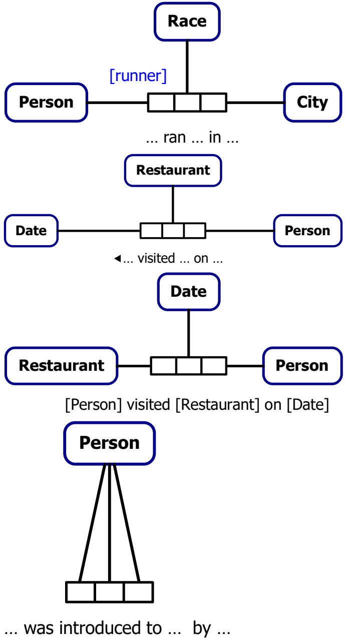
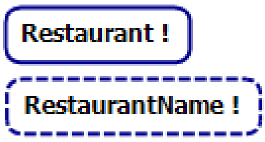
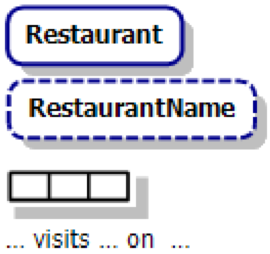
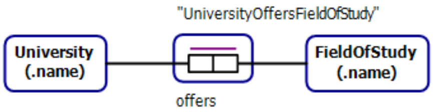
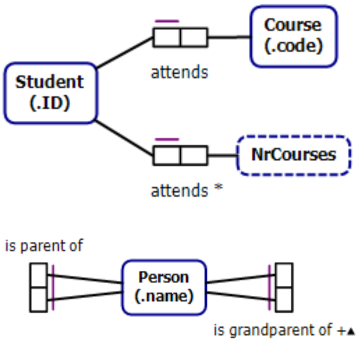
Name	Example	Description
binary fact type	 Person [teacher] [student] teaches / is taught by runs on Person Surface Restaurant owns ▲ Person	Examples of binary fact types.
ternary fact type	 Race [runner] Person City ... ran ... in ... Restaurant Date Person ◄... visited ... on ... Date Restaurant Person [Person] visited [Restaurant] on [Date] Person ... was introduced to ... by ...	Examples of ternary fact types. If a predicate reading doesn't use the object types in the order of their role boxes, the object type placeholder "..." is replaced by the object type name in square brackets.

Table 2: Table of further elements of ORM's notation as used in NORMA.

Name	Example	Description
Independent object type		Instances of these object types can exist without playing a role in a fact

Continued on next page

Table 2 – continued from previous page

Name	Example	Description
duplicate object type or predicate		An object type or predicate can appear in a schema multiple times and is displayed with an underlying shadow to indicate this.
objectification	 "UniversityOffersFieldOfStudy" University (.name) FieldOfStudy (.name) offers	A fact type can be objectified and the resulting entity type (here “UniversityOffersFieldOfStudy”) can take part in further relations (unless it is declared independent)
derived fact types (derivation rule not included)	 Course (.code) attends Student (.ID) NrCourses attends * is parent of Person (.name) is grandparent of +▲	Facts types are either asserted, derived or semiderived. Derived fact types are indicated by an asterisk ”*” following the predicate reading. A double asterisk ”**” indicates that the fact type is derived and stored. Semiderived fact types are indicated by a ”+” symbol after the predicate reading. ”++” stands for a semiderived fact that is stored.

Continued on next page

Table 2 – continued from previous page

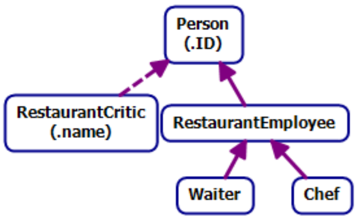
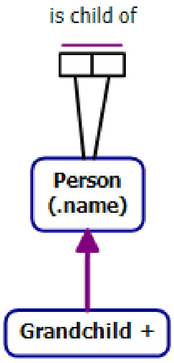
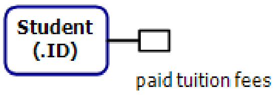
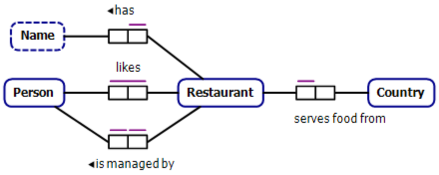
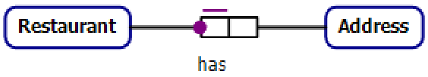
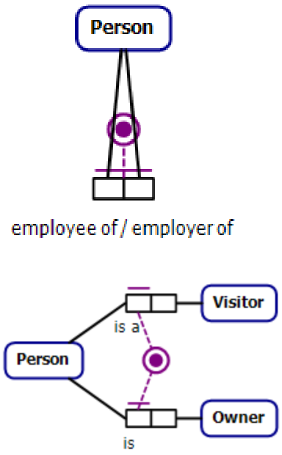
Name	Example	Description
subtyping	 Person (.ID) RestaurantCritic (.name) RestaurantEmployee Waiter Chef	An arrow going from one entity type to another indicates a subtype relation between the origin entity type and the target entity type. A solid arrow means that the subtype has the same preferred identifier as its supertype. A dashed arrow indicates that the subtype's preferred identifier differs from that of its supertype.
derived subtypes		Subtypes can be asserted, derived or semi-derived. Their derivation status is indicated by a "+"(semi-derived) or a "*" (fully derived) following the subtypes name. In this example, the subtype Grandchild is semiderived. It can be defined directly or through derivation (**Each derived** Child **is a** Person **who** is a child of **some** Person **who** is a child of **some** Person.).

