TUHH
Technische Universität Hamburg-Harburg

Conceptual-to-Object Schema Mapping

submitted by
cand.ing Aurel Mihail Ganga

supervised by
Prof. Dr. Sibylle Schupp
M. Sc. Miguel Garcia

Hamburg University of Science and Technology
Software Systems Institute (STS)

Abstract

This thesis aims at mapping syntax from the conceptual language level to the object relational level. For this goal it proposes a mapping of the most popular conceptual modeling language variant called *Object Role Modeling 2* into Object/Relational Mapping. In order to convince data modelers and database practitioners of the benefits of *Conceptual Modeling*, a tool is needed to translate from the Conceptual Level automatically to Object/Relational Schemes.

Because the open-source tools for *Object-Role Modeling* are fragmented, an opportunity arises to metamodel and implement a *Domain Specific Language*of a chosen subset of the specified conceptual language. With this tool, a transition from ORM2 to O/R Mapping can be performed. After this is achieved a solution to bridge the gap between modern DB query languages and the conceptual language is near.

We will discuss the advantages as well as the short-comings of this approach from the moment we design our database on the conceptual level to the translation and the information loss that can occur in the process. We will focus on a subset but give hints on generalizing the problem, with future development and extension if these kind of mapping tools in mind.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 20th July 2009
Aurel Mihail Ganga

I would like to thank Mr. Miguel Garcia for the helpful guidance throughout my thesis work, Prof. Sibylle Schupp for her support of my work, forum users Sebastian Zarnekow and Meinte Boersma for the useful insights into the xText tools and all contributors of useful tutorials and hints about eclipse and xText usage that helped me in writing this thesis.

Contents

List of Figures

Chapter 1

Introduction

Summary. *This chapter is meant to introduce the reader to the universe of conceptual modeling and mapping, giving reasons and insights about the work done. We will be describing the differences between Conceptual Modeling and Object-Relational Modeling.*

1.1 Motivation

The conceptual modeling languages emerged in order to improve the process of designing a database schema. A conceptual model represents concepts (entities) and relationships between them. They are not competing with relational designs, being rather a stage before.

Database systems are build with system efficiency in mind, rather than for human convenience or comprehension. Premature notions like storing records into tables can hinder data modeling. Conceptual modeling does not involve records, tables or attributes. It is designed to be intuitive and fully detached of implementation issues, such as concurrancy or data storage, being on a higher level then relational modeling. The aim of conceptual model is to express the meaning of terms and concepts used by domain experts to discuss the problem, and to find the correct relationships between different concepts. Once the domain concepts have been modeled, the model becomes a stable basis for subsequent development of applications in the domain. The concepts of the conceptual model can be used as basis of object-oriented design and implemented in program code.

1.2 ORM2, ER and UML

The most popular variant of conceptual modeling is Object-Role Modeling, a language that has come to its second edition and is described in Halpin's *Information Modeling and Relational Databases* [HM08]. ORM2 is focusing on fact-orientated modeling and is based on just two construct: objects and relationships. Objects play different roles in their relationships with other objects. While in record-based modeling, there are inter-record and intra-record relationships among attributes possible, in ORM2 all relationships are represented the same way with a single construct.

A conceptual model is composed of a conceptual schema - the design - and a conceptual database - the instances. It focuses on the analysis phase of an information systems development and offers advantages like ease of validation by domain experts and at the same time understanding by business experts that aren't familiar with database management systems, because of its resemblence with the natural language.

ORM2 models are attribute-free and because of this they are inherently more stable than ER models or UML models, where attributes are incapsulated into entities. ORM2 can capture many more rules and constraints than ER or UML. These rules can be expressed naturally through predicates of any arity. In contrast, ER supports only binary associations while UML forbids unary associations and does not support value-based identification schemes.

ORM, the predecessor of ORM2, was born out of the necessity to have a language that is close to the natural language. Since people naturally communicate (to themselves or others) with words, pictures, and examples, the best way to arrive at a clear description of the universe of dicourse is to use natural language, intuitive diagrams, and examples. To simplify the modeling task, ORM2 examines the information in the smallest units possible: one fact at a time. ORM2 aims for simplicity, viewing the world in term of *objects* playing *roles*, as part of relationships.

1.3 Prototype

This student work will be dealing with ORM2 beginning with establishing a metamodel for the language, translating the metamodel into a grammar with an equivalent editor, and designing a prototype to translate an ORM2 schema into a E/R schema. The next chapters will be building up the necessary background information so that in the last chapters the work on the prototype can be summarized.

Chapter 2

Related Work

Summary. *This chapter is meant to give an overview of the software already available for working with ORM.*

2.1 Summary of similar software approaches

There are numerous tools out there to work with ORM2. The development of these open-source tools is fragmented and there is no consensus in the community for a mainstream tool. This is why alot of projects have emerged that look to cover the aspects of conceptual modeling from different perspectives. The lack of a common conceptual language is the reason behind the multitude of tooling practices.

The growth of ORM has followed the availability of a series of steadily improving ORM tools. The early ORM tools such as IAST (Control Data) and RIDL were followed by InfoDesigner, InfoModeler and VisioModeler. When Microsoft bought the Visio Corporation, Microsoft extended VisioModeler and made it a component of Microsoft Visual Studio. This was Microsoft's first ORM implementation and it was published in the 2003 Enterprise Architects release of Visual Studio as a component of the tool called called *Microsoft Visio*for Enterprise Architects.

Some of the most widely used tools for ORM2 are NORMA (Neumont ORM Architect), ActiveFacts, InfoModeler (VisioModeler), VEA, GanttPV and DogmaModeler. Some of these tools have support only the ORM1 standard while some support also the extended ORM2 language.

2.2 Description of the main tools

Neumont ORM Architect (NORMA) for Visual Studio is a free, open source ORM tool released as a community technology preview. This release has a diagram editor that supports the new ORM2 notation to include mapping of the ORM schema to 4 RDBMSs. It is supported and developed by the ORM Foundation [orm09]. The latest release adds verbalization support for default custom property values, improves interpretation of compound names during column name generation, plus other minor features and stabilization over the May 2009 release. Details are available in the included Documentation installed with NORMA. Some of the features of NORMA:

- More compact display of ORM models without compromising clarity
- Improved internationalization (e.g. avoid English language symbols)
- Notation changes acceptable to a short-list of key ORM users
- Simplified drawing rules to facilitate creation of a graphical editor
- Full support of textual annotations (e.g. footnoting of textual rules)
- Extended use of views for selectively displaying/suppressing detail
- Support for new features (e.g. role path delineation, closure aspects)

ActiveFacts is a semantic modeling toolkit that is intended to help the processes of software specification, design, and implementation. It incorporates the Constellation Query Language (CQL) and the Constellation API, which together enable data to be designed, expressed and queried in a completely natural form. The language incorporates natural language expressions into a formal framework. This allows the business user - in conjunction with the programmer and database experts - to use the language to express the rules and behaviour of the business domain, in the process formulating efficient database designs without needing specialist database skills. ActiveFacts is packaged as a Ruby Gem. The latest release can be found online. [act09]

InfoModeler/VisioModeler was one of the first commercially viable/successful ORM tools to hit the database modeling market. InfoModeler was renamed VisioModeler by Visio and now exists as an unsupported software release from Microsoft. Although the database drivers are a bit outdated, with a little extra work one could use this tool to produce ORM models. Most of the functionality found in VisioModeler is now incorporated into the Microsoft Visio for Enterprise Architects product which is shipped with Visual Studio Enterprise Architect Edition.

Visio for Enterprise Architects (VEA) is a Microsoft tool with two releases, in 2003 and in 2005 respectively. Microsoft design a powerful ORM and logical database modeling solution for the product Visual Studio.

Another open source software for project management is the tool **GanttPV** [gan09]. Its creators describe it as a *simple, open-source tool that will help [managers] to manage their projects.* GanttPV allows the scheduling of tasks, task durations, dependencies and start dates. It allows easy task assignment and the identification and prioritizing of the follow-up activities. The tool can be used to monitor the team's productivity and expenses. GanttPV is thus a commercial tool that gained popularity in project management, and allows users to customize it through scripts in the Python programming language.

The last tool that we are looking at is **DogmaModeler**, an Ontology Modeling Tool based on ORM [dog09]. The philosophy of DogmaModeler is to enable non-IT experts to model ontologies with a little or no involvement of an ontology engineer. This challenge is tackled in DogmaModeler through well-defined methodological principles: the double-articulation and the modularization principles. Other features include:

- support for ORM as a graphical notation for ontology modeling

- the verbalization of ORM diagrams into pseudo natural language, that allows non-experts to check, validate, or build ontologies
- the automatic composition of ontology modules; the incorporation of linguistic resources in ontology engineering
- the automatic mapping of ORM diagrams into the DIG description logic interface and reasoning using Racer

2.3 Conclusion

The tools for ORM are fragmented, most of them being focused on only some aspects of the language, and restricting themselves to only a subset. This fact scatters the users and the developers of ORM, each finding the tool that suits best to its needs. This is why new tools are being developed in an effort to improve the usefullness of conceptual modeling notions by making them accesible to a wider group of users.

Chapter 3

Main Concepts of ORM2

Summary. *In this chapter a short description of the main ORM2 concepts will be given, with brief examples. The concepts of fact types, predicates and constraints will be looked upon more closely to determine the elements that we will be using for our prototype.*

3.1 The Basics

In order to be able to map a specific language first there must be a good understanding of the language. The concepts and understanding of ORM2 of this thesis is mostly based on the information from the book *Information Modeling and Relational Databases*[HM08]. This book is about information systems, focusing on information modeling and relational database systems. A major part of this book deals with fact-oriented modeling, a conceptual modeling approach that views the world in terms of simple facts about objects and the roles they play. Fact-orientation is today used worldwide and comes in many flavors, including the Semantics of Business Vocabulary and Business Rules approach adopted in 2007 by the Object Management Group.

