CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MODEL DRIVEN APPLICATION AND DATABASE CO-EVOLUTION

Ing. Ondřej Macek

A thesis submitted for the degree of Doctor

PhD programme: Electrical Engineering and Information Technology
Specialization: Computer Science and Engineering

July 2014

Abstract and Contributions

An evolution of application's persistent objects affects not only the source code but the stored data as well. The change is usually processed in three steps: application evolution, database schema evolution and data migration. Because the process is often done manually, it is ineffective and error prone. We provide a solution in form of a model driven framework. The framework is described as a formal model which is capable to migrate database according to an evolution of the application code. The feasibility of the change and its data-secure processing is addressed in the framework as well. Because the evolution is not always straightforward process the capabilities of the proposed framework in the area of versioning are discussed as well and an operation-based versioning system is proposed. Finally the prototype implementation is introduced as well as lessons learned from its implementation and usage.

The main contributions are:

1. The architecture of the framework for application and database co-evolution is provided and discussed in context of application and database co-versioning.
2. The formal model in the Z-language specifies the most common evolutionary cases (refactorings) and their impact on application and database. This formal framework can be used as an entry point for an implementation of a MDD tool for co-evolution.
3. The formal model describes the basic cases of application and database co-versioning such as branching, merging of the repository and reverting of a transformation.
4. The formal is verified by prototype called MigDb, which provides feedback on the formal framework in a real-world scenario.

The results of the thesis improve the understanding of the area of model driven application and database co-evolution and co-versioning.

Keywords:
application and database co-evolution, application and database co-evolution, model driven development, formal model

Thesis Supervisor:

doc. Ing. Karel Richta CSc.

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Praha 2
Czech Republic

Acknowledgements

First of all, I would like to express my gratitude to my thesis supervisor, doc. Ing. Karel Richta CSc. His efforts as thesis supervisor contributed substantially to the quality and completeness of the thesis. I thank my colleagues for their support during my study. I thank especially to Božena Mannová, Martin Komárek, Jan Kubr, Pavel Strnad and Ondřej Votava. I thank to Tomáš Čechal for proofreading.

Finally, my greatest thanks to my family and friends whose support was of great importance during the whole period of PhD study.

Contents

List of Figures

Listings

Acronyms

CVS	Concurrent Versioning System
CRM	Customer Relationship Management
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
MDA	Model Driven Architecture
MDD	Model Driven Development
ORM	Object-Relational Mapping
OCL	Object Constraint Language
QVT	Query/View/Transformation
SQL	Structured Query Language
VCS	Version Control System
XML	eXtensible Markup Language

Chapter 1

Introduction

The evolution (change) of a software is a common issue during the software development. It occurs for many reasons in all phases of the software lifecycle. A success of a company may rely on the speed of the development team and its capability to create a new evolved software. The software is usually a complex system, which consists of multiple layers (e.g. code, database, GUI) and an evolution very often affect more than one layer. The evolution of different layers has different complexity. The code of an application can be evolved relatively fast thanks to the developer's IDE capabilities, whereas the evolution of database of the same application can be very difficult especially in case there are data stored in the database, because it has to be done manually. A change may affect only a single software layer, but often it has to be propagated from one layer to another.

Moreover, the software evolution is often not a straightforward process. Developers have to implement prototypes of the final product or they have to explore a possible solution of a problem. When software is finished and prepared for release, developers start to develop the next version. They are fixing errors or customizing software for concrete customers and their requirements. All these activities lead to creation of various versions of the software. There are two main reasons for maintaining various versions of software in general. First reason is different versions of the software, which differs significantly, however they have some core functionality in common e.g. software, which design is customized for various customers, but the functionality is the same for all customers. The second reason for maintaining various versions of software is the need for maintaining the history of software development e.g. the case when we want to return to the code of the last stable software release, when a prototype of the new software version contains bugs. Maintaining various versions of software become an integral part of software development process and the version control systems (VCS) are widely used in the community of software developers.

A common situation when an application's model layer (so called entities) evolves together with a database layer is addressed in this thesis. Refactoring [1] is a very popular practice in object-oriented environments for evolving the source code and software architecture. Evolution of database schema and stored data is implemented separately from source code refactoring, although the change of entities also affects the database. Object-relational mapping (ORM) frameworks can help with propagation of the evolution from an application to a database. However, these frameworks are usually neither capable of solving complex refactoring cases nor they migrate data properly.

We propose a solution for the problem of entities and database co-evolution and co-versioning, which uses the model driven development (MDD) paradigm i.e. it is based on model transformations. The solution allows automatization of the evolution process as it allows to co-evolve code, database schema and stored data at the same time. Because the evolution process has to be versioned, we discuss all the important aspects of a VCS tool (versioning, reverting, branching and merging) in context of the MDD software evolution.

The idea of the MDD approach for application and database co-evolution is illustrated on a formal specification in the Z-language and a prototype implementation.

1.1 Goals of the thesis

This thesis discuss the idea of the MDD framework for entities and database co-evolution and co-versioning. The main goals of the thesis are:

1. Show that the MDD solution for the co-evolution and co-versioning of application and database is possible and that co-evolution can speed up the development process.

2. Provide a formal model of code and database co-evolution and co-versioning in context of MDD environment. The model of co-evolution should cover the main evolutionary transformations and the model of co-versioning should define branching, merging of the repository and reverting of a transformation.

3. Verify the idea by a prototype implementation. Moreover the implementation should provide a feedback on the framework behavior in real-world scenarios.

1.2 Organization of the thesis

The problems connected with application and database co-evolution and co-versioning are described in Sect. 2 as well as solution proposals based on the MDD paradigm.

The state of the art is overviewed in Sect. 3, where different approaches to co-evolution and co-versioning and similar solutions are introduced.

