Bachelor thesis
Computer Science
Information Science

Radboud University

Transforming ORM into O/RM

Author:
Luuk Scholten
s4126424

First supervisor/assessor:
prof. dr. ir. Th.P. (Theo) van der Weide
Th.P.vanderWeide@cs.ru.nl

Second assessor:
dr. P. (Patrick) van Bommel
P.vanBommel@cs.ru.nl

June 28, 2015

Abstract

Even though Object-Relational Mapping tools partially solve the differences between object-oriented programming and relational databases, the tools do not solve the problem of modelling the persistent layer of an application. These models are often constructed in a conceptual modelling language such as Object Role Modeling. However, the models do not fully take the object model into account and a formalized method for transforming the models into Object-Relational Mapping does not yet exist, which inhibits mathematical reasoning about the results of the transformation.

This thesis provides the first steps towards the formal transformation of Object Role Modeling into Object-Relational Mapping. The transformation is based on a restricted inductive definition of Object Role Modeling, making mathematical reasoning about the transformation result a straightforward process. The transformation method further advances the research on Object Role Modeling and Object-Relational Mapping, and provides a solid basis for a fully automated transformation process.

Contents

Chapter 1

Introduction

Of all major programming paradigms, the object-oriented paradigm is the most popular. Many of the popular contemporary programming languages are either completely object-oriented, or support the concepts of object-oriented programming.

As with the evolution of programming languages, it was predicted that relational persistence platforms such as SQL would be replaced by platforms that are better aligned with the object-oriented programming paradigm. However, this prediction never became reality and many modern applications are built using object-oriented concepts, while still using the very old concept of relational databases. For example, application programming for the Android platform is typically done in Java, an object-oriented language. When on-device persistence is required for these applications, the default choice is the SQLite platform, which is fully supported by the Android platform.

The fact that many applications are programmed in object-oriented languages while the persistence is handled with relational databases introduces two problems. The first problem entails that the object model used in object-oriented programming is conceptually very different from the relational model used in relational databases. A solution to bridge the object-relational divide is essential when both concepts are combined. The second problem is concerned with the modelling of the persistent layer of applications. These models of information systems must also take the object model into account.

Fortunately, approaches towards solutions for both these problems exist. The object-relational divide is often handled with an Object-Relational Mapping tool, which handles persistence and retrieval in the object-oriented code base. The modelling difficulties are partially addressed by conceptual data modelling techniques such as Object Role Modeling (ORM).

The problem that arises from using both these 'solutions' is that there is no formalized method to transform conceptual ORM models into specific

O/RM implementations. While some work exists on the translation from Object Role Modeling to relational databases and the object model through for example UML, little work has been done to combine these two worlds. Furthermore, these existing transformation techniques are described much like a recipe, with a low level of formalization, which inhibits formal reasoning about the results.

We will therefore answer the following research question in this thesis:

> How can we formally transform ORM models into Doctrine2 Object-Relational Mapping?

We first discuss the details of Object Role Modeling and provide an inductive definition of ORM models. We proceed by explaining the relational model and the object model, and define the problems that arise when both models are combined. These problems are then addressed by presenting the often-used solution of Object-Relational Mapping and introduce a specific O/RM implementation called Doctrine2.

We explain our transformation method in the next chapter by providing transformation rules. We finalize this thesis by reflecting on our results and providing directions for future work.

Chapter 2

Object Role Modeling

Object Role Modeling (ORM) is a fact-oriented approach to information modelling, specification, transformation and querying [8]. In ORM, information is modelled in terms of the underlying facts of the universe of discourse. These facts and the constructed rules may be verbalized in a language easily understood by non-technical users of the modelled business domains [9]. ORM treats all facts as relationships, which can be unary, binary, ternary or more generally, n-ary. How these facts are then grouped into structures such as classes is considered an implementation issue that is not relevant to capture the essential business semantics [9]. This abstraction facilitates semi-natural verbalization, which in turn facilitates communication with all (not necessarily technically literate) stakeholders.

The basic building blocks of the Object Role Modeling notation consists of the following elements:

Entity Types Entity types specify the kind of entities being referred to. An entity in this case is a "described object" [7]. A typical example of an entity type is a *Student*.

Value Types Value types are, just like entity types, descriptions of objects. Unlike entity types which rely on value types, value types can be represented in communication directly. An example of a value type is a *Name*.

Fact types Fact types are the glue that connects the entity types and the value types. Fact types describe the relations inside the model. A binary fact type for example is *Person* **has** *Name*.

Constraints Constraints describe the limitations to the fact types. A uniqueness constraint can for example specify that each *Person* can only **have** one *Name*.

2.1 Inductive definition of ORM

A formalised method for describing ORM schemas has existed for a while [1]. These formalised methods evolved into the Predicator State Model (PSM). Models can be formally described in full by these methods, which is useful for mathematical reasoning about the models. Tulinayo, van der Weide and van Bommel [13] are working on an inductive definition of the PSM. An inductive definition of a domain consists of a description of the initial state and describes the construction operators to obtain other states.

There are a couple of advantages to this inductive definition. The first advantage is that an inductive definition of an information structure follows the construction of an information structure supported by construction tools. The second advantage comes from the consequence of the inductive definition. The consequence is that properties of information structures can be proven by structural induction. Simply put, this means that a property has to be proven for the initial state (the empty model), and every construction step. This makes reasoning about model properties very natural and mathematically sound.

The inductive information structure definition is described in definition 2.1.

Definition 2.1. The full inductive definition of ORM.

1. $\emptyset$ is the empty information structure
2. If IS is an information structure, then also the following:
   (a) label type: $IS + Label(N, D, \sigma)$
   (b) fact type: $IS + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)$
   (c) derived type: $IS + Derived(N, \sigma)$
   (d) constraint: $IS + Constr(N, C)$
   (e) specialization: $IS + Spec(N, X, \rho)$
   (f) generalization: $IS + Gen(N, X)$
   (g) extension: $IS + GenExt(N, X)$
   (h) set type: $IS + Set(N)$
3. Only what can be formed by these information structure composition rules is an information structure.

This thesis provides a basic set of transformation rules from the inductive definition of ORM to Object-Relational Mapping. Therefore we will limit the construction rules of the inductive definition to a smaller set of rules.