Table 3: Table of ORM constraints as used in NORMA.

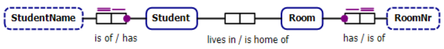
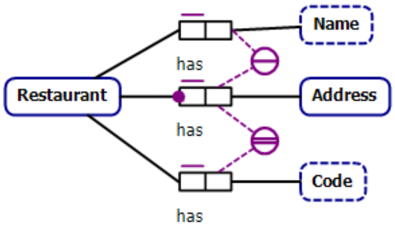
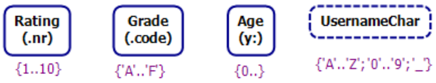
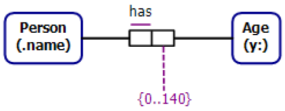
Name	Example	Description
		Continued on next page

Table 3 – continued from previous page

Name	Example	Description
internal uniqueness constraint on unaries		Internal uniqueness constraints are displayed as lines above the roles they apply to. The uniqueness constraint on unary relations is implied and thus not displayed here, since facts are not allowed to be duplicated.
internal uniqueness constraint on binaries	 ◂has Name likes Person Restaurant Country serves food from ◂is managed by	An internal uniqueness constraint must span at least $n - 1$ roles of a fact type. This makes 4 possible combinations of uniqueness constraints on binary fact types.
mandatory role constraint		The mandatory role constraint indicates that the constrained role must be played by the object type hosting it. It is indicated through a purple dot on the role box.
inclusive-or constraint (disjunctive mandatory role constraint)	 Person employee of / employer of Visitor is a Person Owner is	The inclusive-or constraint is a purple dot within a circle that is connected to two or more roles played by the same object type. It means that this object type must play at least one of these roles.

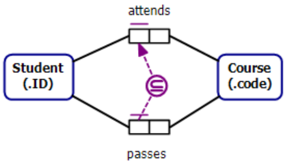
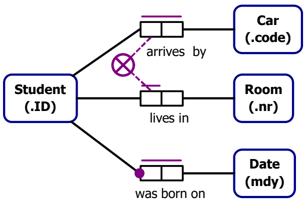
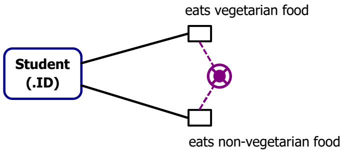
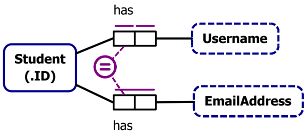
Continued on next page

Table 3 – continued from previous page

Name	Example	Description
preferred internal uniqueness constraint	 StudentName is of / has Student lives in / is home of Room has / is of RoomNr	The second bar above the internal uniqueness constraint bar indicates that the value type hosting this role it the preferred reference scheme for the entity type playing the other role.
external uniqueness constraint	 	This constraint indicates that the combination of the constrained roles is unique for all instances of Restaurant. The double bar indicates that the combination of Address and Code is the preferred reference scheme for Restaurant.
object Type value constraint	 Rating (.nr) {1..10} Grade (.code) {'A'..'F'} Age (y:) {0..} UsernameChar {'A'..'Z';'0'..'9';'_'}	Objects may only take on the values defined in the value constraint. The values can be provided in the form of an enumeration (1,2,3) or a range (50..100) or a combination of both.
role value constraint	 	Objects playing the constrained roles may only take on the defined values.