The formal model of application and database co-evolution is introduced in Sect. 4 and 5, where the architecture of the framework is introduced as well as static models. The set of transformation for co-evolution is defined in Sect. 6. Finally the model of co-versioning is introduced in Sect. 7.

The implementation of a prototype is introduced in the Sect. 8. The implementation is verified on a case-study and a lessons learned are discussed as well.

Chapter 2

Problem Statement

The evolution of the software is a common issue during software development lifecycle. The evolution can affect one or more layers of the software. The process of change propagation between various software layers is sometimes time consuming and error prone. The importance of evolution grows with the use of agile development where the prototyping is a common approach as well as small releases. The change in requirements, continuous integration of requirements during prototyping, software architecture improvement - all this can cause an evolution of the whole software. This chapter introduces the problem of software evolution in the context of data evolution from the point of view of a software developer.

We focus on a situation when data structure changes as a consequence of an evolution. The area of interest is not the data evolution in the context of the whole application, which can consist of many layers such as GUI, security layer, etc. Our focus is only on the layer of persistent objects (so-called entities) and the database itself. From our point of view, the software is reduced to one layer (entities), whereas the database consists of two layers - the database schema and data.

This thesis address the common situation, when a software is developed in an object-oriented language (such as Java [2] or C# [3]) and a relational database (such as MySQL [4] or PostgeSQL [5]) is used. This software architecture was chosen because it is commonly used by programmers in real-world software applications. Entities' layer and database are linked together by an object-relational mapping which overcame the gap between the world of objects and the world of relational data.

The developers' point of view limits the software evolution to cases which origin on the level of entities and are propagated to the database (so called code-first approach). The change of a database structure or of object-relation mapping (e.g. because of database performance optimization) is not regarded as an evolution in this thesis.

2.1 Data Evolution Process

The process of data co-evolution in context of application and database is often processed by the object-relational mapping framework, which is used by developers. There are lot of object-relational mapping frameworks available for developers, and some of them provide tools for database evolution as well. Hibernate [6] is one of the most popular ORM frameworks in the Java community. It provides customizable ORM for a wide range of databases, however it does not support complex database evolution. It is capable only to create a new table or to add a new column, hence it is not possible to, for example, drop a table or copy values from one column to another. Another example is the Active Record [7], which is an ORM framework in the Ruby on Rails environment. Since its first version, it has contained support for database evolution according to the create-update-delete principle. In the form of so-called migrations [8] which can be extended by adding user (SQL) commands. Entity Framework [9] is Microsoft's ORM solution for the .NET platform, which evolves rapidly in last years. Its capabilities of data evolution support are similar to those of Active Record.

Each of presented ORM frameworks provides support for evolutions, which change the database structure according to change in of the application. These frameworks are capable e.g. to add a new class as an entity and to create a corresponding table in the database etc. On the other hand, only two of them (Active Record and Entity Framework) are capable to update or delete structure and preserve stored data. In the case of evolution, which needs to manipulate data, e.g. moving an attribute, none of the mentioned frameworks provides a built-in solution. Active Record and Entity Framework allow developers to describe the data migration manually.

Another solution widely used by developers is a tool for database refactoring and evolution called Liquibase [10]. It is capable to migrate both database schema and stored data. The evolution is described in form of a XML document, which can be interpreted in various databases. At the moment the transformations for data migration are not implemented in the default set of Liquibase transformations.

The process of data evolution can vary from project to project, but the main scenario follows the illustration in Fig. 2.1. The process is initiated at the level of entities and

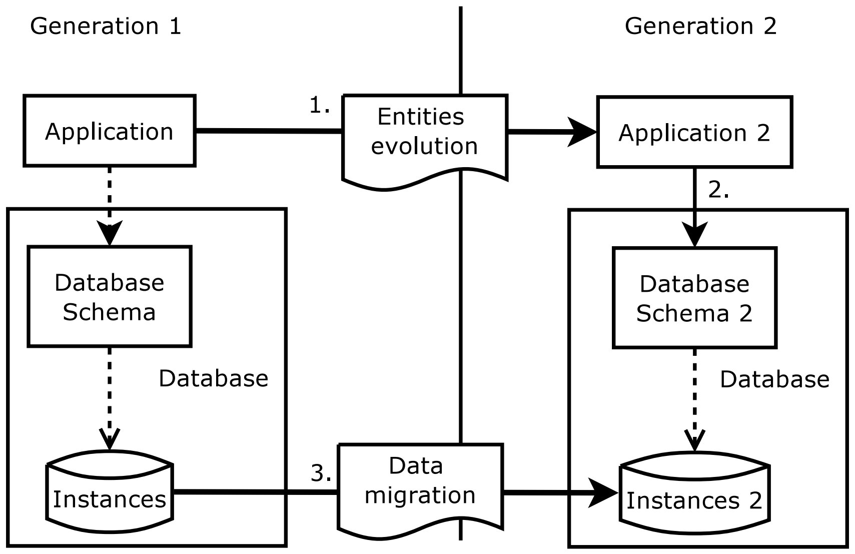


Figure 2.1: The illustration of the software components and theirs evolution: first application code is evolved, then the database schema is generated and finally data are migrated.

then the change is propagated to the other parts of the software. The process consists of the following steps:

1. A new version (generation) of entities is created - the code is changed. The developer processes this change.
2. The database schema has to be adapted to the new version of entities. This change can be processed manually, however numerous ORM frameworks provide a generator of the schema according given entities and mapping.
3. Stored data have to be migrated from the old schema to the new one. ORM frameworks usually do not support data migration, therefore the developer together with the database administrator (if needed) prepare scripts for data migration. This step is challenging because the migration has to respect not only new entities, ORM and database schema (let us say technical features of software), but it has to respect the domain as well. Another problem with the data migration is its feasibility needs to be verified for each running instance of software, because the stored data and their relations can vary from instance to instance. Therefore developers should prepare not only migration scripts, but a verification script as well.