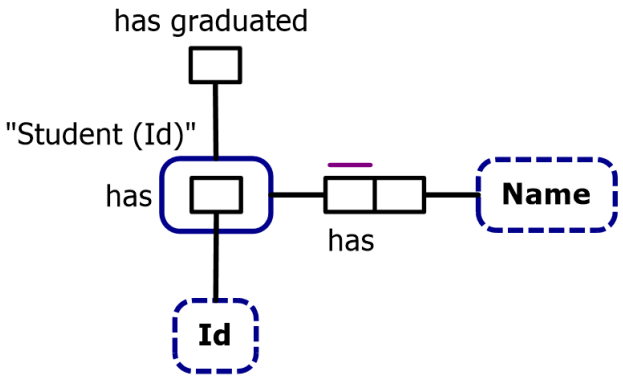
Definition 2.2. The limited inductive definition of ORM.

1. $\emptyset$ is the empty information structure
2. If IS is an information structure, then also the following:
   (a) label type: $IS + Label(N, D, \sigma)$
   (b) fact type: $IS + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)$
   (c) constraint: $IS + Constr(N, C)$
3. Only what can be formed by these information structure composition rules is an information structure.

When we take this inductive definition into practice, we can see that the construction of models is fairly straightforward.

Example 2.1. Inductive schema and corresponding graphical representation

$\emptyset$ + $Label$(Id, Integer, 'student id' Integer)
+ $Label$(Name, String, 'student name' String)
+ $Fact$(Student, {has:Id}, 'student with' Id)
+ $Fact$(Graduated, {has_graduated:Student}, 'has graduated' Student)
+ $Fact$(StudentName, {stud:Student, name:Name}, stud 'has' name)
+ $Constr$(StudentHasOneName, $Unique$({stud}))



Chapter 3

Object-Relational Mapping

In this chapter we will explain the definition and the implications of Object-Relational Mapping (O/RM). We will firstly look at the necessity of O/RM tools. Then we will provide the definition of O/RM and address a specific Object-Relational Mapping tool that uses one of the defined design patterns.

3.1 Motivation

Object-Relational Mapping tools were introduced to cross the divide between the object world and the relational world. As object oriented programming increased in popularity and relational databases remained popular, the demand for tools that can handle both paradigms increased. In this section we will introduce the object model, the relational model and the problems that arise when combining both these paradigms.

3.1.1 The object model

The object model describes systems as built out of objects. These objects are programming abstractions that have identity, state and behaviour [6]. Data and variables are put together in one single concept in these objects. We briefly discuss the three basic building blocks.

Identity Any object has an identity by which it can be distinguished from all the other objects. This identity is not necessarily based on the state of the object, since an object can be different from another object even when its state is equal to the state of the other object.

State The state of an object consists of the values of the attributes of the object. The state of objects can stay the same throughout the life of the object, or go through many state transitions. The state of an

object is only visible by examining the behaviour of the object, because of the concept of encapsulation, which is explained below.

Behaviour and Encapsulation The behaviour of an object is the collection of operations an object provides (the interface), the responses these operations may give, and the changes the operations may cause to the state of the system as a whole. The state of an object can never be directly manipulated, but can only be addressed through the defined behaviour. Objects that interact with other objects can only see the public behaviour of the object, but they can not see how the behaviour and state are implemented. This is called *encapsulation*.

At least the above building blocks have to be implemented for a system or programming language to be object oriented. A few higher level concepts that most, but not all, object oriented systems implement are described below.

Types and Classes A type is the interface specification that the object will support. An object implements a type if it provides the interface described by the type. One approach to implementing objects is to have *classes*. Every class defines an implementation for multiple objects; how to remember state information and which type the object will implement and how to perform the behaviour required for the interface of that type.

Associations When two types are associated, the objects of the one type can be linked to the objects of the other types. This provides some kind of object traversal.

Inheritance The inheritance mechanism specifies that a type may inherit another type. When a type A inherits a type B, all objects of type A can be used just like an object of type B. When applied to classes, inheritance specifies that a class of type A uses the implementation of type B, with possible overriding modifications.

3.1.2 The relational model

The relational model was first introduced in 1970 by Ted Codd [2]. The strong mathematical foundation and its simplicity caused a lot of attention to this paper [4].

The relational model consists of *relations*, *tuples*, *attributes* and *domains*. A database is represented as a collection of *relations*. These relations are more commonly referred to as *tables*.

One of the core concepts of the relational model is the fact that all relations have a flat structure. This means that all elements of a relation are only linear and no hierarchy is present. Each relation consists of multiple

tuples, which describe the data. In the same way that the term *table* is used for *relations*, these *tuples* are more commonly referred to as *rows*. It is important to note that a tuple is identified by its value, and no other identification method is present. Each *tuple* in a *relation* can be described by multiple *attributes*, which is a *column* in the table terminology. Each attribute has an atomic value from a *domain* or has the special value of `NULL`. A *domain* is a set of atomic values, such as the set of all integers. When an attribute of a tuple has the value of `NULL`, it means that the value is unknown or that the value does not exist for that tuple.

Furthermore, constraints can be defined for the relational model. These constraints describe the allowed tuple states, whether an attribute can be `NULL` or not, and the *key* (identifier) of the relation. References between tables can be constructed through *foreign keys*.

For a more in-depth introduction to the relational model, we refer you to the paper by Ted Codd [2] or one of the many books about databases, such as [4].

3.1.3 The object-relational impedance mismatch

The object model and the relational model are both based on very different paradigms. Since the view of the universe of discourse is very different for both models, some incompatibilities are bound to arise when both worlds are combined. These incompatibilities are more often called the object-relational impedance mismatch.

The difficulties can be classified on four levels [10]:

Paradigm These issues are related to incompatibilities between the two different models; the relational model and the object model.

Language These issues are related to incompatibilities between implementations of the models, for example between SQL and Java.

Schema Schema issues are related to the design issues between the implementations of the core concepts in each model. The issues are about the class based and the table based schema design. Note that these issues can be subdued by the ORM to O/RM translation presented in this thesis.

Instance These issues are related to the storage and retrieval of an object in the context of an application that is object-oriented and has a relational database.