An information system may be viewed from four levels: conceptual, logical, physical and external. The conceptual level is the most fundamental, describing our world naturally in human concepts. At this level, the blueprint of the *Universe of Discourse* (referred as UoD from now on) is called the conceptual schema. This describes the structure or grammar of the business domain (e.g., what types of object populate it, what roles these play, and what constraints apply). We will be interested only in this level of view, and see how ORM2 describes it.

The conceptual model is comprised of a conceptual schema - the design based on the metamodel - and a conceptual database - containing the instances. For the conceptual schema, the first building blocks to use are the *facts* (fact instances) - a proposition that we consider to be true. **Fact types** will be the elements of our UoD and include *object types*, *value types* and the relationships between them, called *predicates*. These first fact types are primitive types, as they are considered to be true and valid from the start, as are axioms in mathematics. These facts are found under different names in literature, all meaning the same: asserted, primitive or base facts.

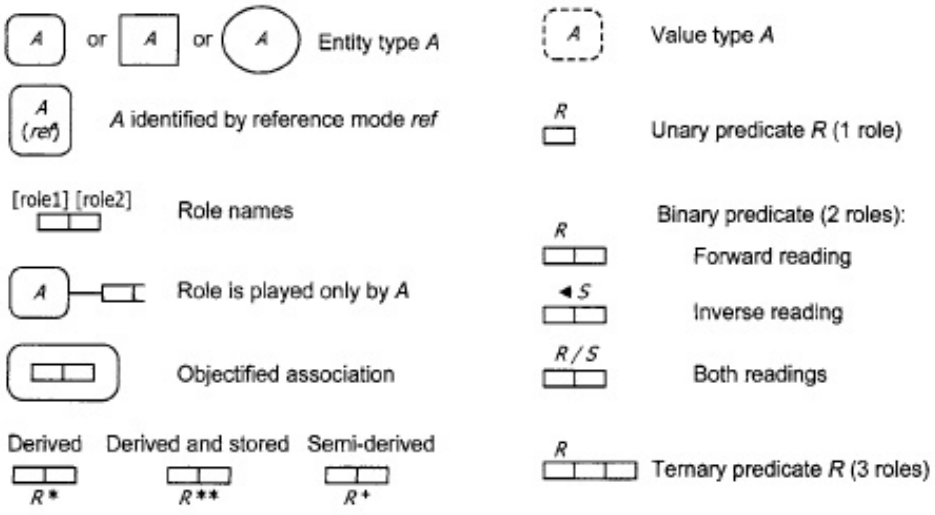


Figure 3.1: Some basic symbols used in conceptual schema diagrams

Derivation rules are statements to derive other facts from already existing facts. Derivation may involve mathematical calculations and logical inference, some of them may not even need documentation, as alot of operators and functions are already defined for most value types. For example, the sum of years spent in school and year spent in college will can be a derivated type without an explicit rule, as the sum operator is the one would expect from a logical point of view.

Constraints are another vital part of the conceptual schema. Also known as *validation rules* or *integrity rules*, they list the constraints or restrictions on instances of the fact types. These may be static or dynamic. *Static constraints* determine what values are allowed for a fact type. *Dynamic constraints* state what transitions between different values are allowed.

Fact types, constraints and derivation rules build up the main section of the conceptual schema. The main symbols often used in ORM2 diagrams are represented in figure 3.1. A full description of the graphical elements used in ORM2 is given in the appendix C.

3.2 Fact types

Fact types are the elements of our UoD. They are either entity types or value types. Entity types are the elements in our universe that we want to model in our database, while value types are the types that we will use to measure our entity types. For example, in a universitary environment, entity types could be student, professor, guest lecturer, lecture room etc. For some entity types we can define value types, for example *kilogram* can be a value type for *weight*. Value types are either established value types like meters, kilograms, degrees, but can be also domain specific values like *person/people* when referring to the entity type *population*. A population is then measured in people,in our example.

An entity type has usually a reference scheme through which one entity is referred to. The usual reference scheme is called injection, where an entity type has a 1:1-relationship with another entity type that is used with the only purpose to refer the initial entity type. For example, a student can have an ID that is used to uniquely identify that student. We can model both *student* and *ID* as entity types with a 1:1-relationship called *identifies*, or just model *student* as an entity type and give it a reference scheme called *ID*. The second modeling option is more elegant, but both are accepted.

A fact instance is any member of the fact type in our model. As an example, if *student* is a fact type, and John is a student, then *John* would be an instance of the fact type *student*.

3.3 Predicates

The predicates are the associations between the different fact types. There can be no predicate without at least one fact type involved in the predicate. The predicates of a UoD connect the different entity types, establishing relationships between them. The entity types are said to play *roles* when referring to a particular predicate they are in. If for example, we have two entity types - *person* and *car* - and the relationship *...has a...*, person would play the role of *owner* of the car, while car would play the role of *owned car*. The role of an entity type is always bound to a relationship in which that entity type is part of.

A distinct property of predicates is the *arity* of a predicate. The arity is the numer of participants that a predicate has. We can have *unary, binary, n-ary* predicates, where n stands for the number of entity types envolved in the relationship.

Some examples (the words in italics are the entity types/instances envolved):

- unary: *John* smokes.
- binary: *Teacher* teaches *student*.
- 4-ary: *Tom* has a *car* with an *engine* with *fuel-ignition*.

Another characteristic of predicates is the *predicate traversal*. A given relationship can be read differently dependent of the order of the entity types envolved in it, but is still considered the same predicate. As an example, the statement "*John* has *a car*." is equivalent to the statement "*A car* is owned by *John*". A predicate can have as many traversals as permutations of the entity types allow it.

3.4 Constraints

Constraints are essential to any good conceptual design. They assure that constraints and restrictions on the instances of the fact types are kept and that the integrity of the database is not compromised. There is a large variety of constraints, both static and dynamic ones. The focus is usually on the static constraints, that determine what value is allowed for an entity type.

Uniqueness constraints aim to eliminate redundancy. They make sure that the instances of an entity type or the combination of certain roles of a predicate is unique. These constraints are useful to remove duplicates after projections or joins of tables, for example.

Mandatory role constraints ensure that each member of an entity type is part in a relationship. For example, if we have a mandatory role constraint on the entity type *person* to play the role of *offspring* in a relationship *...is child of...*, that would mean that each person must be child of somebody else.

A subtype constraint is a constraint that limits the subtype from its supertype after a certain subtyping rule.

*O*ther common constraints are *value, subset, equality, exclusion, comparison constraints*, constraints that mainly refer to the values an entity type can have. Less used constraints include occurrence frequencies, ring constraints, object cardinality constraints, role cardinality constraints, value-comparison constraints. To be noted it that some constraints operate on entity types while others operate on predicates or arities. For a detailed explanation of these constraints refer to Halpin's descriptions of these constraints in chapter 6 and 7. [HM08]

3.5 Chosing a subset

As we can see the ORM2 conceptual language is vast and powerful, while also very flexible. For the purpose of this student paper we will chose to focus on a subset of the language to generate the prototype and show the implications of such an approach. The chosen subset will be described later on in the chapter 6.1.

Chapter 4

Tools Used

Summary. *The prototyping has been done in Eclipse SDK Galileo 3.5 and Ganymede 3.4 using the plugins for EMF2.4.2, xText 0.7.0 TMF and oAW 4.3.1, EMFT ecoretools 0.8, emfatic 0.3.0 and their dependencies. In the following chapter the IDE will be briefly described as well as the plugins and their respective features.*

4.1 Eclipse

Eclipse is a software development platform comprising an IDE and a plug-in system to extend it. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others. In its default form it is meant for Java developers, consisting of the Java Development Tools. Users can extend its capabilities by installing plug-ins written for the Eclipse software framework. This plug-in mechanism is a lightweight software componentry framework. In addition to allowing Eclipse to be extended using other programming languages, the plug-in architecture supports writing any desired extension to the environment. This is of course good news for the purpose of this thesis as the prototype is written using these capabilities. The IDE makes use of a workspace - a filespace used by the environment coupled with metadata - and can easily switch between multiple workspaces. Also multiple instances of the Eclipse runtime are allowed at anytime. This flexibility and ease of extension through plugins makes it very popular among developers.

The latest releases are *Ganymede* (version 3.4) and *Galileo* (version 3.5). Because some of the plugins aren't supported by the latest version, some development has been done in the Ganymede release. Each version comes with a different build of xText. These differences are important when deciding about the features that the plugins must support. For example, the plugins for EMFT ecoretools and for emfatic are compatible only with the Ganymede release, while the newest version of xText, TMF release, is only compatible with the Galileo release.

Useful reading for getting started with Eclipse are books like Thomas Künneth's introductory book for Eclipse Ganymede [Kue08]. For solving code-problems one might find Berthold Daum's codebook [Dau06] very useful. The online documentation

under the form of a wikipedia [ecl] is providing documentation to Eclipse and its plugins.

4.2 xText

Xtext is a framework/tool for development of external textual DSLs. A very own DSL can be described using Xtext's simple grammar language and the generator will create a parser, an AST-meta model (implemented in EMF) as well as a full-featured Eclipse Text Editor. The Framework integrates with technology from Eclipse Modeling such as EMF, GMF, M2T and parts of EMFT. Adding new features to an existing DSL is intuitive and with the new TMF version more sophisticated programming languages can be implemented.