These steps have to repeat every time an evolution occurs. At least the last step of data evolution (data migration) needs manual work. Moreover, the transformation semantic has to be defined twice - once for the code and once for the database. Automatization of the data migration can speed up the development process.

2.2 Evolution and Versioning

The software evolution is often not a straightforward process. Developers have to implement prototypes of the final product or they have to explore a possible solution of a problem. When software is finished and prepared for release, developers start to develop the next version. They are fixing errors or customizing software for concrete customers and their requirements. All these activities lead to creation of various versions of the software. Maintaining various versions of software become an integral part of software development process and VCS are widely used in the community of software developers.

There are numerous implementations of VCS available. We differ them into two groups according to the approach they use for versioning. First group of state-based VCS is based on maintaining the states of the software, whereas the second group of operation-based VCS is based on maintaining the transitions from state to its following state. In the following sections, we introduce both groups and we discus how they approach to versioning and their other features (see Sec. 3.2).

The activities connected with the software versioning can be described in terms of evolution. The process of creating new versions is equal to evolution, which preserves all previous states of the software. Branching is the process of creating different versions of the same software - different evolution lines. Reverting a change is equivalent to the degeneration of the software (i.e. backward evolution). The revert in previous state means that the state of the application (and the database) structure is restored. Because the data themselves are not stored in the VCS (which is a best practice) we can lose data we stored since the version we are reverting too. This is problem in case of deployed software when the stored data contain important business information. We have to know the semantics of the change in case we are going to revert it and we want to preserve stored information. The semantics of the change can provide information if the revert is feasible without the data loss and how the data have to be adapted to the reverted state. The semantics of the change is maintained as a text message from a developer, which often explains why the code was changed (e.g. to solve some bug) but not how it was changed. The information how the code was changed is important for the revert without data loss. If it is not in the VCS than it has to be obtained e.g. by comparing both states.

Because of this connection between versioning and evolution we decide to pay attention to the phenomenon of software versioning in this thesis as well.

2.3 Problems of Co-Evolution and Co-Versioning

The process of data evolution is considered to be a difficult task during the software development. This is caused due to following reasons:

1. **Manual processing** A lot of current ORM frameworks do not provide support for the data evolution as they implement the ORM only. Some of them provide the solution for the database schema evolution and do not handle the data migration; therefore the data evolution process needs to be processed manually.

2. **Multiple definitions of the evolution** Even when there is only one small evolutionary step to be processed, a developer has to define its semantics twice. The basic definition is for entities; the second is for the database (both schema and data).

3. **Feasibility verification** Because data evolution influences three different layers (entities, database schema and stored data). Its feasibility depends on the evolution's feasibility on each layer. The structure cannot be evolved if it is not feasible to migrate data. The feasibility needs to be verified for each deployed database as stored data differ from the software instance to the instance and thus the evolution feasibility may differ as well.

4. **Verification of success** Once the database code is changed and the database migrated the result should be verified if the migration has been processed properly.

5. **Lack of automatization of the whole process** Set of standard evolutions of the level of entities is known (refactorings) and it is automated by many IDEs. However, this set is not propagated to the database automatically. The lack of automation slows down the software development.

The automatization of data evolution can significantly speed up the time needed for developing a new software release. Our goal is to provide a solution, which is able to verify the feasibility of the evolution process. Functionality of such a tool has to be verified carefully because a data loss during evolution could have serious consequences.

We have extended the data evolution by terms from software versioning - such as versioning, branching etc. This adds two more problems to the list:

6. **Reverting a change** can result in an inconsistent state of the software or in data loss. The existing tools does not provide mechanisms for assuring safe revert.

7. **Semantics versioning of multiple software layers** is not supported by current VCS. Although there is usually one change which affect multiple software layers, the VCS tools are not able to keep the semantics of the change effectively.

2.4 Summary

Data evolution is a common part of a software development process. The current approaches prefer the data evolution on the database schema level and its propagation to the level of entities. We focus on the developers in this thesis in contrast. It means we try to provide support for common developers' activities such as coding and refactoring. The current process of application and database co-evolution has several issues, which have to be solved.

Chapter 3

State of the Art

The need of application and database co-evolution results in many industry or research projects and implementations. These projects are at various formal levels and use various approaches to the data evolution problem. We provide an overview of the main directions and success in the area in this section.

3.1 Database Evolution

There are many tools which help with database administration. These tools are capable of database refactorings. Some of them use a conceptual modeling and MDD.

3.1.1 Tools for Database Evolution

The MeDEA project [11] offers a tool for automatic evolution of both database schema and stored data based on a model-driven approach. The framework is meant to be used by database administrators; therefore it is aimed at database structures (such as views), which are not considered by software developers

PRISM is a research project for data management under schema evolution [12] in contrast with our proposal and with MeDEA it extends the SQL command set by so-called schema modification operators which implement the schema evolution. The project is meant to be used by database administrators (as well as MeDEA).

Project DB-MAIN [13] provides a MDD approach to data evolution of application and database as well. The project is well documented formally. DB-MAIN starts with a database modification and a database migration, and entities are created accordingly;

whereas our focus is on entities evolution, which is then propagated to a database with an emphasis on automation. Our goal is to hide the entire database level from our users.

The project IMIS [14] follows the same idea of applying MDD into evolution of a whole software, but does not provide a formal model or an overview of capabilities (defined transformations). An evolutionary approach to data evolution is described in [15], which allows a database schema change to be propagated into the stored data and entities in the programming language. The evolutions described in the paper are for creating, updating and deleting of basic structural elements.

Most of the frameworks provide good service for database administrators, but they are not design to be used by developers. The frameworks implement only the basic transformations (i.e. the refactorings are not implemented).

3.1.2 Formal and Informal Models

The problem of application and database co-evolution is examined in formal models as well. We provide a several models, which describe the problem. There are many other formal frameworks published [16, 17, 18, 19]. These frameworks use various approaches to describe the changes - extension of relational algebra, graph transformations and others. Most of them focus only on simple scenarios (adding or deleting of table, reference or column) or they consider only the database structure.