A paradigm problem for example, is that there exists some sort of inheritance and hierarchy in the object model, while the relational model only deals with flat relations. Another problem is that a tuple in the relational

model is supposed to be some statement of truth about the universe of discourse. The structure of a class in the object model, however, may be fairly arbitrary.

For a more complete overview of issues that are related to the object-relational impedance mismatch, we refer you to [10].

3.2 Object-Relational Mapping as a solution

In the 1990s, when the object-oriented approach gained traction, efforts were made to deal with the impedance mismatch problem. One of the most promising solutions in that time was the object database. In these object databases the persistence is not handled in relations (as in the relational model), but instead handled by concepts of the object model. Although this seemed very promising, the object database is still only used in small numbers. Some issues are outlined in [11].

Unlike the relative failure of object databases, object-oriented programming is still very widespread. The use of relational databases also continues to stay relevant. Naturally, a different solution to the impedance mismatch was needed. This gave rise to the concept of Object-Relational Mapping (O/RM). In essence, O/RM tools are about the translation mechanism from objects to relational data and backwards. These O/RM tools therefore provide a structured and predefined approach to dealing with the impedance mismatch problem.

3.2.1 Architectural O/RM patterns

O/RM tools all follow a specific implementation, and each one follows a slightly different method in which the domain logic talks to the database. Fowler [5] describes four general approaches in which objects can talk to the database. These four approaches all move beyond simply incorporating SQL in object oriented classes.

Row Data Gateway The first two patterns, the Row Data Gateway pattern and the Table Data Gateway pattern, use the table structure of a database as a base to construct the objects. These objects then form a gateway to the database. In particular, the Row Data Gateway is a gateway where each instance of it represents a row that's returned by a query.

Table Data Gateway The Table Data Gateway pattern creates objects that correspond to a whole table in the database. The data is then returned through a record set that mimics the tabular nature of the database.

Active Record In the Active Record pattern, the Row Data Gateway is extended with methods that provide some more complicated business logic. The objects know how to persist themselves to the database, update themselves and remove themselves from storage. The O/RM tools are generally implemented as a base class with basic behaviour such as persistence. These base classes are then to be extended by the classes that define the domain logic. Examples of tools that make use of the Active Record pattern are: ActiveJDBC for Java, Propel for PHP and ActiveJS for JavaScript.

Data Mapper All above patterns stick tightly to the designed database structure, preventing the refactoring of domain logic into smaller classes, and preventing the use of more sophisticated object oriented methods such as inheritance. According to Fowler, a data mapper handles all of the loading and storing between the domain model and the database, allowing both to vary independently. It is more complicated than the other patterns, but it benefits from the complete isolation of the two layers. Since the data mapper, that sits on top of the objects, knows the mapped structure of the objects and the database, it can even create and update the relational database schema. Different implementations of the Data Mapper pattern can then also be used to use different SQL platforms interchangeably. The Data Mapper pattern is the most advanced pattern to bridge the object-relational divide, and the most interesting one to research. Implementations of the Data Mapper pattern are: Hibernate O/RM for Java, Doctrine2 O/RM for PHP and Bookshelf.js for JavaScript.

3.3 The Doctrine2 O/RM

As described in the previous section, the Doctrine2 O/RM is an implementation of the Data Mapper pattern for the PHP programming language. The Doctrine2 O/RM library is based on the Hibernate O/RM for Java. We choose the Doctrine2 O/RM library as the target of our transformation for a couple of reasons. The first reason is that we simply have the most experience with this library. Secondly, very few research has been done on the PHP language and the language has been the target of a lot of criticism (for example [3]). However, PHP is currently the most popular web language and it is therefore important that at least some research is aimed at this language.

Doctrine2 uses the concept of entities. Entities are PHP objects (classes) that can be identified over many requests by a primary key or another unique identifier [12]. There are not many requirements for the classes, as they do not need to extend any base class or implement an interface. A few minor requirements must be fulfilled. For example, the classes can not

be final or contain final methods. These entities consist of attributes that can be persisted. The data mapping capabilities of Doctrine2 can map the persistable attributes to the database schema and save and retrieve those attributes.

Aside from these entity classes, Doctrine2 defines the attribute and entity mapping information in special metadata. The metadata can be configured in three ways: through 'annotations' that are annotated in comments of the PHP classes, through mapping configurations in YAML files, and finally through mapping configurations in XML files. In this thesis we will provide an example in the XML notation, since it provides a very structured schema. We specifically avoid the annotation mapping to emphasize the distinction between the PHP code and the Object-Relational Mapping. An example of two entity classes with a corresponding mapping can be found in appendix A.

The persistence and retrieval capabilities are provided by the Doctrine EntityManager. This EntityManager can load in entities from the database and persist entities to the database.

3.3.1 Incompatibilities between ORM and Doctrine2

In this section we will discuss some incompatibilities or inconsistencies between ORM and the Doctrine2 O/RM. The first concept of the Doctrine2 O/RM that does not exist in ORM, is the idea of directionality. Associations between entities can be mapped either bidirectionally or unidirectionally in Doctrine2. This makes sense in the object-oriented world, since objects of one class don't need to know that they're being used in the property of another class. In ORM, all associations are bidirectional through their fact types. When we transform an ORM schema into Doctrine2 O/RM mapping, a choice has to made whether to map the relation unidirectionally or bidirectionally. We choose for the least restricting approach; mapping all relations bidirectionally. A developer implementing and using the transformation can then choose to make the association unidirectional whenever this would make sense in her business logic.

Another incompatibility is originated in the ORM side of the transformation. Relations (fact types) can be objectified in Object Role Modeling. This means that fact types can be used as the base for other fact types, making them their own objects. This mechanism does not exist in the object model, as relations cannot implicitly be made their own objects. This problem can be partially solved by providing a sound transformation schema that handles these cases. We will start with the transformation schema in the following chapter, and we will encounter many other interesting incompatibilities.

Chapter 4

Transformation

In this chapter, we make a first step in the formal transformation of Object Role Modeling into Object-Relational Mapping. We briefly explain the approach for the transformation and define the transformation for the most basic elements. After this we will define the actual transformation method.