A domain-specific language (DSL) is a small programming language, which focuses on a particular domain. Such a domain can be more or less anything. The idea is that its concepts and notation is as close as possible to what you have in mind when you think about a solution in that domain. Of course we are talking about problems which can be solved or processed by computers somehow. There are a couple of well-known examples of DSLs. For instance SQL is actually a DSL which focuses on querying relational databases. Other DSLs are regular expressions or even languages provided by tools like MathLab. Also most XML languages are actually domain-specific languages. The whole purpose of XML is to allow for easy creation of new languages. Unfortunately with XML you are not able to change the concrete syntax, which is the major problem with it. The concrete syntax of XML is way too verbose. Also a generic syntax for everything is a compromise. Xtext is a sophisticated framework that helps to implement your very own DSL with appropriate IDE support. There is no such limitation as with XML, you are free to define your concrete syntax as you like. It may be as concise and suggestive as possible being a best match for your particular domain. The hard task of reading your model, working with it and writing it back to your syntax is greatly simplified by Xtext.

The two existing versions of xText are the openArchitectureWare version [oAW09] and and the new TMF version [tmf09]. After working with both releases, some of the advantages of the TMF release are:

- newsgroup support
- imports of models are supported
- overall improvements and optimizations

The openArchitectureWare version is not developed anymore. There is a forum for discussion, but for new development it is recommended to use the TMF version. Tutorials and getting started material is still mostly based on the openArchitectureWare release. The grammar is mostly the same between the versions, and most of the code has been tested to work under both releases. For easier reference the code examples will refer from now on to the TMF version if not stated otherwise.

To get started with xText one must first check that xText is correctly installed in its Eclipse environment as well as all dependencies. After that, one can create a new xText project, which includes a folder to define the grammar and configure the runtime aspects of the language, and a second folder with the generator for the

DSL, and a third folder containing the user interface aspects like editor, outline view, code completion (See picture 4.1). The hierarchy can be viewed with the help of the *Package Explorer* on the left side. The user-defined code and the compiled code are strictly separated into different folders to avoid confusion. The user should write code only in the folder named *src*, while the folder *src-gen* is reserved for the compiled code.

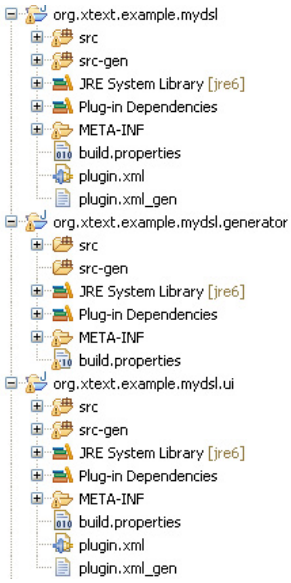


Figure 4.1: Package Explorer of an xText Project

The grammar itself is is made by the principle "*easy to learn but hard to master*" and is documented on the developer's homepage [xte09]. Note that you can import EPackages, a feature new to TMF and not available in the oAW release. The grammar is written in a file with the xtext extension. The chk-file is the file that contains the checks that should be performed in the DSL editor. The mwe-file is the workflow-file, a file that configures the workflow of the editor.

After the xtext grammar is written, the workflow configured and the appropriate checks written, one must run the workflow. After running the workflow, one must export all files included in the project and restart the application. After restart, one can select to create a new project of the new DSL, and write a program according to the grammar rules of the DSL. The editor will have syntax highlighting, real-time error warnings as well as an outline editor for the current grammar.

By editing the grammar and then using the DSL editor to see the changes one can see if the grammar behaves as expected as well as test different scenarios with different grammars for a possible best solution. xText offers a good universal tool for DSL developers that is easy to use and offers a testing ground for developers of new DSLs.

4.3 EMTF Ecore Tools

The EMTF Ecore Tools were compatible only with the Ganymede release of Eclipse at the time of development of the prototype. Thus the development and the visualization of the metamodel for ORM2 has been done with the help of the Ganymede release.

The Ecore Tools component provides a complete environment to create, edit and maintain Ecore models. This component eases handling of Ecore models with a Graphical Ecore Editor and bridges to other existing Ecore tools. The *Graphical Ecore Editor* implements multi-diagram support, a custom tabbed properties view, validation feedbacks, refactoring capabilities among other. The main files used are the .ecore-file and the .ecore_diagram-file, which both work in dependency. If one updates one of the files, the other one is updated as well and might have errors if updated poorly. This is why editing is done mainly in the ecore_diagram-file through the graphical editor, when using this tool. The ecore-model is graphically modeled using the following:

- EClass (for class)
- EPackage (for packages)
- EAnnotation (for Annotations)
- EDataType (for DataTypes)
- EEnum (for Enumerations)
- EAttribute (for Attributes)
- EOperation (for Operations)
- Association (creates Association link)
- Aggregation (creates Aggregation link)
- Generalization (created Generalization link)

4.4 Emfatic

Emfatic is a text editor supporting navigation, editing, and conversion of Ecore models, using a compact and human-readable syntax similar to Java. The best way to gain hands-on experience with Emfatic is to right-click on any .ecore file and choose Generate Emfatic source, a similar converter works in the opposite direction. The Outline view displays the same elements as the Sample Ecore Editor, toolbar actions are available for hiding/showing annotations, attributes, references, operations. Mark Occurrences highlights usages of the same EClassifier, range indication on the vertical bar spans the EClassifier declaration. Folding is supported, with an annotation hover for collapsed regions.

This tools was useful when navigating through the metamodel and when trying to convert the metamodel into the xText grammar.

4.5 Development Environment

The development environment while using Eclipse is easy to install, portable and highly adjustable to one's needs. Alot of aspects of Eclipse and its plugins are customizable as well as solid and reliable. Even a person not familiar to the platform will find it easy to work with Eclipse and its plugins.

Chapter 5

Metamodel

Summary. *This chapter describes the metamodeling stage of the prototyping process. It introduces the metamodel used and follows the work done to get to the final metamodel.*

5.1 ORM2 Metamodeling

In order to create a grammar tool, a metamodel for the ORM2 language needed to be established. The inspiration for the metamodel was the descrition in chapter II of the paper "Information Modeling Methods and Methodologies" [JKS05]. The metamodel A described in this chapter was chosen to be the backbone of the subset for the prototype. This chapter provides two models - named A and B - and specifies the grammar of syntactically valid ORM models. It is noted that a tool that would support editing of ORM models should allow storage of in-progress ORM models that violate rules, while being capable of checking compliance with these rules when a model error check is requested.

For metamodel A the following descriptions are made: An *object type* is either an *entity type* - displayed as a named ellipse - or a *value type* - a dotted ellipse. If an entity type has a simple reference scheme, this may be abbreviated by a *reference mode*. A *role* is depicted as a box and is always part of a relationship type. The *arity* of such an association is the number of roles it has. Thus we can have unary, binary, tertiary etc. relationships, that are composed of a logical *predicate* with open placeholders for objects, and the respective object types that play the roles in the relationship. These can have different readings, depending on the order of traversal. Arrow-tipped bars over roles indicate *internal uniqueness constraints*. A black dot on a role connector depicts a *mandatory constraint* and a circled black dot stands for a disjunctive-mandatory (inclusive-or) constraint. There can be also specified *value constraints*.

Other constraints modeled in this metamodel are *set-comparison constraints* that may apply between compatible role-sequences - *subset* (depicted as a cicled inclusion), *equality* (depicted as a circled =), and *exclusion* (depicted as a cicled x). *Subtyping relationships* are depicted using solid arrows from subtype to the supertype. Subtypes are usually declared through a subtype definition. An object can be a primitive or a subtype. Subtyping allows multiple inheritance. Subtyping is complemented by

completeness or exclusion constraints. A predicate is either primitive or derived. Subtypes and derived predicates are usually defined by rules.

In ORM2 an association can be *objectified*. In our model, the fact type *ObjectType* is a subtype of *ObjectType* is objectified as the entity type SubtypeConnection. This nesting is done in this metamodel as a 1:1 association where EntityType objectifies Predicate. This metamodel makes no reuse of elements in the UML metamodel, its main purpose being to clarify the semantics first using ORM2.

The metamodel started out by trying to model the three main figures desribed for the metamodel A of the paper, the **main types in ORM2** , the **naming of ORM2 roles, predicates and associations** and the **metamodel constraints** (See Cuyler&Halpin's Paper [JKS05]). This resulted in three ecore-files with their distinct diagrams and emf-file. The main part of the metamodel modeling the types was modeled in first in ecore with the help of the ecore modeling tools.

5.2 MainTypes Model

The starting diagram was figure 1B.1 of the mentioned paper, which models the metamodel A in respect to the main concepts. The main types of ORM2 are modeled as *EClasses*: ObjectType, Role, SubtypeConnection, ObjectTypeKind, SubTyping_Constraint, Primitive_ObjectType, Subtype, Value_Type, Entity_Type, Predicate, Subtype_Definition, DataType, Unnested_EntityType, Nested_EntityType, RefModeName, RefModeType, Derivation_Rule. The subtypes are modeled through *Generalization* arrows. The predicates are done through *Associations* with the respective arities and uniqueness constraints attached. Nesting is done by linking the predicate with a respective object type through a *note*. Injection reference schemes are *EAttributes*. Constraints like *acyclic* and *intransitive*, equality constraint or the *XOR* connector are also done through notes. Derivation rules and Subtype definition were left out in this graphical representation of the metamodel, as the only posibility to integrate would be to list them in a note under the model. The result 5.1 was the metamodel **MainTypes**. A graphical view is enabled by opening the file *maintypes.ecore_diagram*, while the hierarchy is saved under *maintypes.ecore*.