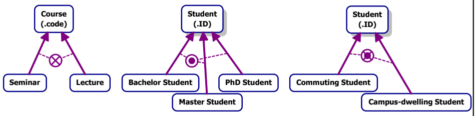
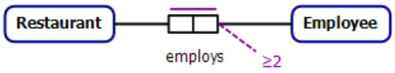
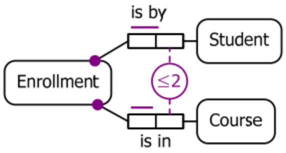
Continued on next page

Table 3 – continued from previous page

Name	Example	Description
subset constraint		This constraint is directed and points from the subset to the superset. Subset and superset instances must be compatible. This constraint can also apply to role sequences of compatible object types.
exclusion constraint	 Car (.code) arrives by Student (.ID) Room (.nr) lives in Date (mdy) was born on	This constraint is used to indicate that the constrained roles exclude each other – i.e. it is not possible to play all constrained roles.
exclusive-or constraint	 eats vegetarian food Student (.ID) eats non-vegetarian food	This constraint is the combination between the disjunctive mandatory role constraint and the exclusion constraint: at least one role must be played and not all roles can be played. This constraint can also apply to role sequences of compatible object types.
equality constraint		This constraint indicates that the populations of the constrained roles must be equal. This constraint can also apply to role sequences of compatible object types.

Continued on next page

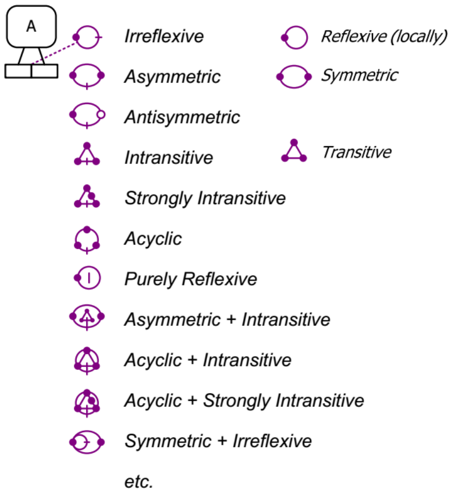
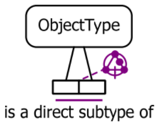
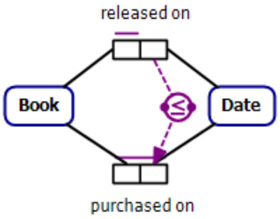
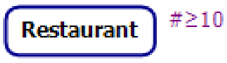
Table 3 – continued from previous page

Name	Example	Description
subtype constraints	 Course (.code) Seminar Lecture Student (.ID) Bachelor Student PhD Student Master Student Student (.ID) Commuting Student Campus-dwelling Student	The circled ”X” indicates that the subtypes are mutually exclusive. The circled dot indicates that the supertype is the union of its subtypes. The combination of the two previous constraints is an XOR constraint and indicates that the subtypes partition the supertype.
internal frequency constraint		This constraint limits the number of times a certain instance of a role or role sequence can occur in the role/role sequence’s population. In this case each Employee works for at least two restaurants.
external frequency constraint		This constraint limits the number of times a certain combination of instances of a roles or role sequences can occur with respect to the existence of their common object type. In this case this means that there can be at most two enrollments by the same combination of student and course. This means that a student can only enroll into a specific course at most two times. [1]

Continued on next page

[1] Source of figure: `http://www.orm.net/pdf/ORM2GraphicalNotation.pdf`

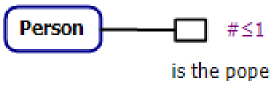
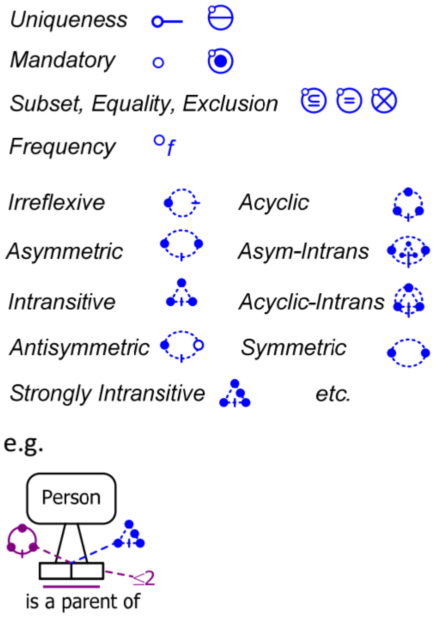
Table 3 – continued from previous page

Name	Example	Description
ring constraints	 A Irreflexive Reflexive (locally) Asymmetric Symmetric Antisymmetric Intransitive Transitive Strongly Intransitive Acyclic Purely Reflexive Asymmetric + Intransitive Acyclic + Intransitive Acyclic + Strongly Intransitive Symmetric + Irreflexive etc. E.g. 	Ring constraints are used to further define the ring relation. [2]
value comparison constraints		This constraint is used to enforce certain relations between the values of the objects playing the constrained roles. In this example, the release date of a book must be $\leq$ the purchase date. The other operators that can be used are $\leq$, $\geq$, $\geq$, $=$ and $\neq$.
object cardinality constraint		This constraint limits the number of instances an object type can have.