4.1 Approach

The approach we take to transform Object Role Models into Doctrine2 Object-Relational mapping is heavily based on the inductive definition of ORM described earlier. The resulting transformation mechanism will also be defined inductively. This means that after defining a formal model of Doctrine2, we will first transform the empty schema ($\emptyset$) into an instance of that formal model. We will then be able to transform every possible model, if we are able to transform every additional construction step to the target implementation.

The second step, after defining the transformation rule for the empty schema (definition 2.2, rule 1), will be to map the label construction rule to O/RM (definition 2.2, rule 2a). The fact type construction rule (definition 2.2, rule 2b) will be the target rule for the third step. Since the complexity of the fact type construction rule is high, we will first assume that no objectifications other than the objectifications that define the entity types exist. In the fourth step we will release this assumption and incorporate the objectifications in our transformation. Finally, we will transform total role and uniqueness constraints construction rules (definition 2.2, rule 2c).

4.2 A formalisation of Doctrine2

As described in previous chapters, the Doctrine2 O/RM consists of entity classes and their corresponding mapping files. We will not transform ORM into actual entity classes and their corresponding mapping files, but we will

transform the models into an intermediate format. This format is a formal description from which the entity classes and mapping files can easily be generated. This generation step is out of scope for this thesis, but we will provide a simple example later on.

For our purposes a Doctrine model is a structure

$$\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin \rangle$$

where $\mathcal{A}$ a set of so-called attributes and $\mathcal{E}$ is a set of the previously mentioned entities. Dom is the function that assigns a value type to each attribute and $Values$ is the set of base domains.

Definition 4.1. $Dom : \mathcal{A} \to Values \cup \mathcal{E}$

Furthermore, an entity $e \in \mathcal{E}$ is a structure $\langle N, A, I \rangle$ where N is the name of the entity with attributes $A \subseteq \mathcal{A}$, of which $I \subseteq \mathcal{A}$ are the so-called identifying attributes.

Finally, *bin* is a symmetric relation over $\mathcal{A}$. This relation assigns a 'sister' attribute in another entity, that sister is associated with the relation between the two entities.

Definition 4.2. $bin \subseteq \mathcal{A} \times \mathcal{A}$, where the following properties hold:
$p \text{ bin } q \iff q \text{ bin } p$
$\forall q \in \mathcal{A} : (p \text{ bin } q_1 \wedge p \text{ bin } q_2) \Rightarrow q_1 = q_2$
$Dom(p) = e \Rightarrow \exists \beta \in bin : \beta = p \text{ bin } q \wedge q \text{ bin } p \wedge q \in A \text{ of } e$

4.3 The empty schema

With this definition of a Doctrine2 schema, we can define a mapping function *Dmap* that maps an ORM schema into a Doctrine2 Object-Relational Mapping model.

Definition 4.3. $Dmap : ORM \to O/RM$

With these definitions we can provide the first transformation rule; the rule that transforms the empty schema.

Transformation Rule 1. $\text{Dmap}(\emptyset) = \emptyset = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$

4.4 Label types

The label type construction rule (definition 2.2, rule 2*a*) says that if IS is an information structure, then also $IS + Label(N, D, \sigma)$, where N is an unused name, $D \in Doms$ and $\sigma = \omega_1 D \omega_2$.

Adding a label type to the information schema will only add a domain to the set of domains, since the label type on its own cannot be an attribute that is part of any entity. This gives the following transformation rule:

Transformation Rule 2.

$$Dmap(IS + Label(N, D, \sigma)) = \\ \langle \mathcal{A}, \mathcal{E}, Dom, Values + D, bin \rangle$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin \rangle = Dmap(IS)$

4.5 Fact types

The introduction of fact types is the start for the actual transformation of information schemas to Doctrine2 Object-Relational Mapping. The assumption in this section is that no objectifications except the objectification of unary fact types attached to label types exist. This means that we can assume that every fact type introduced will not be used as the base of other fact types. The first rule consists of the transformation of entity types, which are objectified unary fact types attached to a label type. This transformation rule adds an entity to the set of entities. This entity has the role as an attribute and is identified by the role r.

Transformation Rule 3.

$$Dmap(IS + Fact(N, \{r : X\}, \sigma)) = \\ \langle \mathcal{A} + r, E + \langle N, \{r\}, \{r\} \rangle, Dom + \{r : Doms(X)\}, Values, bin \rangle$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin \rangle = Dmap(IS)$ **and** $X \in Labels$

Another simple transformation is the transformation of a bridge type. A bridge type is a fact type that connects a label type to a non-label type. Even without the information of constraints, we can assume that all labels of bridge types can be absorbed into the entity type. This is especially the case when we can assume that no objectifications exist. This will provide one transformation rule.

Transformation Rule 4.

$$Dmap(IS + Fact(N, \{r_1 : X_1, r_2 : X_2\}, \sigma)) = \\ \langle \mathcal{A} + r_2, \mathcal{E}' + \langle N', A + r_2, I \rangle, Dom + \{r_2 : Doms(X_2)\}, Values, bin \rangle$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin \rangle = Dmap(IS)$
and $X_1 \notin Labels \wedge X_2 \in Labels$
and $\mathcal{E}' = \mathcal{E} \setminus \langle N', A, I \rangle$
and $N' = Name(X_1)$

We have now defined transformation rules for nearly all binary fact types. We will add one more transformation rule for fact types where the base of both roles is an entity type. This transformation rule adds two attributes and two elements of *bin* to the model.

Transformation Rule 5.

$$
\begin{aligned}
&Dmap(IS + Fact(N, \{r_1 : X_1, r_2 : X_2\}, \sigma)) = \\
&\quad \langle \\
&\qquad \mathcal{A} + r_1 + r_2, \\
&\qquad \mathcal{E}' + e_1 + e_2, \\
&\qquad Dom + \{r_1 : e_1\} + \{r_2 : e_2\}, \\
&\qquad Values, \\
&\qquad bin + \{r_1 \text{ bin } r_2\} + \{r_2 \text{ bin } r_1\} \\
&\quad \rangle
\end{aligned}
$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin \rangle = Dmap(IS)$
and $X_1 \notin Labels \wedge X_2 \notin Labels$
and $\mathcal{E}' = \mathcal{E} \setminus \{\langle N_1, A_1, I_1 \rangle, \langle N_2, A_2, I_2 \rangle\}$
and $e_1 = \langle N_1, A_1 + r_2, I_1 \rangle$ **and** $e_2 = \langle N_2, A_2 + r_1, I_2 \rangle$
and $N_1 = Name(X_1)$ **and** $N_2 = Name(X_2)$

If the fact type added to the information schema does not comply to any of the above constraints, we have to make it 'simpler'. We will define a transformation function φ that transforms a difficult fact type construction rule into a simpler construction schema, such that the rules given above can actually handle the newly constructed fact type.