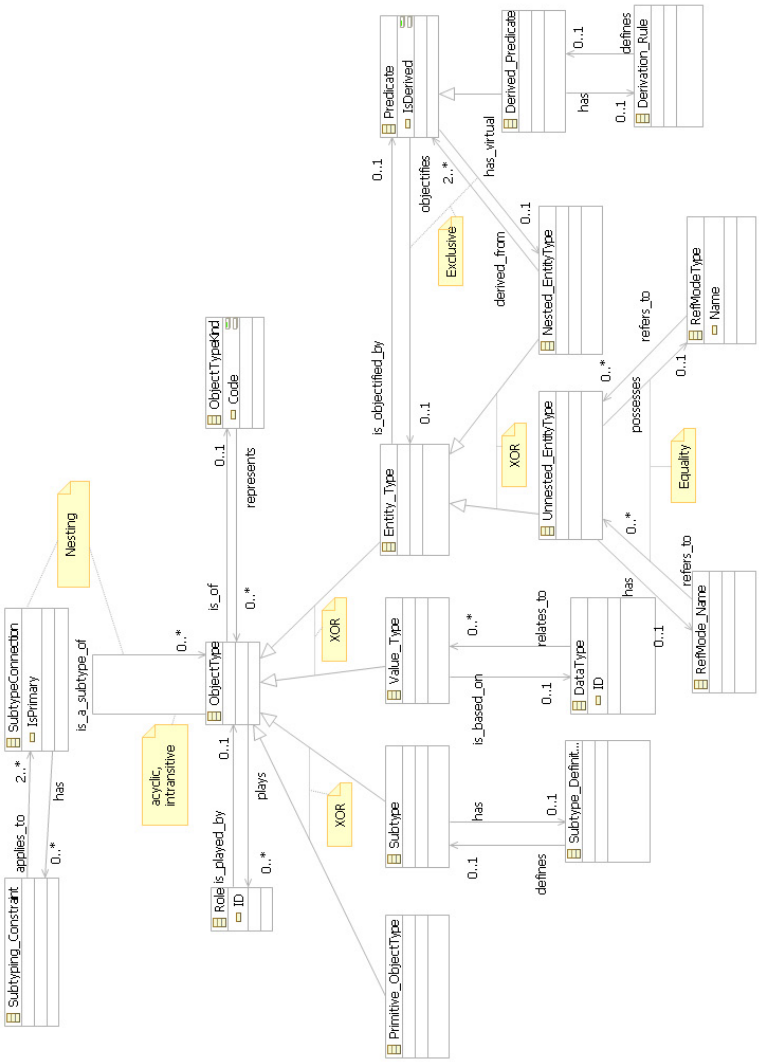


Figure 5.1: Main Types in ORM2 modeled in ETMF

5.3 Naming Model

Another aspect of ORM2 metamodeling is the naming of ORM2 roles, predicates and associations. This was modeled after figure 3B.2 of the discussed paper. Here we have the *predicate* modeled, in relationship with its arity, predicate traversal, roles and association readings. Here the *predicate* denotes an unordered set of roles. The traversal is the ordering (permutation) of that roles, to correspond to a predicate in the logical sense. A simpler model can be obtained by restricting the readings, a property that I will use later on.

In the resulting ecore metamodel, the *EClasses* are: RoleName, Role, Predicate, ObjectType, Predicate_Traversal, Arity, Model, Association_Reading, Position and Predicate_Reading. *EAttributes* are used for the injection reference schemes. Because Ecore models only 1:1 associations, we use an *ENote* for the predicate with an arity of three used in the original metamodel. The usual inclusion and equality constraints are modeled through ENotes. Derivation Rules and Textual Constraints are left out, as they can be only commentary outside the actual graphical model. The result 5.2 was the metamodel **Naming**. A graphical view is enabled by opening the file *naming.ecore_diagram*, while the hierarchy is saved under *naming.ecore*.

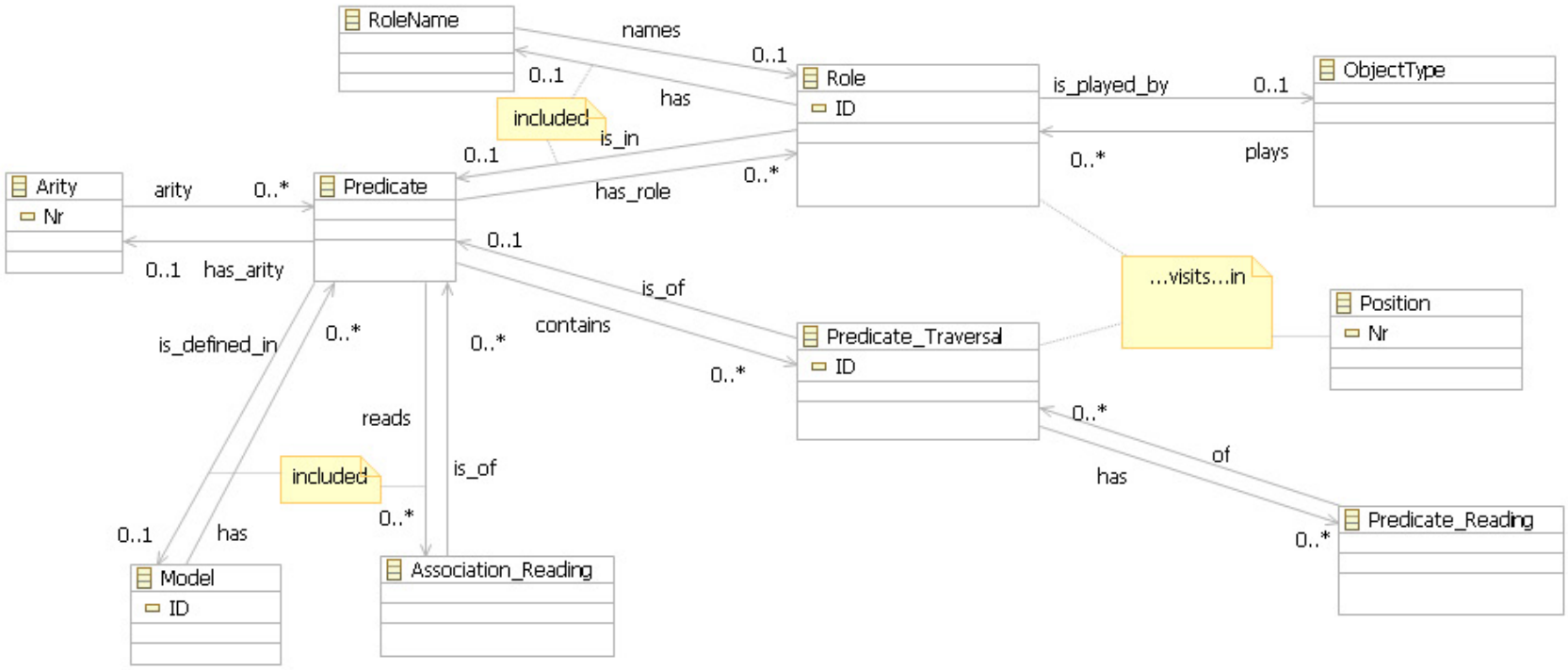


Figure 5.2: Naming of roles, predicates, associations in ORM2 modeled in ETMF

5.4 Constraints Model

Probably the most difficult part of the metamodeling arises when constraints come into play. First one has to limit them and decide which constraints to use, as the ORM2 language allows for a multitude of constraints that are both flexible in use and hard to generalize. The metamodel for the constraints was inspired by figure 8B.3. All constraints derive from the entity type *Constraint*. Here are listed the following constraints as entity types: Subtyping Constraint, MandatoryOrRingConstraint, ObjectType ValueConstraint, UniquenessOrFrequencyConstraint, SetComparison Constraint, Textual Constraint. The constraints are modeled with respect to their relation to other entitities and among themselves. The ecore model contains this representation, using as usual *EClasses* for the mentioned entity types, *Generalization* arrows for the subtyping and *Association* arrows for the predicates. The result 5.3 was the metamodel **constraints**. A graphical view is enabled by opening the file *constraints.ecore_diagram*, while the hierarchy is saved under *constraints.ecore*.

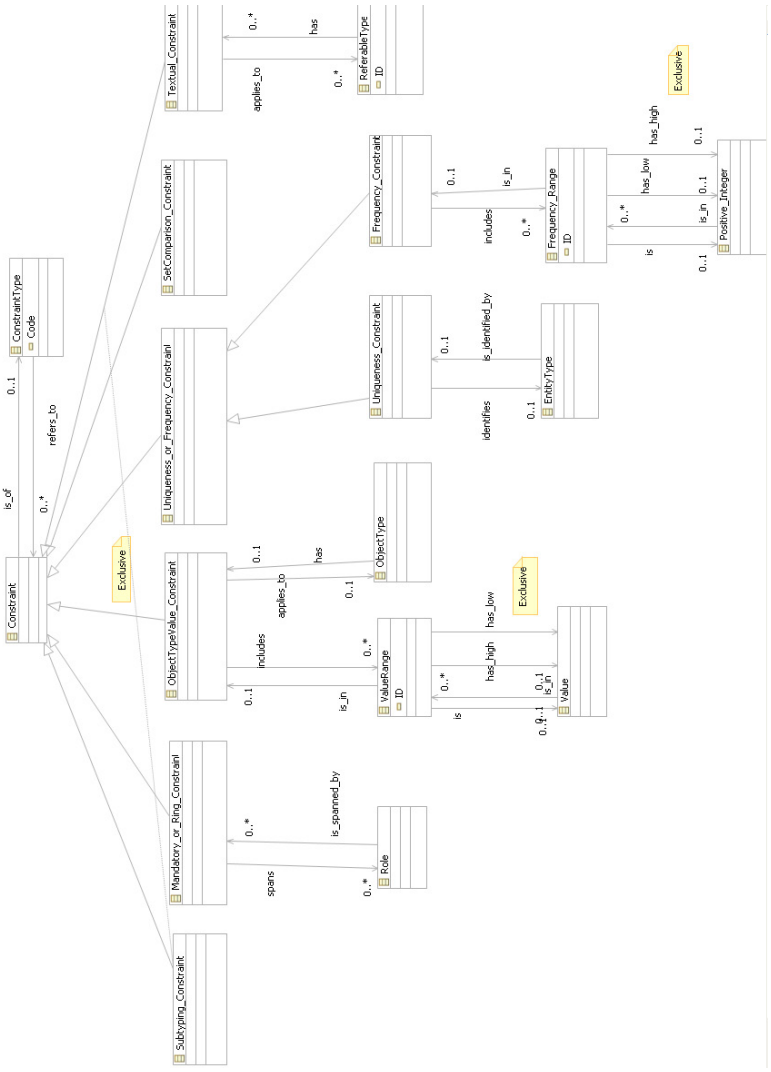


Figure 5.3: Contraints in ORM2 modeled in ETMF

5.5 From Metamodel to Grammar

The first step of applying this metamodel was to see the compatibility with eText and the possibilities of import. After experimenting with the code and learning about the different possibilities in xText, a different hierarchy developed, which integrated some of the parts of the metamodel and left other part due to xText's limitations or due to limiting the subset of ORM2 for this tool. Thus this metamodel was rather a starting point from which the code development for the tool began. This will be discussed in more detail in the chapter *Prototype* 6.1.