Continued on next page

[2] Source of figure: http://www.orm.net/pdf/ORM2GraphicalNotation.pdf

Table 3 – continued from previous page

Name	Example	Description
role cardinality constraint		This constraint limits the number of instances that can play a certain role. This schema indicates that there can only be at most one pope at any point in time.
deontic constraints	 Uniqueness Mandatory Subset, Equality, Exclusion Frequency Irreflexive Acyclic Asymmetric Asym-Intrans Intransitive Acyclic-Intrans Antisymmetric Symmetric Strongly Intransitive etc. e.g. Person ≤2 is a parent of	Deontic constraints are obligatory but not mandatory. They are colored blue rather than violet and contain an ”o” for obligatory. Deontic ring constraints have a dashed line. [3]

[3] Source of figure: `http://www.orm.net/pdf/ORM2GraphicalNotation.pdf`

B Mapping ORM to grUML

This section provides additional information concerning the mapping of ORM schemas to grUML schemas. The basics of this mapping were introduced in chapter 2.

Table 4 shows the mapping of ORM reference mode value domains to grUML value domains.

Table 5 shows the mapping of the ORM value type domains to grUML value domains.

Table 4: ORM reference modes and their value domains mapped to grUML value domains

ORM reference mode	**ORM value domain**	**corresponding domain in grUML**
.ID/.Id/.id	Numeric: Auto Counter	Integer
.#	Numeric: Signed Integer	Integer
.code/.Code	Text: Fixed Length	String
.name/.Name	Text: Variable Length	String
.nr/.Nr	Numeric: Signed Integer	Integer
.title/.Title	Text: Variable Length	String
AUD:	Numeric: Money	Double
CE: [common era]	Temporal: Date	String
Celsius:	Numeric: Decimal	Double
cm:	Numeric: Decimal	Double
EUR:	Numeric: Money	Double
Fahrenheit:	Numeric: Decimal	Double
kg:	Numeric: Decimal	Double
km:	Numeric: Decimal	Double
mile:	Numeric: Decimal	Double
mm:	Numeric: Decimal	Doube
USD:	Numeric: Money	Double

Table 5: ORM reference modes and their value domains mapped to grUML value domains

ORM value domain	corresponding domain in grUML
Logical: True or False	Boolean
Logical: Yes or No	Boolean
Numeric: Auto Counter	Integer
Numeric: Decimal	Double
Numeric: Float (Custom Precision)	Double [1]
Numeric: Float (Double Precision)	Double [1]
Numeric: Float (Single Precision)	Double [1]
Numeric: Money	Double
Numeric: Signed Big Integer	Long
Numeric: Signed Integer	Integer
Numeric: Signed Small Integer	Integer [2]
Numeric: Unsigned Big Integer	Long
Numeric: Unsigned Integer	Long [2]
Numeric: Unsigned Small Integer	Integer [2]
Numeric: Unsigned Tiny Integer	Integer [2]
Other: Object ID	Integer
Other: Row ID	Integer
Raw Data: Fixed Length	-
Raw Data: Large Length	-
Raw Data: OLE Object	-
Raw Data: Picture	-
Raw Data: Variable Length	-
Temporal: Auto Timestamp	String
Temporal: Date	String
Temporal: Date & Time	String
Temporal: Time	String
Text: Fixed Length	String
Text: Large Length	String
Text: Variable Length	String

[1] An additional constraint is required to adjust the values to the specified precision. This is not implemented in the current mapping procedure

[2] An additional constraint is required to adjust the value range. This is not implemented in the current mapping procedure