Definition 4.4. $\varphi : (ORM \to ORM) \to (ORM \to ORM)$

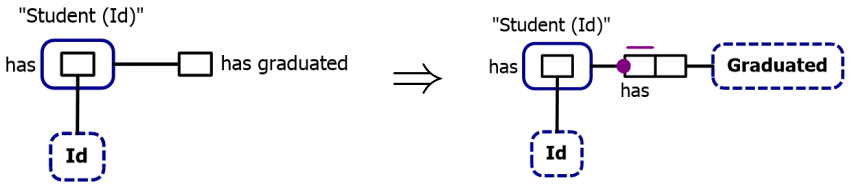
$$
\begin{aligned}
&\varphi(IS + Fact(N, \{r : X\}, \sigma)) \\
&\quad \textbf{if } X \notin Labels : \\
&\qquad IS + Label(N, Boolean, \sigma') + Fact(N', \{r : X, r' : N\}, \sigma'') \\
&\quad \textbf{else} : IS + Fact(N, \{r : X\}, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
&\varphi(IS + Fact(N, R = \{r_i : X_i | 1 \leq i \leq k\}, \sigma)) \\
&\quad \textbf{if } |R| \geq 3 : \\
&\qquad IS + Label(Id, Integer, \sigma') \\
&\qquad\quad + Fact(N, \{r' : Id\}, \sigma'') \\
&\qquad\quad + \Sigma_{i=0}^{k} Fact(N', \{r'_i : N, r_i : X_i\}, \sigma_i) \\
&\quad \textbf{else} : IS + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)
\end{aligned}
$$

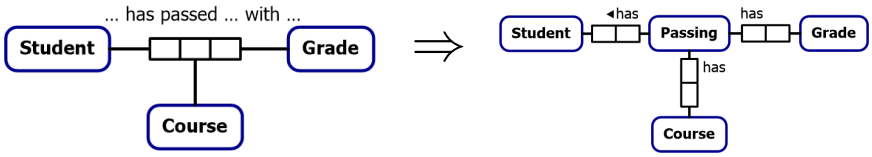
Where σ's, names and roles that are not mentioned in the construction rule are freshly introduced.

This transformation function can handle two different cases:

1. The first case is the case where the unary fact type has an entity type as its base. In this case, this could be represented by a boolean attribute in the entity. This is represented by the following figure:



2. The second case handles any fact type that contain three or more roles. The basic and naïve transformation of this is to make an entity type of that fact type and expand all the object types that are involved in the fact type. For the identification of this newly created entity type, we use an id instead of a composite identification. We do this because Doctrine2 recommends using only singular ids. This is represented by the following figure:



This transformation function φ leads to the final transformation rule for the transformation of fact types.

Transformation Rule 6.

$$\begin{aligned} &Dmap(IS + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)) \\ &\quad = Dmap\big(\varphi(IS + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma))\big) \end{aligned}$$

We will recap the defined rules thus far by giving an extensive example that covers most of the rules given above.

Example 4.1.
Let's suppose we have the information schema displayed in figure 4.1. This is the same information schema as the following schema defined by the inductive ORM construction rules:

$$\begin{aligned} IS = \emptyset &+ Label(Id, Integer, \sigma) && (1) \\ &+ Label(Name, String, \sigma) && (2) \\ &+ Fact(Student, \{sId : Id\}, \sigma) && (3) \end{aligned}$$

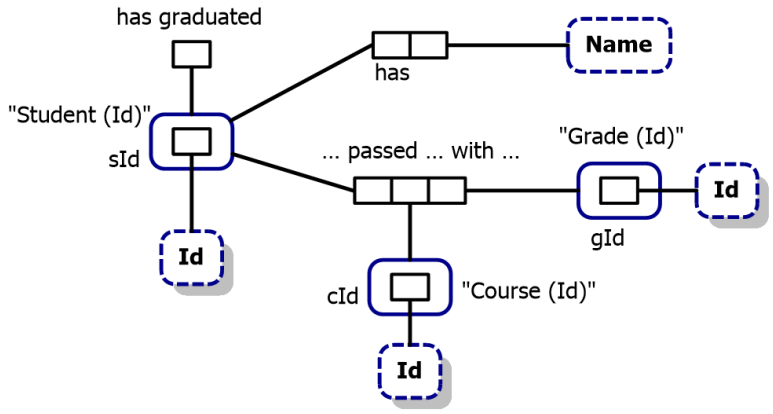


Figure 4.1: Example 4.1

$$+ Fact(Course, \{cId : Id\}, \sigma) \quad (4)$$
$$+ Fact(Grade, \{gId : Id\}, \sigma) \quad (5)$$
$$+ Fact(Graduated, \{has\_graduated : Student\}, \sigma) \quad (6)$$
$$+ Fact(StudentName, \{nm : Name, nmStud : Student\}, \sigma) \quad (7)$$
$$+ Fact(Passing, \{stud : Student, crs : Course, grd : Grade\}, \sigma) \quad (8)$$