Chapter 6

Prototype Implementation

Summary. *This chapter summarizes the work designing the xText grammar as well as the prototype to translate ORM2 schemes into E/R counterparts.*

6.1 The xText Grammar

In order to integrate the ecore metamodel into the xText editor, xText TMF offers the possibility to import existing ecore models. This possibility was explored but proved to be not practical for our ORM2 grammar. For this reason, the metamodel was used as inspiration, but a completely new grammar was written, without importing the existing ecore models from the previous chapter. The grammar was developed starting with the help of online tutorials and the online newsgroup for xText TMF.

In order to write a grammar a subset of the language must be established, to underline what aspects of ORM2 will be supported and which will be left out. This version of the prototype aims to support closed-world types of UoDs, that model static world assumptions.

Aspects that are covered:

- Conceptual Level
- Asserted (Primitive) Fact Types
- Object Types, Entity Types, Value Types
- Nesting
- Definitions of Object Types
- Subtypes
- Predicates
- Roles
- Arity

- Static Constraints: Uniqueness, KeyLengthCheck, InclusiveOR, ExclusiveOr, Enumeration, Range, RoleValueConstraint, Subset, Equality, Exclusive
- Derivation Rules (Basic Level)

Aspects left out:

- Logical, Physical, External Level
- Arity checks
- Sample populations, instances
- Conceptual joins, External uniquness constraints
- Projections
- Reference schemes (only simple injection supported)
- Dynamic Constraints
- Occurrence Frequencies, Ring Constraints, Value-Comparison Constraint, type-Cardinality-Constraint, Textual Constraint
- Checks for redundancy, consistency, completeness.
- Different traversals of same predicate

The resulting xText grammar has a tree-like structure, where the *model* is a list of *elements*. An element is either an *object_type*, a *predicate*, a *constraint*, a *derivation_rule*, a *scheme* or an *instance*. An *object_type* is either an *entity_type* or a *value_type*. An *entity_type* can be a subtype, can be nested and thus linked to a *predicate*, can have a definition and a subconstraint definition -in case it is a subtype. A *value_type* can be a predefined data_type or user-defined. Predefined data_types are STRING, INT or user-defined. A *predicate* is defined as having a list of *participants*, a possibility to specify the arity and a possibly derived. A *participant* refers to an object_type having a role. The role can be specified as mandatory. The simplest reference scheme is modeled, the *injection*, a 1:1 association where an *object_type* is referenced by an *entity_type*. Furthermore following constraints have been modeled: *the uniqueness constraint, the key-length-check constraint, inclusive-OR constraint, exclusive-or constraint, enumeration-constraint, range-constraint, role-value-constraint, subset-constraint, equality and exclusion constraint.* The possibility to write derivation rules as text is given through *derivation_rule*, but due to high complexity the possibility to evaluate the rules is not given at the moment. *Instances* can be creates for any *object_type*; these are yet not given any restrictions, limitations or usage.

The result is a grammar that is conform with the subset of ORM2 chosen for our prototype. For further usage one can create models through it or use its classes in further programs.

6.2 Editor Design

From the designer perspective, the design and implemetation of the tool was both a challenge and an exercise of software design. Starting from examples and tutorials I needed to adapt my subset and model to the existing grammar given in the documentation. Also, some aspects were new and undocumented, and needed to be solved by having contact with other members of the online community through the official forums. There were many decisions I had to take that could go both ways with possible pros and cons, some of which were taking when restricting the subset used or when deciding not to import the ecore models already existing, due to more problems showing up then actually solving. As a designer, one must also think of a reasonable grammar with commands that are intuitive, easy to understand and remember, while at the same time not restricting the user in his choice of model too much. The grammar should be easily expandable, as it is rather a proof of concept and a first step towards a tool that could be someday used on a larger scale.

As a user of the external xText editor, the modeler should have the ORM2 model that he wishes to model. The input of the graphical ORM2 model is designed to be intuitive, so that any person familiar with ORM2 after seeing the documentation can easily understand and remember the main commands. The concept of the modeling in the editor is taking a linear approach and entering the main concepts in an arbitrary order, for example by entering the *entity_types* first, followed by the *predicates* and the *constraints*. All other aspects can be filled at any time, with real-time syntax highlighting and error messages that help to fasted a correct input of the model.

6.3 Editor Usage

The editor is designed to give modelers in ORM2 a tool to implement their ORM2 model. Its first version is meant to find a compromise between the complexity of ORM2 concepts and the vastness of the language on one hand and the user-friendliness and easy access to the prototype without extensive learning of the documentation on the other hand. It is also meant to be a proof of concept which can be later expanded.

1. Drafting a Model

   The first step is having a case study or model that needs to be modeled in ORM2. Recommended: Apply the CSDP steps explained by Halping in the chapters 3-7 [HM08]. You should have the graphical representation in ORM2 notation of the model before using the editor for easier input.

   The model consists mainly of object types and the relationships between them. These should be modeled first through the respective keywords. The constraints can be added similar through the respective keywords, with references to the object types and/or relationships they constrain.

2. Keywords, Structures:

   Keywords are written in **bold**, custom names in *italic*. [ ] means optional. Keyword string means a string is expected. Keyword int means an integer is expected.

Object type: **object** *object_name* [**subtypes** *object_name*]

Entity type: **entity** *entity_name* [**subtypes** *object_name*] { [**nests** *predicate_name*] [**definition** string] [**subconstr** string] }

Value type: **value** *value_name* [**subtypes** *object_name*]

Data type (reference to an existing value type or a string or an int): **data** *value_name*

Predicate: **predicate** *predicate_name* { **part** *object_name1* **role** *role1*, *object_name2* **role** *role2*, *object_name3* **role** *role3*,, [**UC** *uniqueness_constraint1*, *uniqueness_constraint2*, *uniqueness_constraint3*, ,] [**arity** *int*] [**derived**] }

Constraints - *Note! Constraints' keywords all end with the letter C.*

Uniqueness Constraint: **uc** *uc_name* { *predicate_name rolea roleb rolec*, *roleb rolec*, *rolea rolec*,, }

KeyLengthCheck-Constraint: **klc** *klc_name* { *predicate_name* }

Inclusive-OR: **iorc** *iorc_name* { *role1 role3 role5* }

Exclusive-OR: **xorc** *xorc_name* { *role1 role3 role5* }

Enumeration_Constraint: **enumc** *enumc_name* { *value_name* (*instance_of_value1 instance_of_value2 instance_of_value3*) }

Range-Constraint: **rangec** *rangec_name* { *value_name* (*instance_of_value_low .. instance_of_value_high*) }

RoleValueConstraint: **rvc** *rvc_name* { *value_name* (**role** *instance_of_value1 instance_of_value2 instance_of_value3*) }

Subset-Constraint: **ssc** *ssc_name* { *subset_role_name superset_role_name* }

Equality-Constraint: **eqc** *eqc_name* { *role1 role2* }

Exclusion-Constraint: **xc** *xc_name* { *role1*, *role2* }

Derivation Rule: **derivationRule** *dr_name* string

The full code of the xText grammar can be found at the end of this paper in Appendix A.
With this grammar, one can create a ORM2 model with the chosen extension, in our

example called *.mydsl*. The model can be then exported and used for a creation of a Java-based translating tool that can translate the ORM2 notions into E/R counterparts. For a tree-like representation of the model, one can open the file in the Sample Reflective Ecore Model Editor, and have an Ecore-like representation of the model.

6.4 Using the Models - Visitor Pattern

The motivation is to create a visitor for the model that can navigate through the tree and perform tasks. The visitor design pattern is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch. [vis09]

The idea is to use a structure of element classes, each of which has an accept() method that takes a visitor object as an argument. Visitor is an interface that has a visit() method for each element class. The accept() method of an element class calls back the visit() method for its class. Separate concrete visitor classes can then be written that perform some particular operations, by implementing these operations in their respective visit() methods.

With a visitor, one can use the classes created by the xText Editor, in order to translate them into E/R counterparts. The initial attempt of me to create a visitor for the xText generated model was not successful, due to missing documentation on behalf of xText file usage and export.

6.5 Translating into E/R counterparts

Translating into ER counterparts can be done by translating ORM2 for input into Microsoft's Entity Framework, into a Java-based application or into a JPA schema. In turn, a JPA schema can be represented as an instance of the metamodel depicted in Fig. 4 of the paper *Formalizing the well-formedness rules of EJB3QL in UML + OCL* by *Miguel Garcia* (For referance see [Gar06]). The figure 6.1

My initial efforts to finish the translator in the time allocated fot this student thesis has been unfruitful due to not finding documentation on how to export and use the model provided by xText. This can be done in the near future, when documantation, tutorials and similar work for xText TMF will most likely be available.