The taxonomy of relational database evolution based on the entity-relationship model is proposed in [20]. The evolution is described as a change in the entity-relationship model and change in the relational database. The change semantics patterns in the context of a conceptual schema is described in [21], although its impact on the database schema or data is not described. The main cases of data evolution are defined in both publications, however the description is informal. The extensive set of possible database refactorings is provided in [22], where both schema and data evolution is discussed. The refactorings are intended to be used by database administrators, thus it assumes database-first approach to evolution whereas our proposal is application-first.

A general formal framework for database evolution is defined in [23]. The framework is based on a set of basic graph transformations, which are then extended to transformations of the entity-relationship model. The framework does not consider stored data. The contribution of the formal framework is the definition of equivalent structures in schemas.

A categorical framework for migration of object-oriented systems is proposed in [24]. This framework defines the refactoring of objects, data and methods, which are the main

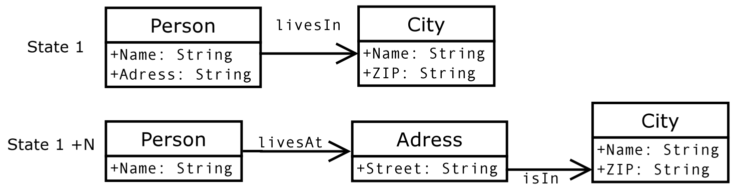


Figure 3.1: A difference between two software states with unclear semantic - one explanation is one attribute (Person.Address) and one association (Person–City) was removed and a new class was added, as well as a new association. Another explanation is the attribute was extracted into a new class and the association was updated.

objectives of the framework. The influence of the object change on a relational database is not considered in the paper as it is aimed at object-oriented systems only.

3.1.3 Model Driven Frameworks

The model driven frameworks for application and database co-evolution described in this section are direct competition to our proposal. In this section we introduce more projects which use the MDD approach to application and database co-evolution (some MDD projects are introduced in Sec. 3.1.1) and we introduce possible MDD approaches to the problem of application and database co-evolution.

Co-Evolution Based on Model Matching

The co-evolution based on model matching uses the backward round-trip engineering, when a backward round-trip algorithm is used to describe the difference between the old and new model of software state (in our case in a model of software). The differences then serve to derive the semantics of the change - the way in which the model was changed between versions.

There is one problem with the use of model matching: cases of evolution, which cannot be derived automatically. A user input is needed in order to verify the differences between the old and the new model. An example of a problematic situation is shown in Fig.3.1. The difference between two states is a missing attribute of a class, association and a new class. There could be several possible interpretations for such a change. The interpretations are not so important for the source code layer, but can have fatal consequences if they refer to stored data in a database.

The process of data evolution proceed in this case in following steps:

1. Initial and final model is compared. The probable causes are derived from the difference between models.

2. User chooses the right cause of evolution and adds detailed information about the transformation.

3. The evolution is propagated into all relevant models of software or into database (via an SQL script).

Advantage of this approach is that there is only one type of artifact needed - the model of software. The rest is derived from the models and the user input. On the other hand, it is therefore difficult to identify large changes in the system.

Co-Evolution Based on Transformations

The second MDD approach to data evolution is based on forward engineering. This approach is used e.g. in [25], where a forward-oriented evolution of application is proposed. In our proposal we focus on the co-evolution and the database. It means that the evolutionary transformations has to be defined for the entities as well as for the database schema and data. In our case, the evolution of the whole software is interpreted as entities evolution and database evolution. The difference between initial and final software version is defined as a transformation or sequence of transformations.

The principle is illustrated in Fig. 3.2. Each transformation contains information needed for entities evolution as well as for database evolution. These transformations are interpreted in the context of the application, the database schema and data. The evolution of the database schema model needs to be interpreted in a real database instance - it means the SQL scripts for schema alternation (re-generation) and for data migration have to be created and executed.

The evolution then proceeds in following steps:

1. The evolution is defined as a subset of transformations from the set of the evolutionary transformations.

2. The evolution is interpreted in the application level and a new model of the application is created. Verification of evolution feasibility on the application level is a part of the creation of the new application model.

3. The evolution is interpreted for the database level which means:

   (a) A new model of the database schema is created.

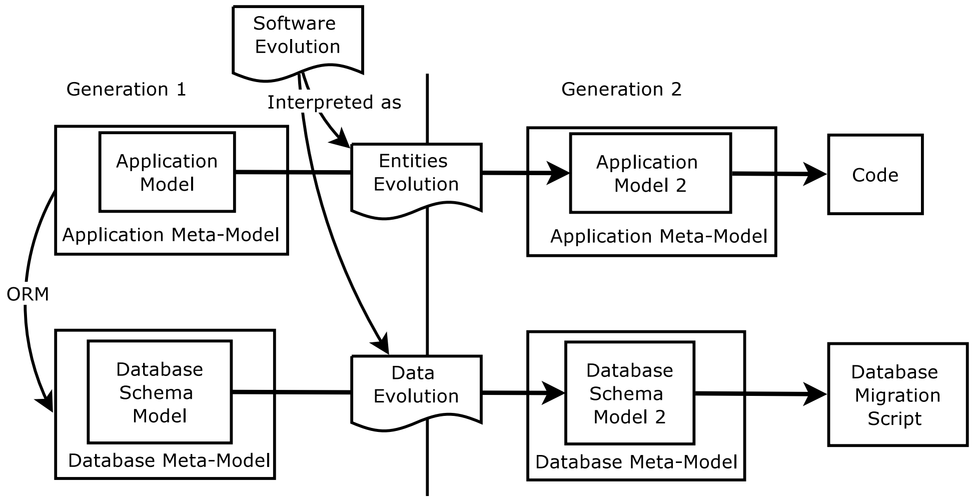


Figure 3.2: The architecture of MDD based framework for data evolution.