We first reduce the information schema to a simpler variant by applying the φ function. Specifically, we apply the first case of φ to (6) and we apply the second case of φ to (8). This provides the following construction schema:

$$IS' = \emptyset + Label(Id, Integer, \sigma) \quad (1)$$
$$+ Label(Name, String, \sigma) \quad (2)$$
$$+ Fact(Student, \{sId : Id\}, \sigma) \quad (3)$$
$$+ Fact(Course, \{cId : Id\}, \sigma) \quad (4)$$
$$+ Fact(Grade, \{gId : Id\}, \sigma) \quad (5)$$
$$+ Label(Graduated, Boolean, \sigma) \quad (6a)$$
$$+ Fact(StudentGraduated, \{gradStud : Student, grad : Graduated\}, \sigma) \quad (6b)$$
$$+ Fact(StudentName, \{nm : Name, nmStud : Student\}, \sigma) \quad (7)$$
$$+ Label(Id, Integer, \sigma) \quad (8a)$$
$$+ Fact(Passing, \{pId : Id\}, \sigma) \quad (8b)$$
$$+ Fact(PassingStudent, \{studP : Passing, stud : Student\}, \sigma) \quad (8c)$$
$$+ Fact(PassingCourse, \{crsP : Passing, crs : Course\}, \sigma) \quad (8d)$$
$$+ Fact(PassingGrade, \{grdP : Passing, grd : Grade\}, \sigma) \quad (8e)$$

When we use the *Dmap* procedure for transforming IS', we get the fol-

lowing result:

$$
\begin{aligned}
Dmap(IS') = \langle\ & \\
& \{sId, cId, gId, grad, nm, pId, studP, crsP, grdP, stud, crs, grd\}, \\
& \{ \\
& \quad \langle Student, \{sId, grad, nm, p\}, \{sId\}\rangle, \\
& \quad \langle Course, \{cId, crsP\}, \{cId\}\rangle, \\
& \quad \langle Grade, \{gId, grdP\}, \{gId\}\rangle, \\
& \quad \langle Passing, \{pId, stud, crs, grd\}, \{pId\}\rangle \\
& \}, \\
& \{ \\
& \quad sId : Integer, cId : Integer, gId : Integer, grad : Boolean, nm : String, \\
& \quad pId : Integer, studP : Passing, crsP : Passing, grdP : Passing, \\
& \quad stud : Student, crs : Course, grd : Grade \\
& \}, \\
& \{Integer, Boolean, String\}, \\
& \{ \\
& \quad studP \text{ bin } stud, stud \text{ bin } studP, \\
& \quad crsP \text{ bin } crs, crs \text{ bin } crsP, \\
& \quad grdP \text{ bin } grd, grd \text{ bin } grdP \\
& \} \\
& \rangle
\end{aligned}
$$

Even though we do not provide the Doctrine2 generation mechanism, we generated PHP entity classes and corresponding Doctrine2 mapping manually for this example. The results of a possible implementation can be found in appendix A.

4.6 Objectifications

In this section we will drop the assumption that no objectifications except for the objectification of unary fact types of label types exist. Any fact type introduced in the schema thus far can be objectified from this point on. The first difficulty with objectifications in the inductive definition of ORM is that objectifications do not exist explicitly. All roles of all fact types have either a label type as its base, or another fact type as its base. Up until now, any role where the base was not a label type was assumed to be an entity type, which is a unary fact type with a label as its base. We cannot rely on this anymore and simple mapping as in section 4.5 will not work anymore. We

solve this by checking whether we need to do anything with the roles and the bases of those roles, before actually applying the rules given in section 4.5.

We therefore define a function ϱ that transforms all fact types if any of their roles have an objectifcation as its base. ϱ will function as an identity function if none of the preconditions match.

Definition 4.5. $\varrho : (ORM \to ORM) \to (ORM \to ORM)$

We will examine all possible fact types that can be objectified and used by a role in another newly introduced fact type in increasing arity.

4.6.1 Unary fact types

Objectifications of unary fact types were partially addressed before. In particular, objectifications of unary fact types with a label type as a base were handled as entity types. These object types were then mapped to an entity in the Doctrine2 mapping. When the base of the single role of the unary fact type is not a label type, it must be another fact type. The first case of the φ function transformed this unary fact type into a boolean attribute.

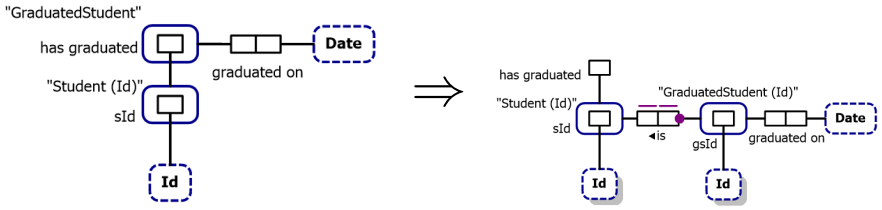
To handle objectifications of unary fact type, a few possible options exist. First, we could add a new entity type and provide a specialization of the newly introduced entity type to the base entity type of the unary fact type. Another possibility is to absorb all fact types of the objectified fact type into the original entity type. The third possibility is to create a new entity type, which represent objects in the objectified entity type, and create a one-to-one relation from this new entity type to the original entity type.

We will not use the first option, since specializations are not supported in the current *Dmap* procedure. It would be possible in an extended version. We will use the third variant, since it provides the best result and makes more sense in the object oriented world.

Definition 4.6.

$$
\begin{aligned}
&\varrho(IS + Fact(N, R = \{r_i : X_i | 1 \leq i \leq k\}, \sigma)) = \\
&\quad IS + Label(Id, Integer, \sigma') \\
&\qquad + Fact(X_j, \{r' : Id\}, \sigma'') \\
&\qquad + Fact(N', \{r'' : X_j, r''' : Base(X_j)\}) \\
&\qquad + Constr(N'', Unique(r'')) + Constr(N''', Unique(r''')) \\
&\qquad + Constr(N'''', Total(r'')) \\
&\qquad + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)
\end{aligned}
$$

Where $\exists x \in R : X_j = x \wedge x \notin Labels \wedge |x| = 1 \wedge Base(x) \notin Labels$
and σ's, names and roles that are not mentioned in the construction rule are freshly introduced.