6.6 Future Work

The next steps would be to use the grammar editor for a translator designed to automate the transition from ORM2 model into ER counterparts. This should be done by implementing a visitor or by using the files generated by the xText editor.

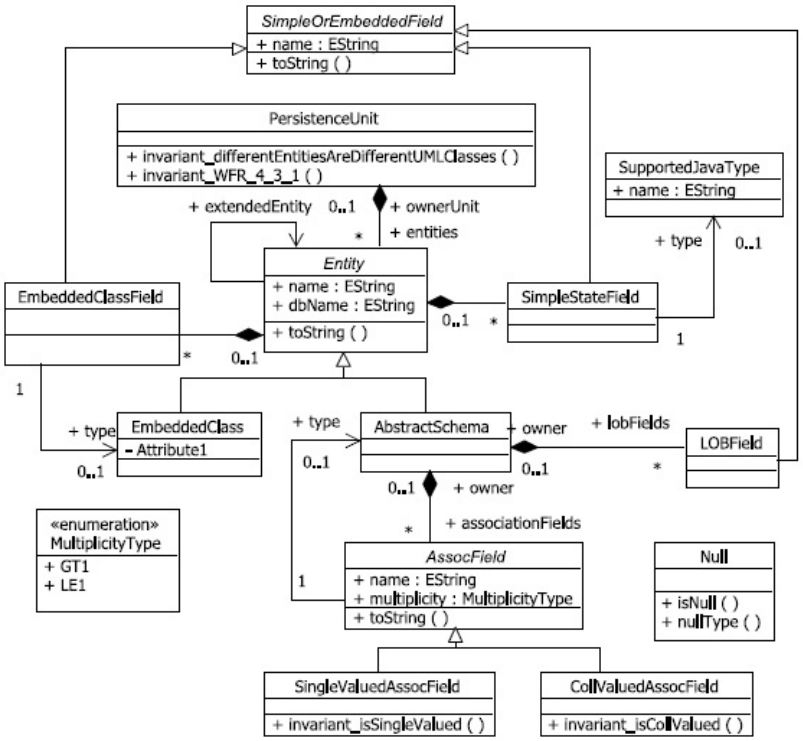


Figure 6.1: Logical database schemas for EJB3QL are instantiations of this metamodel

Chapter 7

Case Study

Summary. *In this chapter we will conduct a case study to exemplify the prototype.*

7.1 Case Description

The following case study is the same used by Halpin to illustrate the CSDP steps in his book, pg.193-198. [HM08]

A description of the case study is given briefly: A business domain concerns a compact disc (CD) retailer who uses an information system to help with account and stock control, and to provide a specialized service to customers seeking information about specific musical compositions and artists. Each disc contains several individual musical items, referred to as *tracks*. Although compact discs usually have about 20 tracks, for this example only a few tracks are listed. Each compact disc has a CD number as its preferred identifier. Although not shown here, different discs may have the same name. Note that *CD* is used here in a genetic sense, like a catalog stock item or car model.
The retailer may have many copies of CD 654321-2 in stock, but for our purposes these are all treated as the same CD. An artist is a person or a group of persons. For a given CD, a main artist is listed if and only if most of the tracks on the disc are by this artist. The record company that releases the disc must be recorded. Within the context of a given CD, tracks are identified by their track number, or sequential position on the disc. But there are many CDs in this domain, so we need both the CD number and the track number to identify a track.
The duration of a track is the time it takes to play. Each track has exactly one duration, measured in seconds. Most tracks have one or more singers. Some tracks may have no singers. For each month that has passed, figures are kept of the quantity of copies sold and the net revenue (profit) accruing from sales of the compact discs in that month.

The first step in order to model this application with the xText editor is to create a graphical ORM2 notation of the UoD. This can be done by applying the CSDP

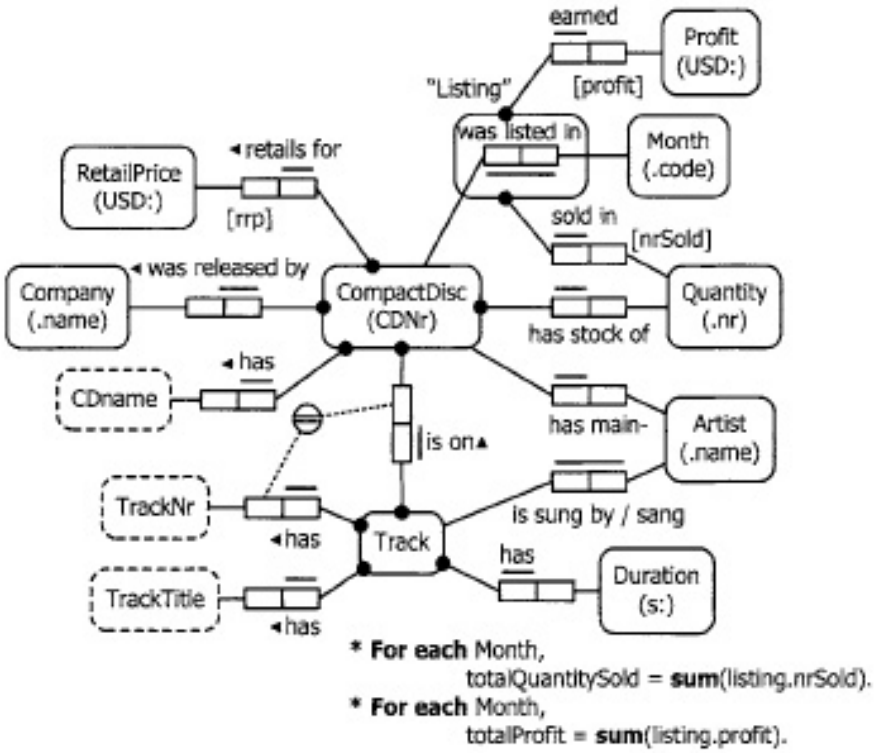


Figure 7.1: Schema of the application

steps. (see [HM08]). For simplicity we will use the graphical ORM2 model used by Halpin (see 7.1).

After having the schema, we can begin writing our model. First we declare the entities, the values and the predicates of the schema. The arity will be two for all predicates, we can add that property or skip it. Then we can slowly add other attributes like nesting. When done, the constraints can be added. Starting with uniqueness constraints we can add the constraints to our model.

This case modeled in the editor looks something like in 7.2. This model can be translated into a E/R counterpart.

For the complete source-code for the model see Appendix D.

Figure 7.2: Model in the outline view of the editor

Chapter 8

Conclusion

Summary. *This chapter serves as an opportunity to list the results of this paper.*

This paper covered an attempt to translate ORM2 to O/R counterparts by designing and implementing a prototype, from the theoretical perspective of the language until the implementational issues. The work in this paper has showed some of the difficulties when trying to translate from a conceptual model like ORM2 into a O/R model.
One of the first problems comes in the design phase with the need to restrict the conceptual language to a subset well defined but less powerful, in order to be able to input the data in a software tool. We must limit the expressiveness of ORM2 on a subset, well aware that we lose some of the features of the language.

There are a multitude of constraints that have no equivalence in other models, that must be regarded when designing a translating tool. Thus comes the problem of either losing information or passing it along as annotations. Also, two different models in ORM2 could end up being translated and implemented similar in an O/R model, if the constraints that set them apart are not translatable. Constraints like occurrence contraints, object cardinality, role cardinality, value-comparison and many more. Derivation rules are complex and can't be translated due to high complexity. They can only be *translated* as text. Some aspects of ORM2 are only partially considered by O/R counterparts, like for example arity of predicates, where many models dont have unary associations.

About the tooling aspect of this paper, xText has proven a powerful tool with many possible usages. Still in its infancy, not all aspects of this tool have been explored. While trying to export the results from the editor, more support would lead to easier use of the models created in different tools. This is the main reason why a working translator was not yet possible to be finished from the model.