(b) An SQL migration script is created for the stored data. The script is created in SQL dialect specific to the concrete database used in the software.

4. The SQL script created in the previous step 3b can be executed on all databases deployed. Feasibility of the evolution and data consistency has to be verified.

This approach is more suitable for developers because it respects the direction of evolution (from old to new) and thus corresponds more with the way a programmer thinks, and because there is no need to implement a round-trip algorithm for model matching. Moreover there is one further advantage: because the evolution has only one source. The semantic of the evolution is known on all levels all the time, and the behavior of the evolution on the entities and the database schema level can be simulated. This approach is able to solve all issues related to evolution mentioned in Sec. 2.3.

Example of such an tool is a meta-model based approach to data evolution is proposed in [26]. The solution is based on extended UML meta-model. It provides similar capabilities to change in application and database as our proposal does. In contrast our proposal is created with respect to ORM domain and therefore we extended the application meta-model with constructs typical for this domain.

3.1.4 Data Evolution of Non-relational Databases

The issue of database evolution is discussed not only for relational databases, but for object-oriented databases as well. There is proposed an approach, which evolve schemes of object-oriented databases in [27] which based on a set of schema invariants (representing the attributes of the schema) and a set of rules for invariant preserving in case schema changes. The impact of evolution on instances is considered as well. The approach is presented informally.

The formal definition of evolution of object-oriented databases is presented in [28]. The approach is based on a type system representing the database schema as axioms, which defines the schema structural constraints, and changes, which evolve the system. There are three change operators: add, remove and modify defined in the paper. The model satisfies the attributes of soundness and completeness.

The SERF framework [29] provides a similar functionality to the user of an object-oriented database as our proposal for the user of a software based on ORM. The SERF approach is based on templates, which a user can create and maintain. Limitation and condition of some evolution cases are discussed in [30].

There are also works in the areas of object databases and XML databases [31]. These works provide solutions specific to the concrete types of databases using various ranges of solutions - domain specific languages [29], extensions of existing standards or MDD [32] or formal specification [33]. These solutions are inspiring, however the domain of the ORM has its specific issues, so a solution from another domain has to be adapted carefully.

3.2 Version Management

The versioning of software is represented by the two main approaches - state-based and operation-based versioning systems. Because we use the MDD we are interested in versioning of models as well. Techniques of version management are introduced in [34], where comprehensive survey on software merging is provided as well. Concepts of a unified versioning tool are presented in [35].

3.2.1 State-based VCS

State-based VCS such as CVS [36], SVN [37], Mercurial [38] or Git [39] are very popular nowadays. Their implementation is based on preserving of an ordered sequence of software states. The states can be saved in various forms in the VCS. The easiest way is to store a complete snapshot of all files, however this could be quite ineffective approach, because the size of the storage increases dramatically with each commit. Thus, some VCS store only the files, which have changed since the previous commit.

It is important to notice that the two following states could differ significantly as the frequency of commits depends on the developer. In an extreme case, the initial state could be an empty initial commit and the following final state can contain the final version of the software. The evolutionary approach depends on developer, which uses the VCS. The semantics of the difference should be described in the message appended to the commit, however many developers are very undisciplined and their messages are very vague (e.g. 'minor changes'), then the difference semantics stays unclear (as it was in example in Fig. 3.1). If a change affects more than one software layer is often hard to identify where the change origins from or its semantics is often unclear.

Approaches similar to the versioning are used for creating branches. Some VCS create a new copy of the software, whereas other create a branch as a new set of files, which changed. The process of branch merging consists of three steps: 1) identification of collisions 2) collision solving 3) merge of compatible branches. The identification of collisions depends on concrete VCS and format of stored files e.g. the comparison based on the difference of lines is used in case the plain text files. The detection of collisions is based on the difference between two states. The collision solving in common state-based VCS is done manually by accepting one or the other version as final. After all conflicts are solved, merging both branches into one can create the final state.

The fact the most popular VCS are used for versioning of plain text files means that other file formats are hard to maintain in these VCS. Problematic can be binary files, whose content is not human-readable and thus the conflict detection is impossible. This problem is observed not only in case of binary files but in the case of XML files as well. The XML format is very popular format for storing configuration records, models and other kinds of structured information. In case of the XML files, the human readability is one problem, the second problem is that a different structure (order of elements) of a XML file is interpreted as a change in the VCS although the semantics of the document did not change.

The versioning of plain text files without exact record of the semantics of the change between versions is problematic in situation when the change affect multiple software layers. A change in security configuration can affect stored data and GUI as well and vice versa. The origin and the semantics of the change is hard to obtain from the VCS if there is no information about change's semantics stored in the VCS.

The state based VCS are very popular nowadays because of its simplicity and orientation on plain text files containing code. Although the semantics of change is defined only by an informal textual message created by a developer. Nevertheless in case of model driven production line we can reach another limitation - the low capability to maintain versions of models, which are represented in XML or in some proprietary format. In model driven production line we can benefit from usage of an operation-based VCS.

3.2.2 Opertation-based VCS

The operation-based VCS [40] are based on maintaining an initial version of the software and sequence of transformations. Application of the sequence on the initial version produces the actual version of the software. This approach can improve the understanding of change between states [41]. The initial version is usually an empty software (e.g. a basic project structure), however any consistent and valid software can be used as the initial one in case the versioning starts later during the software development. Transformation is the difference between software versions therefore the semantics of the change between versions is always known and the difference always represents one single change.

Creating of branches in operation-based VCS is similar to in the case of the state-based VCS. A new copy containing the whole history record can be created or a new history record can be made, which will contain alternative subsequence of transitions since the point, where the branch was created. The process of merging is again similar, however instead for searching a new consistent state, we are searching for a new well-formed sequence of transitions, which produce the requested final state. The process could consist of the following steps: 1) construction of all possible well-formed sequences of transitions, which conform to the history of both branches 2) choosing sequences, which produce the requested final state 3) picking one sequence as the new history record.