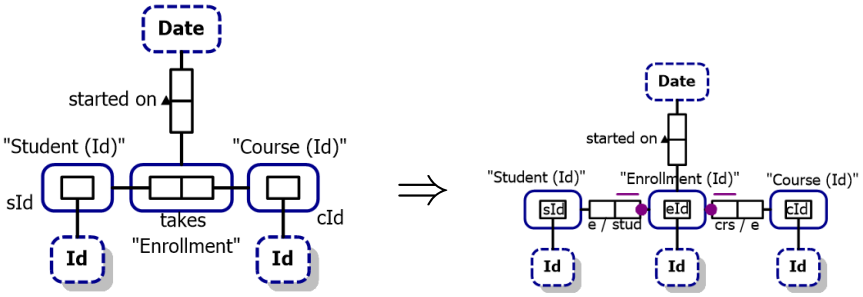
4.6.2 Binary fact types

Objectifications of binary fact types were not yet addressed. Some possibilities could also be considered. The first possibility is simply transforming the binary fact type into its own entity, much like we've done with the higher order fact types. Another possibility is to extract one side of the fact type, and make that it's own entity type. The side to choose, however, depends on the uniqueness constraints, which are not available at this point. We therefore pick the first possibility. The transformation is described as follows:

Definition 4.7.

$$
\begin{aligned}
&\varrho\big(IS + Fact(N', R_1 = \{r'_1 : X'_1, r'_2 : X'_2\}, \sigma') \\
&\qquad + \alpha + Fact(N, R_2 = \{r_i : X_i | 1 \leq i \leq k\}, \sigma)\big) = \\
&\quad IS + Label(Id, Integer, \sigma_2) \\
&\qquad + Fact(N', \{r' : Id\}, \sigma_3) \\
&\qquad + Fact(N_2, \{r'_1 : X'_1, r''_1 : N'\}, \sigma_4) \\
&\qquad + Fact(N_3, \{r'_2 : X'_2, r''_2 : N'\}, \sigma_5) \\
&\qquad + Constr(N_4, Unique(\{r''_1\})) + Constr(N_5, Unique(\{r''_2\})) \\
&\qquad + Constr(N_6, Total(\{r''_1\})) + Constr(N_7, Total(\{r''_2\})) \\
&\qquad + \alpha + Fact(N, \{r_i : X_i | 1 \leq i \leq k\}, \sigma)
\end{aligned}
$$

Where α is a sum of multiple construction rules
and $\forall x \in R_1 : x \notin Labels \wedge |x| > 1$
and $\exists x \in R_2 : x = N'$
and σ's, names and roles that are not mentioned in the construction rule are freshly introduced.



4.6.3 Higher order fact types

If a role is attached to a fact type with three or more roles, we don't need to do anything with the newly introduced fact type. This is due to the fact that we already transformed all fact types with three or more roles to its own entity type using the φ function.

4.7 Constraints

This section describes a transformation method for transforming the constraint construction rule. We will first define a few simple assumptions to limit the scope of the thesis.

1. All constraints added to the schema are either uniqueness constraints or total role constraints.
2. All constraints added span a single fact type.
3. Uniqueness constraints on bridge types are either on both roles, or only on the role that has the entity type as its base.
4. All constraints are constructed after all fact types and all label types have been added.

This fourth assumption is not necessarily true, but for all information schemas the construction steps can be reordered in such a way that the schema is still valid and all constraints are constructed after all fact types and all label types are constructed.

As we observe the results of all previous transformation rules and transformation functions, we can see that all fact types in the transformed schema are binary. This makes our schema very 'simple', and we will use this to reason about the uniqueness and total role constraints.

4.7.1 Total role constraints

The total role constraint is the easier constraint of the two. If a total role constraint is present on a role, then the whole population of the base of

that role must be present in the population of the corresponding fact type. Doctrine2 supports the concept of `NULL`, which is a key feature to map the entity classes to the relational database. A property or relation can be configured to be *nullable*, which means that the value of that property or relation may be `NULL`. To support this concept in our formal mapping definition, we will introduce the set Total, $\mathcal{T} \subseteq \mathcal{A}$, to the Doctrine2 model. This set consists of all attributes that are not allowed to have the value of `NULL`. The generation of Doctrine2 classes and mapping can then make all properties that are not in this set *nullable*.

Transformation Rule 7.

$$Dmap\,(IS + Constr(N, Total(\{r\}))) = \\ \langle \mathcal{A}, \mathcal{E}, Dom, Values, bin, \mathcal{T} + r \rangle$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin, \mathcal{T} \rangle = Dmap(IS)$.

4.7.2 Uniqueness constraints

Uniqueness constraints are more difficult than the total role constraints. The assumptions that all fact types are now binary makes the mapping of uniqueness constraints more straightforward. Each uniqueness constraint added is a statement about a role that is either a bridge type or a normal fact type.

As stated in the assumptions, a uniqueness constraint on a bridge type is either on both roles, or just on the role that is on the side of the entity type. This provides two basic transformation rules:

Transformation Rule 8.

$$Dmap(IS + Constr(N, Unique(\{r\}))) = Dmap(IS)$$

Where $F \in Bridge \wedge r \in F$.

Nothing happens in this first transformation rule. This is because the default case, that was introduced with the mapping of the fact type, was already correct.

Transformation Rule 9.

$$Dmap(IS + Constr(N, Unique(\{r_1, r_2\}))) = \\ \langle \mathcal{A}, \mathcal{E}, Dom - \{r_2\} + \{r_2 : \mathcal{P}(Doms(Base(r_2)))\}, Values, bin, \mathcal{T} \rangle$$

Where $\langle \mathcal{A}, \mathcal{E}, Dom, Values, bin, \mathcal{T} \rangle = Dmap(IS)$
and $F \in Bridge \wedge r_1, r_2 \in F$

All this does is replace the domain of attribute r_2 with the power set of that domain. This means that the value of that attribute is not a singular value, but a set of values from domain $Doms(Base(r_2))$.

The other case is the case where the roles in the uniqueness constraint are not part of a bridge type. In this case, a relation is already present in both entity types that are associated with the fact type. This relation is represented by two elements of the *bin* relation. The semantics of Doctrine2 allow for the specification of different kinds of relations. A Student entity, for example, could have a one-to-many relation with the Course entity, since a Student can be enrolled for more than one Course. The other relation types that exist are the one-to-one relation, the many-to-one relation, and the many-to-many relation. These kind of relationships also exist in ORM, since the uniqueness constraints on a binary fact type have identical semantics. However, our formal model of Doctrine2 does not differentiate between the different kind of relations, as the *bin* relation only provides the associated attribute. Therefore, our final transformation rule does not add anything new to the model. All possibilities of the non-bridge fact type are handled in the final transformation rule:

Transformation Rule 10.

$$Dmap(IS + Constr(N, Unique(R))) = Dmap(IS)$$

Where $F \notin Bridge \wedge \forall r \in R : r \in F$

This final transformation rule concludes our transformation section.