Appendix A

xText Grammar File

```
grammar org.xtext.example.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/MyDsl"

//-----------------Model-------------------------//
model:
  (elements+=element)*;

//Elements can be Objects or Predicates
element:
  object_type | predicate | constraint| derivation_rule| scheme| instance;

//-----------------Objects----------------------//
//Objects are Entities or Values
object_type:
   entity_type | value_type;

//Entities
entity_type:
  "entity" name=ID (isPrimitive?="subtypes" supertype=[object_type])? "{"
  (isNested?="nests" predicate=[predicate])?
  ("definition" def=STRING)?
  ("subconstr" sub=STRING)?
  "}";

//Value is a DataType or not
value_type:
  data_type | "value" name=ID (isPrimitive?="subtypes" supertype=[object_type])? ;

//DataTypes includes Integer, String, Boolean...
data_type:
  "data" STRING | "data" INT | "data" type=[value_type];


//-----------------Predicates--------------------//
//Predicate
```

```
predicate:
 "predicate" name=ID "{"
 "part" (participants+=participant",")+
 (arity=arity)?
 (isDerived?="derived")?
 "}";

//Participants in the predicate
participant:
  type=[object_type] role=role;

//Role
role:
  "role" name=ID (isMandatory?="mand")?;


//Arity
 arity:
   "arity" value=INT;


//-----------------Constraints-------------------//
//Constraints
constraint:
  uniqueness_constr|klc_constr|ior_constr|xor_constr|enum_constr|range_constr|
  rv_constr|subset_constr|equality_constr|exclusive_constr;

//Uniqueness-Constraint
uniqueness_constr:
  "uc" name=ID "{"
  predicate=[predicate]
  (l_lines+=c_line)+
"}";

c_line:
 (content+=[role])+ ",";

//KeyLengthCheck-Constraint
klc_constr:
  "klc" name=ID "{"
  predicate=[predicate]
  "}";

//Inclusive-OR
ior_constr:
  "iorc" name=ID "{" (roles+=[role])+ "}";

//Exclusive-OR
xor_constr:
  "xorc" name=ID "{" (roles+=[role])+ "}";
```

```
//Enumeration_Constraint
enum_constr:
  "enumc" name=ID "{"ref_value=[value_type] "(" (instances+=instance)* ")" "}";

//Range-Constraint
range_constr:
  "rangec" name=ID "{"ref_value=[value_type] "(" from=instance ".." to=instance ")" "}";

//RoleValueConstraint
rv_constr:
 "rvc" name=ID "{" ref_role=[role] (instances+=instance)* "}";

//Subset-Constraint
subset_constr:
  "ssc" name=ID "{"subset_role=[role] superset_role=[role] "}";

//Equality-Constraint
equality_constr:
  "eqc" name=ID "{"role1=[role] role2=[role] "}";

//Exclusion-Constraint
exclusive_constr:
  "xc" name=ID "{" (roles+=[role])+ "}";

//-----------------Derivation Rules-------------//

//Derivation Rule
derivation_rule:
  "derivation" name=ID STRING;

//-----------------Reference Schemes------------//
//Reference_Schemes - Injection
scheme:
 injection;

injection:
   "inject" object=[entity_type] name=[object_type];

//-----------------Instances--------------------//

//Instance
instance:
  name=ID;
```

Appendix B

Cuyler and Halpin's Metamodel

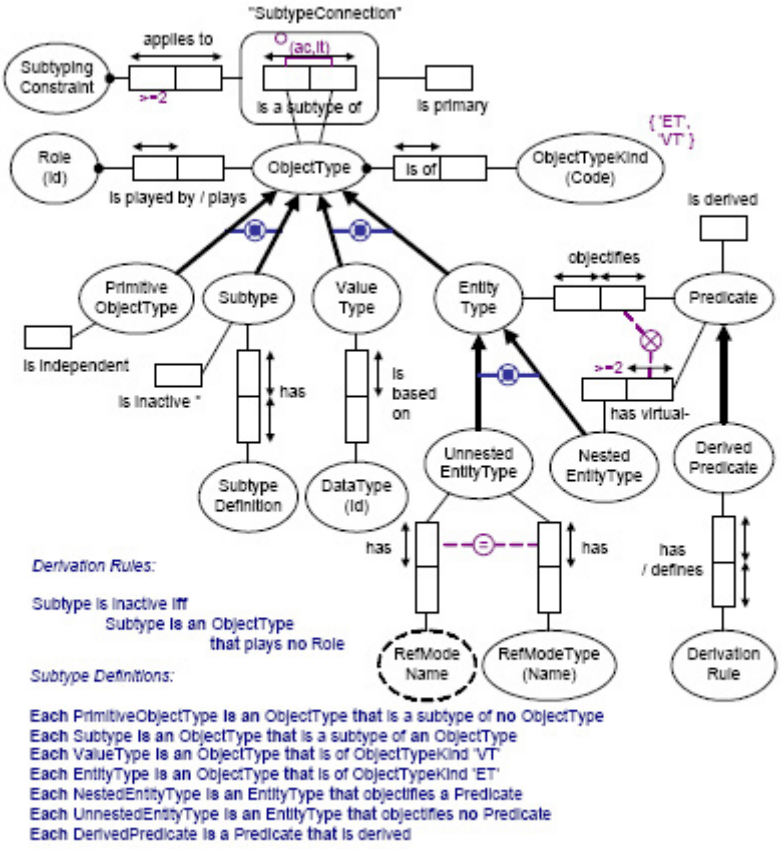


Figure B.1: Main Types in ORM2 as described by Cuyler and Halpin

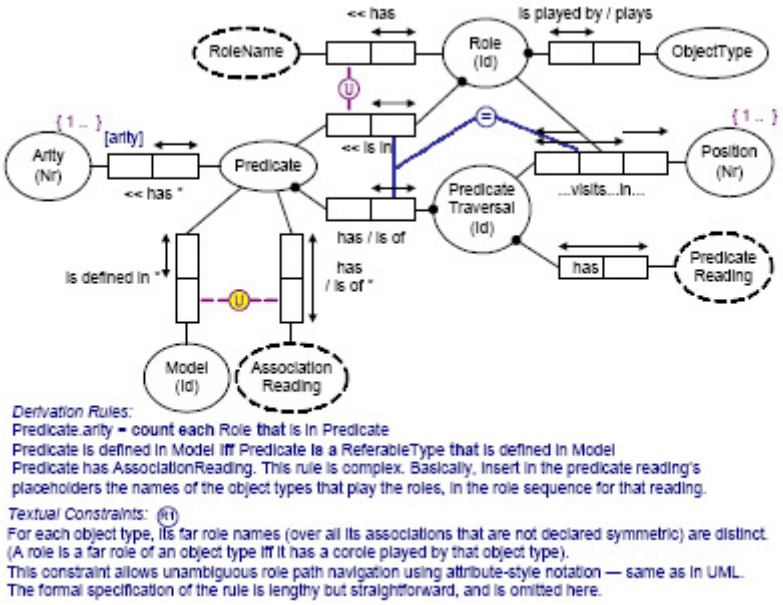


Figure B.2: Naming of ORM2 roles, predicates and associations as described by Cuyler and Halpin

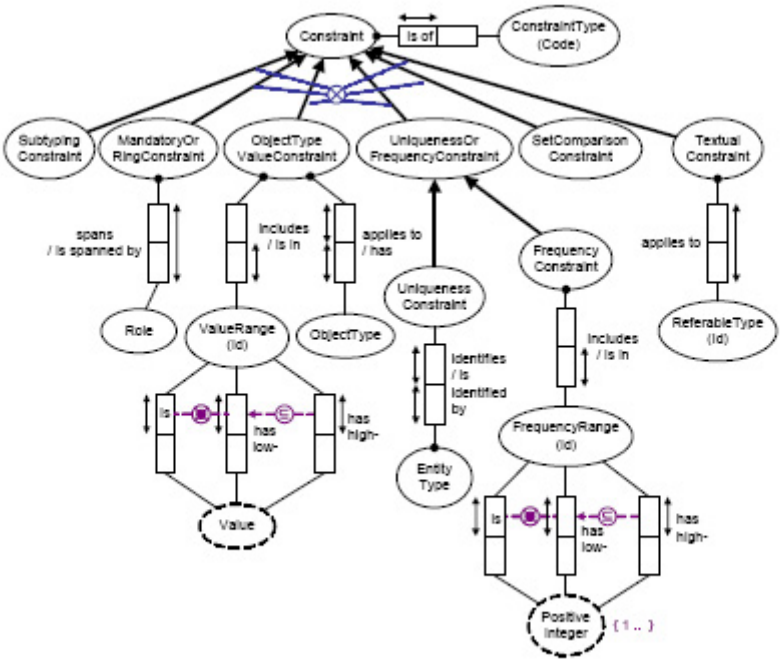
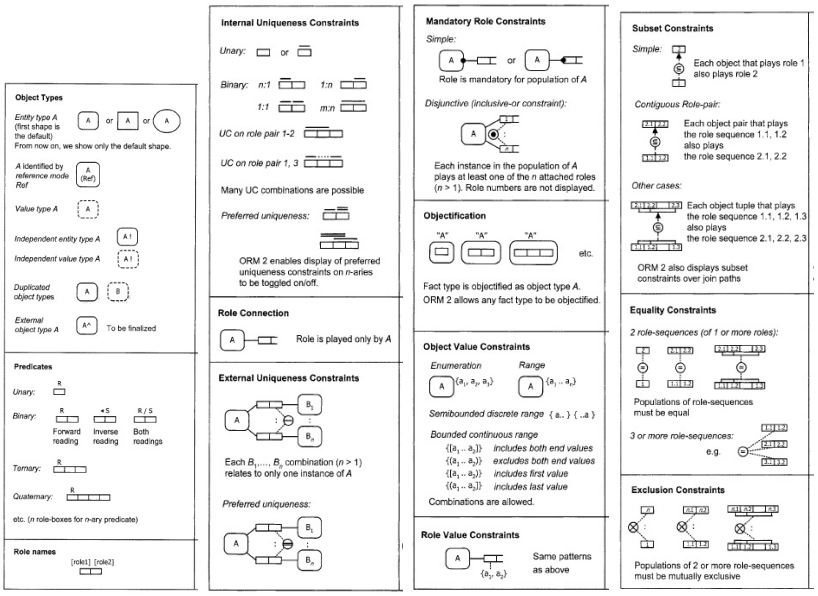
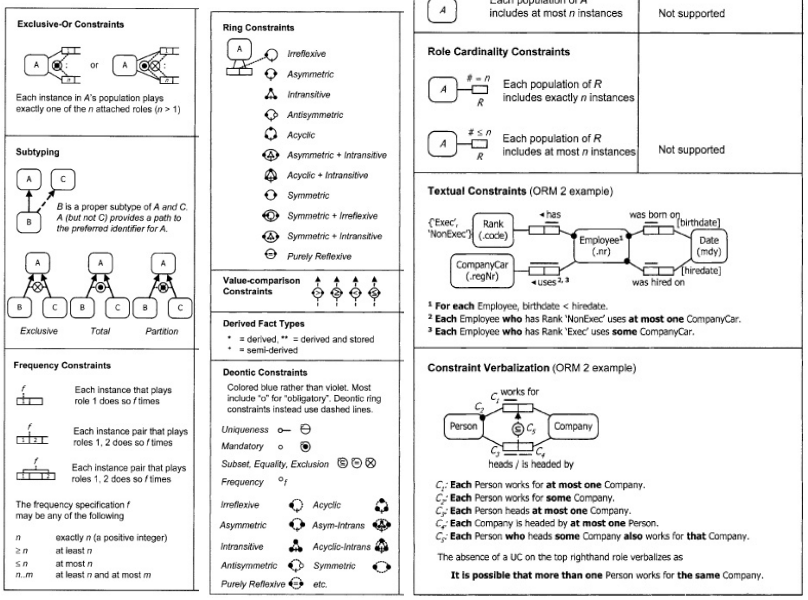