Glossary

arity in reference to a predicate, the arity or degree indicates the number of object term holes in the predicate . 9

entity an object that is referenced by an unambiguous description which relates this object to other objects, e.g. the entity “Country that has the CountryCode ’AU”’, “City named ’Auckland”’, “Course with code ’CS2017”’. Entitites can change with time. 8, 12, 36

entity type a kind of entity, e.g. Country, Person, Language, Phone. It’s the set of all possible type instances (used in an information system). 9, 12

fact a proposition that is taken to be true within the relevant business community. A fact either declares that some individual exhibits a property (e.g. Pam is lost), that one or more individuals take part in a relationship (e.g. Pam speaks Spanish, Pam is in France) or that an individual exists (Pam exists).. 8, 12

fact role role in an elementary fact type. Each entity type in a completed ORM schema plays at least one referential role and one fact role (unless the entity type is declared independent). 17

fact table is the result of populating fact types. Each fact type has its own fact table. A fact table has as many columns as the arity the of fact type’s predicate. Each column of the table stands for a role that is played by the object type connected to the role box. When populating a fact type, the objects are placed into the columns corresponding to the roles they play and one row equals one fact.. 14

fact type a kind of fact which includes all instances of this kind of fact. It is an abstraction of a fact instance. E.g. Person is lost is the fact type of the fact instance “Pam is lost”, Person speaks Language is the fact type of the fact “Pam speaks Spanish”, Person is in Country is the fact type of the fact “Pam is in France”. 10, 12

global schema in modeling, large business domains are often divided into smaller segments to simplify the modeling process. Each segment is transformed into a *subschema*. The global schema is the product of merging all subschemas and covers the entire UoD. 17

object individual thing of interest, e.g. a *specific* person or language. In ORM, objects are either entities or values.. 8

object type concept that classifies objects, e.g. Person or Language. 10

predicate in logic a predicate is a declarative sentence with holes in it that can be filled by object terms (object terms refer to a single object in the UoD). E.g. “The Person with Surname ’Walker’ has a Weight of 80 Kilograms.” contains the predicate “... *has*...”. The ellipses are filled with the object terms “The Person with Surname ’Walker”’ and “a Weight of 80 Kilograms”. This is an example of a binary predicate but they can also be unary (one hole for object term), ternary (three holes for object terms), ... Note that there is an ordering of the object term holes. 9, 12

primitive entity type see primitive object type. 17

primitive object type is an object type that is not a proper subtype of any other object type within the schema. Primitve object types are mutually exclusive and can define subtypes. Object types can be entity types or value types. 17

reference mode is the manner in which a value refers to an entity. In the example fact “Person with Surname ‘Walker’ has a Weight of 80 Kilograms.” the entity types Person and Weight are referenced by the values ‘Walker’ and 80, through the reference mode Surname and Celsius. Each entity type has a preferred reference scheme. If a 1:1 reference scheme exists, this is the preferred reference scheme. 10

reference scheme each entity type has at least one reference scheme and exactly one *preferred* reference scheme. When an entity is identified by a single value, this is called a *simple* reference scheme. If the entity is uniquely identified by a single value, this is a *simple 1:1* reference scheme and the reference mode describes how the value relates to the entity. In general a reference scheme describes how an entity type can be identified through its relations . 12, 23

value constant which does not require a description since its reference is clear from the context it is used in. E.g. character strings such as 'AU' or numbers such as 42. 9, 12, 36

value type a kind of value, e.g. CountryCode, PersonName, LanguageName, PhoneNumber. 12

Bibliography

[BHR+10] D. Bildhauer, T. Horn, V. Riediger, H. Schwarz, and S. Strauß. grUML - A UML based modelling language for TGraphs. Technical report, University of Koblenz-Landau, 2010.

[EB10] Jürgen Ebert and Daniel Bildhauer. *Reverse Engineering Using Graph Queries*, pages 335–362. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[EF95] J. Ebert and A. Franzke. A declarative approach to graph based modeling. In G. Tinhofer E. Mayr, G. Schmidt, editor, *Graphtheoretic Concepts in Computer Science*, chapter LNCS 903, page 38–50. Springer, 1995.

[EXP04] *ISO 10303-11:2004, Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual.* 11 2004.

[Hal09] T. Halpin. Object-role modeling. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems.* Springer US, 2009.

[HM08] T. Halpin and T. Morgan. *Information Modeling and Relational Databases, Second Edition.* Morgan Kaufmann Publishers, 2008.

[IDE12] *ISO/IEC/IEEE 31320-2:2012 Information technology – Modeling Languages – Part 2: Syntax and Semantics for IDEF1X97 (IDEFobject).* 09 2012.

[KK01] Manfred Kamp and Bernt Kullbach. GReQL - Eine Anfragesprache für das GUPRO-Repository - Sprachbeschreibung (Version 1.3). Technical Report 8/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

[Oli07] Antoni Olivé. *Conceptual Modeling of Information Systems.* Springer-Verlag Berlin Heidelberg, 1 edition, 2007.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[sC76] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.