The state-based VCS are used to maintain versions of all kinds of files, although popular implementations recommend to use plain text files. In contrast, operation-based VCS are recommended in situations when the versioned data are structured, because then they can be easily evolved by transformations. Therefore operation-based VCS are popular for versioning of models. On the other hand, this feature limits usability of operation-based VCS, because a set of possible transformations has to be defined for each meta-model (file format or content type). Moreover, a large group of meta-models requires larger group of transitions.

The versioning of multiple software layers in state-based VCS is implemented in the same way as versioning of one software layer or plain text file. In contrast in operation-based VCS we can use the knowledge of the semantics of a change to effectively co-version multiple software layers at once. We propose three possible three approaches to versioning of multiple software layers in operation based VCS in [42].

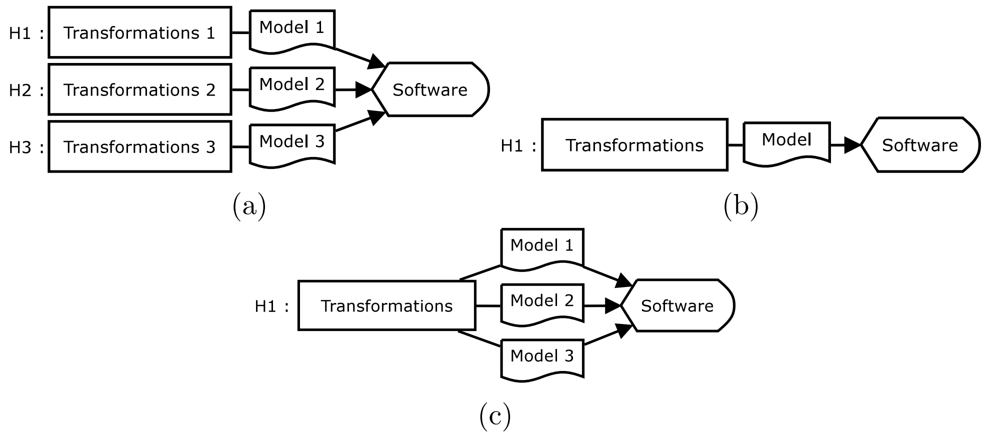


Figure 3.3: Different approaches to versioning of a large software - set of history records for each software aspect can be maintained (Fig. 3.3a) or a model considering all aspect of software can be created (Fig. 3.3b) or transformations can be interpreted for multiple models (Fig. 3.3c).

The first approach (in Fig. 3.3a) assumes there are separated metamodel and history record for each layer of software. When change is applied on a concrete layer of software then the consistency verification with all other models has to be performed as well or the layers have to be separated in a way that their change does not affect other layers. Therefore creation of such an operation-based VCS is challenging. Maintenance of such a tool would be challenging as well. An advantage of this approach is that all information about specific layers of an application is in one place and is not mixed with other information, which can improve the users's understanding of each single model and its changes. This is quite an ideal situation because the software aspects are mutually dependent. Therefore another solution has to be used in real-world scenarios.

Second approach consists of creation of a metamodel, which contains information specific for numerous software aspects. This approach assumes that an application structure can be described by one model as illustrated in Fig. 3.3b. The benefit is that all information is in one place and the consistency of only one model has to be verified when changing an application. The obvious disadvantage is a creation of one omnipotent (meta)model. Not only the creation of such a metamodel is challenging as the metamodel should be stable during the software lifecycle, but also such a metamodel is hard to maintain and share with other developers and new team members. Finally the model describing a whole software can be hard to maintain as well and such a model can quickly become confusing for larger software. The feasibility and consistency verification takes part during (or after) the execution of transformations. Adding a new software aspect into the metamodel means to re-define all existing transformations.

The last solution (illustrated in Fig. 3.3c) consists in interpretation of transformations. Instead of creating one omnipotent (meta)model, an interpretation of each transformation is defined for each layer and its model. In other words, the change described as a transformation from one software state to another is interpreted as transformations on various metamodels. The interpretations should behave as one transaction - interpretation is applied on a set of consistent models and after the last interpretation is finished the result is again consistent. Each interpretation can be considered as a module and this modularity enables to build various operation-based VCS with different capabilities and purposes. This approach combines advantages and disadvantages of both previously mentioned approaches.

Each proposed approach has its pros and its cons and its use depends on a concrete style of MDD, which is used in the model driven workflow. Next criterion is a size of the application and complexity of an application domain. Multiple models may be useful if the application domain is too complicated or if some aspect needs special care, in contrast the omnipotent model could not be a problem in case of small applications. However the approach based on interpretation of transformation is a compromise solution, thus we use this structure of an operation-based VCS for our future experiments.

3.2.3 Model Versioning

The problem of model versioning can be reduced to the versioning of text files if a textual modeling language is used [43]. However, more common approach is based on versioning of models in their native (or proprietary) form, which is usually an XML-based format.

A tool for meta-model and model co-evolution is a part of EMF [44] which uses a transformation based approach to versioning of meta-models and models. On the other hand, it fails to provide all capabilities of VCS.

An approach for model versioning is described in [45]. The solution is represented on an example of databases, which are represented as graphs. Operators and morphisms are defined to manipulate the graph representation. Each change can be propagated into relational databases or XML Schema. Comparison of models is based on model matching in this case, which can cause semantic misunderstandings of the change.

Many CASE tools such as Enterprise Architect [46] support model versioning. A problem with these solutions is that the semantics of the changes are often unclear, because the difference between two states is based on model matching. The next problem is that the solutions are often proprietary and cannot be used together with code versioning. The area of model versioning is covered by surveys on model versioning approaches [47], [48]. The last provides not only information about model versioning approaches and tools, but indicates challenges of model versioning as well. Topics such as generic VCS, fine-tuning of model comparison, accurate conflict detection, representation and resolution are introduced.