Chapter 5

Conclusions

The old concept of relational databases is still very prevalent, even though the object-oriented programming paradigm has evolved to become the most popular programming paradigm. These two worlds are very different and problems arise when both of those world are combined. The object-relational divide is partly bridged by the concept of Object-Relational Mapping. Object-Relational Mapping does not however, solve the problem of modelling the persistent layer of an application. These models are often constructed in a conceptual modelling language such as Object Role Modeling.

The models however, do not fully take the object model into account. This thesis tried to answer the question:

> How can we formally transform ORM models into Doctrine2 Object-Relational Mapping?

To answer this question, we first explained all related concepts such as ORM, O/RM, the object model and the relational model. With these definitions, we introduced a formalisation of a specific O/RM implementation; Doctrine2.

We provided transformation rules for transforming ORM models into this formalisation following the limited inductive definition of ORM. This means that we first provided a transformation rule for the empty model, then provided a transformation rule for the label type, then provided a set of transformation rules for the fact type, and finally we provided transformation rules for total role and uniqueness constraints.

Limiting the inductive definition of ORM implies that not all models can yet be transformed into Doctrine2 Object-Relational Mapping. The fact that no formal transformation from the intermediate formalisation is given has the consequence that no automatic transformation using a computer program can yet be programmed.

This thesis therefore presented the first steps towards the formal transformation of ORM models into Doctrine2 Object-Relational Mapping by

presenting transformation rules from a limited definition of ORM to a formal definition of Doctrine2 models. Furthermore, we presented some directions and an example of how to transform this intermediate formalisation into an actual instance of Doctrine2 Object-Relational Mapping.

5.1 Future work

The inductive definition of ORM is still very new and untried in practice. More work could be done with this inductive definition of ORM.

Furthermore, the procedure presented in this thesis is fairly basic. Only three of the eight construction rules are transformed. It is obvious that future research could shed some light on the other construction rules. Specialization and generalization in particular are an interesting topic for the O/RM target. If transformed correctly, one could make use of the special properties of the object model such as inheritance.

Another topic that can be addressed is the proving of properties of an O/RM instance that is transformed from ORM. A property that one might think of is whether an ORM model always converges on the same O/RM instance, regardless of the order of the construction rules. The proving of these properties can easily be done by using structural induction on every construction rule and every transformation rule.

Finally, the code generation step, which was not presented in this thesis, can also be constructed. This could be done by providing a program that can transform all ORM models into Doctrine2 Object-Relational Mapping. Functional programming languages such as Clean or Haskell are perfectly suited for this task, given the inductive nature of the transformation rules.

Bibliography

[1] GHWM Bronts, SJ Brouwer, CLJ Martens, and Henderik Alex Proper. A unifying object role modelling theory. *Information Systems*, 20(3):213–235, 1995.

[2] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[3] Eevee. Php: a fractal of bad design, 2012. [Online; accessed 27-May-2015].

[4] RA Elmasri and SB Navathe. *Database systems: models, languages, design, and application programming. 6th Global Education.* Pearson Education (US), New Jersey, 2010.

[5] M Fowler. *Patterns of Enterprise Application Architecture.* 2002.

[6] Mark L Fussel. Foundations of object relational mapping. chimu corporation. 1997.

[7] Terry Halpin. *Information modeling and relational databases.* Morgan Kaufmann, 2001.

[8] Terry Halpin. Orm 2. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, pages 676–687. Springer, 2005.

[9] Terry Halpin. Object-Role Modeling. In *Encyclopedia of Database Systems.* 2009.

[10] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. *Proceedings - 2009 1st International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA 2009*, pages 36–43, 2009.

[11] Neal Leavitt. Whatever Happened to Object-Oriented Databases? *Computer*, 33(8):16–19, 2000.

[12] Doctrine Project Team. Getting started with doctrine - doctrine 2 orm 2 documentation, 2015. [Online; accesses 27-May-2015].

[13] F.P. Tulinayo, Th.P. van der Weide, and P. van Bommel. Using the decomposition mechanism to improve system dynamics conceptualization. 2015. [Work in progress; generated 22-January-2015].

Appendix A

Doctrine2 Example

```php
<?php

class Student
{
    /** @var int **/
    private $id;
    /** @var boolean **/
    private $graduated;
    /** @var string **/
    private $name;
    /** @var Passing[] **/
    private $passings;
}

class Course
{
    /** @var int **/
    private $id;
    /** @var Passing[] **/
    private $passings;
}

class Grade
{
    /** @var int **/
    private $id;
    /** @var Passing[] **/
    private $passings;
}

class Passing
{
    /** @var int **/
    private $id;
    /** @var Student[] **/
    private $students;
    /** @var Course[] **/
    private $courses;
    /** @var Grade[] **/
    private $grades;
}
```

```xml
<doctrine-mapping>
  <entity name="Student">
    <id name="id" type="integer">
      <generator strategy="AUTO" />
    </id>
    <field name="graduated" type="boolean" />
    <field name="name" type="string" />
    <many-to-many field="passings" target-entity="Passing" inversed-by="students" />
  </entity>
  <entity name="Course">
    <id name="id" type="integer">
      <generator strategy="AUTO" />
    </id>
    <many-to-many field="passings" target-entity="Passing" inversed-by="courses" />
  </entity>
  <entity name="Grade">
    <id name="id" type="integer">
      <generator strategy="AUTO" />
    </id>
    <many-to-many field="passings" target-entity="Passing" inversed-by="grades" />
  </entity>
  <entity name="Passing">
    <id name="id" type="integer">
      <generator strategy="AUTO" />
    </id>
    <many-to-many field="students" mapped-by="passings" target-entity="Student" />
    <many-to-many field="courses" mapped-by="passings" target-entity="Course" />
    <many-to-many field="grades" mapped-by="passings" target-entity="Grade" />
  </entity>
</doctrine-mapping>
```