Figure B.3: ORM2 Constraints as described by Cuyler and Halpin

Appendix C

ORM2 Symbols


Object Types
Entity type A (first shape is the default)
From now on, we show only the default shape.
A identified by reference mode Ref
Value type A
Independent entity type A
Independent value type A
Duplicated object types
External object type A
To be finalized
Predicates
Unary:
Binary:
Forward reading
Inverse reading
Both readings
Ternary:
Quaternary:
etc. (n role-boxes for n-ary predicate)
Role names
Internal Uniqueness Constraints
Unary:
Binary: n:1 1:n 1:1 m:n
UC on role pair 1-2
UC on role pair 1, 3
Many UC combinations are possible
Preferred uniqueness:
ORM 2 enables display of preferred uniqueness constraints on n-aries to be toggled on/off.
Role Connection
Role is played only by A
External Uniqueness Constraints
Each B1, ..., Bn combination (n > 1) relates to only one instance of A
Preferred uniqueness:
Mandatory Role Constraints
Simple:
Role is mandatory for population of A
Disjunctive (inclusive-or constraint):
Each instance in the population of A plays at least one of the n attached roles (n > 1). Role numbers are not displayed.
Objectification
"A"
etc.
Fact type is objectified as object type A.
ORM 2 allows any fact type to be objectified.
Object Value Constraints
Enumeration
Range
Semibounded discrete range
Bounded continuous range
includes both end values
excludes both end values
includes first value
includes last value
Combinations are allowed.
Role Value Constraints
Same patterns as above
Subset Constraints
Simple:
Each object that plays role 1 also plays role 2
Contiguous Role-pair:
Each object pair that plays the role sequence 1.1, 1.2 also plays the role sequence 2.1, 2.2
Other cases:
Each object tuple that plays the role sequence 1.1, 1.2, 1.3 also plays the role sequence 2.1, 2.2, 2.3
ORM 2 also displays subset constraints over join paths
Equality Constraints
2 role-sequences (of 1 or more roles):
Populations of role-sequences must be equal
3 or more role-sequences:
e.g.
Exclusion Constraints
Populations of 2 or more role-sequences must be mutually exclusive


Exclusive-Or Constraints
Each instance in A's population plays exactly one of the n attached roles (n > 1)
Subtyping
B is a proper subtype of A and C. A (but not C) provides a path to the preferred identifier for A.
Exclusive
Total
Partition
Frequency Constraints
Each instance that plays role 1 does so f times
Each instance pair that plays roles 1, 2 does so f times
Each instance pair that plays roles 1, 2 does so f times
The frequency specification f may be any of the following
n exactly n (a positive integer)
≥ n at least n
≤ n at most n
n..m at least n and at most m
Ring Constraints
Irreflexive
Asymmetric
Intransitive
Antisymmetric
Acyclic
Asymmetric + Intransitive
Acyclic + Intransitive
Symmetric
Symmetric + Irreflexive
Symmetric + Intransitive
Purely Reflexive
Value-comparison Constraints
Derived Fact Types
* = derived, ** = derived and stored
* = semi-derived
Deontic Constraints
Colored blue rather than violet. Most include "o" for "obligatory". Deontic ring constraints instead use dashed lines.
Uniqueness
Mandatory
Subset, Equality, Exclusion
Frequency
Irreflexive
Asymmetric
Intransitive
Antisymmetric
Purely Reflexive
Acyclic
Asym-Intrans
Acyclic-Intrans
Symmetric
etc.
Object Cardinality Constraints
Each population of A includes exactly n instances
Each population of A includes at most n instances
Not supported
Role Cardinality Constraints
Each population of R includes exactly n instances
Each population of R includes at most n instances
Not supported
Textual Constraints (ORM 2 example)
{'Exec', 'NonExec'}
Rank (.code)
has
Employee[1] (.nr)
was born on
[birthdate]
Date (.mdy)
CompanyCar (.regNr)
uses[2, 3]
was hired on
[hiredate]
[1] For each Employee, birthdate < hiredate.
[2] Each Employee who has Rank 'NonExec' uses at most one CompanyCar.
[3] Each Employee who has Rank 'Exec' uses some CompanyCar.
Constraint Verbalization (ORM 2 example)
works for
Person
Company
heads / is headed by
C1: Each Person works for at most one Company.
C2: Each Person works for some Company.
C3: Each Person heads at most one Company.
C4: Each Company is headed by at most one Person.
C5: Each Person who heads some Company also works for that Company.
The absence of a UC on the top righthand role verbalizes as
It is possible that more than one Person works for the same Company.

Appendix D

Case Study Model

```
entity CompactDisc
{
}
entity Track
{
}
entity RetailPrice
{
}
entity Company
{
}
entity Quantity
{
}
entity Artist
{
}
entity Duration
{
}
entity Profit
{
}
entity Month
{
}
entity Listing
{
nests was_listed_in
}


value CDname
value TrackNr
value TrackTitle
```

```
value USD
value name
value code
value nr
value s
value CDNr

predicate was_listed_in
{
part CompactDisc role compactdisc1, Month role month1,
}
predicate retails_for
{
part CompactDisc role compactdisc2, RetailPrice role retailprice2,
}
predicate was_released_by
{
part CompactDisc role compactdisc3, Company role company3,
}
predicate has
{
part CompactDisc role compactdisc4, CDname role cdname4,
}
predicate has2
{
part Track role track5 mand, TrackNr role tracknr5,
}
predicate has3
{
part Track role track6 mand, TrackTitle role tracktitle6,
}
predicate earned
{
part Listing role listing7, Profit role profit7,
}
predicate sold_in
{
part Listing role listing8, Quantity role quantity8,
}
predicate has_stock_of
{
part CompactDisc role compactdisc9 mand, Quantity role quantity9,
}
predicate has_main
{
part CompactDisc role compactdisc10 mand, Artist role artist10,
}
predicate is_on
{
part CompactDisc role compactdisc11 mand, Track role track11,
}
```

```
predicate is_sung_by
{
part Track role track12, Artist role artist12,
}
predicate has4
{
part Track role track13 mand, Duration role duration13,
}

inject RetailPrice USD
inject Company name
inject Profit USD
inject Month code
inject Quantity nr
inject Artist name
inject Duration s
inject CompactDisc CDNr

uc uc_was_listed_in
{
was_listed_in
compactdisc1, month1,
}

uc uc_retails_for
{
retails_for
compactdisc2, retailprice2,
}

uc uc_was_released_by
{
was_released_by
compactdisc3, company3,
}

uc uc_has
{
has
compactdisc4, cdname4,
}

uc uc_has2
{
has2
track5 , tracknr5,
}

uc uc_has3
{
has3
track6 , tracktitle6,
```

```
}

uc uc_earned
{
earned
listing7, profit7,
}

uc uc_sold_in
{
sold_in
listing8, quantity8,
}

uc uc_has_stock_of
{
has_stock_of
compactdisc9 , quantity9,
}

uc uc_has_main
{
has_main
compactdisc10, artist10,
}

uc uc_is_on
{
is_on
compactdisc11, track11,
}

uc uc_is_sung_by
{
is_sung_by track12, artist12,
}

uc uc_has4
{
has4 track13, duration13,
}
```

Bibliography

[act09] Homepage activefacts, 07.07.2009. http://dataconstellation.com/ActiveFacts/.

[Dau06] Berthold Daum. *Das Eclipse-Codebuch - 182 Tipps, Tricks und Lösungen für Eclipse-spezifische Probleme*. Dpunkt Verlag, 2006.

[dog09] Homepage dogmamodeler, 07.07.2009. http://www.jarrar.info/Dogmamodeler/.

[ecl] Eclipsepedia - the eclipse wikipedia. http://wiki.eclipse.org/Main_Page.

[gan09] Homepage ganttpv, 07.07.2009. http://www.pureviolet.net/ganttpv/whyganttpv/index/.

[Gar06] Miguel Garcia. Formalizing the well-formedness rules of EJB3QL in UML + OCL, 2006. http://www.sts.tu-harburg.de/~mi.garcia/pubs/atem06/JPQLMM.pdf.

[HM08] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann, 2008.

[JKS05] Terry Halpin John Krogstie and Keng Siau. *Information Modeling Methods and Methodologies*. Idea Group Publishing, 2005.

[Kue08] Thomas Kuenneth. *Einstieg in Eclipse 3.4 - Aktuell zu Ganymede, 2te Auflage*. Galileo Press, 2008.

[oAW09] Homepage openarchitectureware, 07.07.2009. www.openarchitectureware.com.

[orm09] Homepage orm foundation, 07.07.2009. http://www.ormfoundation.org.

[tmf09] Homepage tmf project, 07.07.2009. http://www.eclipse.org/Xtext/.

[ung08] Sebastian Boßung. *Conceptual Content Modeling (Languages, Applications and Systems)*. dissertation.de, 2008.

[vis09] Visitor pattern, 07.07.2009. http://en.wikipedia.org/wiki/Visitor_pattern.

[xte09] xtext documentation, 07.07.2009. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xtext.doc/help/syntax.html.