One of the crucial issues when implementing a VCS is merging of branches. An approach based on versioning of models and users collaboration is presented in [49], a formal approach for merging EMF models is described in [50] and [51] introduces model merging based on transformations in the EML language. A practical implementation of model comparison is the EMF versioning tool called EMF Compare [52].

The next important topic in VCS construction is verification of consistency. Comparing lines of text in common VCS solves the problem. Another approach has to be used for versioning models or in case of an MDD. The problems of model consistency are addressed in [53], [54] and [55].

There exists an approach to detect model inconsistencies introduced in [54], which is based on transformations as well. It detects inconsistencies in use-case or requirements models, but it can be extended to work for more kinds of UML models. The problem of structural and methodological inconsistencies is solved by a set of validations predicates.

Various approaches to creation of version control by using MDD and thirs benefits to developers' work are measured according to their productivity and performance [56]. As an example, the domain of relational databases is used in the paper. We share some ideas described in this paper such as using a set of transformations for software evolution.

3.3 Related Work Summary

There are many approaches to data evolution and versioning. These approaches vary in used language, database type, its formal definition or by its complexity. Most of them should be used by database administrators and their use by developers is complicated. There are frameworks, which share our idea of MDD approach to data evolution. However, we were not able to find a framework, which handles the code, the database schema and stored data and describes complex transformations at the same time. On the other hand, many frameworks especially at the database level are very mature.

Chapter 4

Model of Software Evolution

The Model Driven Architecture (MDA) [57] describes the MDD as a software development approach based on multiple models at various levels of abstraction. The models are created with respect to meta-models, which helps to define the concrete problem and its context. Models and abstraction levels are connected by semantic links and by transformations. Semantic links are used to represent that an element in one model represents the same entity modeled as an element in another model; transformations provide the execution logic, which can change the state of the model or create a new model. A specific transformation is code generation, when a model is transformed into a programing language, however the code can be considered to be a (textual) model. Because of various levels of abstraction we differ horizontal transformations on the same level, and vertical transformations between different levels.

The model driven development provides a good framework for automatic evolution of software [26] [58]. In this section, we describe how the MDD could help in addressing the problems connected with the data evolution. We published the formal framework presented in Sect. 4 and Sec. 6 in [59]. The framework defined in the Z language is very large. All schemas and functions important for the model are described in the following sections or are attached in appendices. However we have to shorten some declaration, therefore we provide the full version of the Z definition on-line [60].

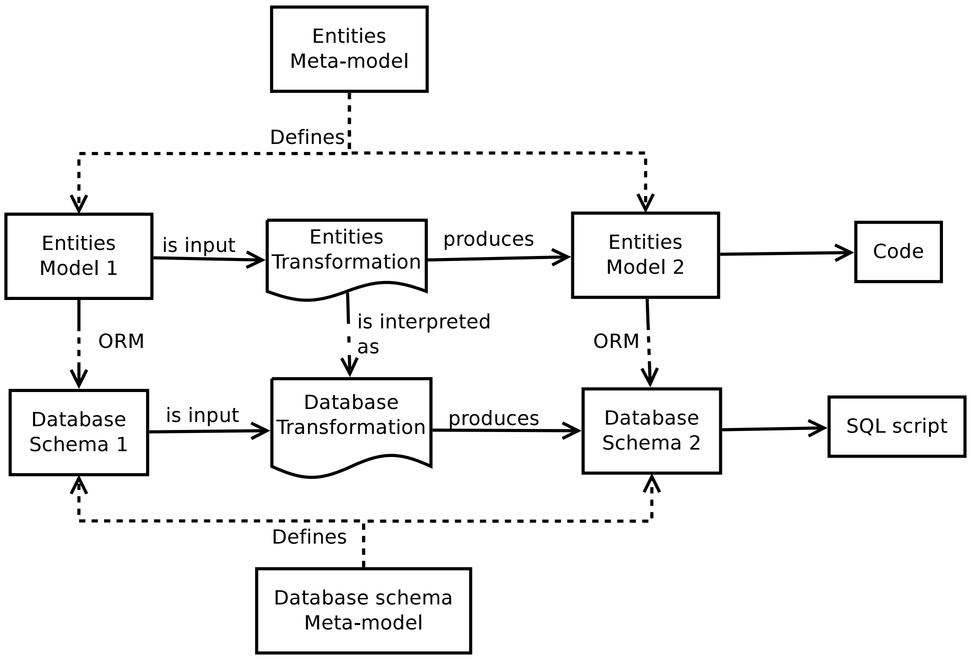


Figure 4.1: The problem of data evolution from the MDD point of view.

4.1 The Architecture of a MDD Framework for Data Evolution

The data evolution can be modeled as showed in Fig 4.1. Entities and database schema are represented as models created according to their meta-models. Some authors prefer to use only one conceptual model representing both layers of entities and database schema. Our approach uses two different models because it gives us the possibility to use various meta-models in future so we can e.g. model XML or other No-SQL databases instead of relational ones. Next advantage of separating models is that platform specific models can be used and the models can be used in more complex real-world scenarios.

The ORM can be described as a vertical transformation, which produces a database schema model according to the given model of entities. The ORM affects the way how entities and database co-evolve.

The data evolution is represented as a horizontal transformation. Each change of entities results in a change of database. Each transformation defines a semantic difference between the two states. The transformation is interpreted as an SQL script for migrating a real database.

We choose to use the transformation-based approach to evolution. Therefore there is the meta-model of the application and the set of transformations to evolve an application model (see Fig. 3.2). The model of the database is not a necessary component of the framework and there are many related projects, which use only one (conceptual) model. However, we included it in the model for following reasons: i) the database model can be used in more complex MDD scenarios e.g. another transformations may affect the model or multiple models can be used as input for database schema generation ii) the database model allows us to work on the platform specific level iii) the model of database helps us to simulate the evolution before it is applied on the real instance of the database iv) the database model provides the opportunity to simulate the behavior on stored data.

The evolution is code-first therefore they have to respect the application domain. In contrast, database migration often needs information not only about the database level but the application level as well the database migration needs information, which is not available at the application level. Therefore we need special transformations, which contain information needed for both levels. These transformations have to be interpreted on both levels - application and database. This enables us to avoid multiple definition of evolution.

All transformations and their interpretation are defined correctly so the user of the framework can rely on it (e.g. the data preservation should be assured after their execution). Therefore we introduce a formal definition for all transformations in Sec. 4.

The code of an application can be obtained directly from the model as well as the database schema. However, this is not part of our model, but it was implemented in prototypes (see Sec. 8).

We introduce the architecture of the framework for data evolution and models of its parts. The meta-models of static models are introduced and the evolutionary transformations are defined. We can define the operation-based versioning system, which uses defined transformations as the history sequence, which is interpreted for the application and for the database.

4.2 Note on Notation

The formal model is defined by using the Z notation [61] and some practices used in the presented models were inspired in [62] and [63]. We briefly introduce the meaning of Z notation's symbols used in the model in this section.

4.2.1 Types

A type can be declared by its name only:

$[TYPENAME]$

or as an enumeration of its values:

$BOOL ::= True \mid False$

4.2.2 Declaration

A variable is defined by its name and its type:

$varname : TYPENAME$

4.2.3 Schemas

The main symbol of the Z notation is the schema, which can represent a static definition of named tuple, transformation or predicate. Each schema is divided into two parts - the first part defines the variables used in the schema and their types, whereas the second part contains predicates defining a structural unit of a model or a change of the state of a model. A schema can be denoted in a named vertical form:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{SchemaName} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{declaration} \\
\hline
\mathit{predicate} \\
\hline
\end{array}$$

or in an anonymous horizontal form:

$\mid \quad [declaration \mid predicate]$

We use the vertical form to define meta-models and their elements and transformations. The horizontal form is used in the transformations's definitions as a container for sub-transformations.

4.2.4 Predicates

The predicates can be composed by logical symbols. A new line between two predicates is considered to represent '$\wedge$'.

Schemas themselves can be used as predicates, thus the schemas can be concatenated by logical operators as well.

Predicates often use following notation: $\forall\, t : \mathit{TYPE} \bullet P$ which has to be read as $\{\forall\, t \in \mathit{TYPE} \mid P\}$, where P is a predicate.

4.2.5 States

The system can be in various states. The representation of one variable in two states is denoted by the apostrophe symbol. A variable without apostrophe is considered to be defined in the initial state, whereas variable with the same name, which is decorated by the apostrophe is the same variable in the final state. The same can be used for schemas. The schema, which defines the difference between two states is called transformation.

Delta Notation

To shortcut the notation of declaration in case we define change between states we use 'δ' and 'Ξ' notations.

To note the schema changed during the transformation, we use the symbol 'δ', which is defined as:

$$\begin{array}{|l}
\hline
\Delta[X] \\
X \\
X' \\
\hline
\end{array}$$

Xi Notation

If the change of a variable in the schema is not defined explicitly, we considered the value of the variable was not changed by the transformation. To note the whole schema is not affected by the transformation we use the symbol 'Ξ':

$$\begin{array}{|l}
\hline
\Xi[X] \\
X \\
X' \\
\hline
X' = X \\
\hline
\end{array}$$

If the X represents a whole schema not just a simple variable, then the inner declarations can be accessed directly by theirs names (e.g. $X.varname$). The whole schema can be addressed by using the θ symbol.

Preconditions of a Transformation

The function *pre* [62] is defined to obtain all predicates describing the initial state of a transformation and the function *decl* can be used to obtain all declarations of a schema:

$$\begin{array}{|l}
pre : SCHEMA \rightarrow \mathbb{P}\, PREDICATE \\
decl : SCHEMA \rightarrow \mathbb{P}\, DECLARATION
\end{array}$$

4.2.6 Axioms

Axioms, which defines features of a model, and functions, which are used to query models, are defined in Z by using so-called axiomatic definitions:

$$\begin{array}{|l}
declaration \\
\hline
predicates
\end{array}$$

4.2.7 Name Conventions

We decide to use different name conventions in the model:

- Schemas, which represents elements of (meta-)models, have names in upper case letters (e.g. *CLASS*).
- Schemas, which represents transformations are in lower camel case (e.g. *initClass*) as well as functions' names.
- A variable name which ends with the '?' symbol denotes an input variable of a schema (transformation) - e.g. $inputParameter?$.
- Variables, which end with the '!' symbol denoted an output variable of a schema (transformation) e.g. $outputParameter!$.
- Variables without a special symbol are considered to be local variables.

Chapter 5

Meta-Models of Entites and Database

The model of application has only one layer - entities, whereas the model of database models two database concepts - database schema and data. Although the database evolution and data migration could be modeled using one conceptual model we decide to use two different models. It is because we like to emphasize that we focus on application evolution and its propagation into database. Next reason for two models is that separate models could be extended or adapted to specific conditions easier than one model used for both concepts. Of course our models are limited in contrast with real-world systems, but they illustrate the main problems of entities and database co-evolution.

In all models we assume there is a set of labels which serves as identifiers of the elements in the model:

$[LABEL]$

5.1 Meta-Model of Entities

The application model models a simple structure of classes, their attributes and associations between classes. The meta-model is in Fig. 5.1 and all meta-models' elements are defined in the following sections.

5.1.1 Cardinality

The cardinality is used to specify the associations between classes. It is common to use positive natural numbers to set the cardinality of an association, however for sake of

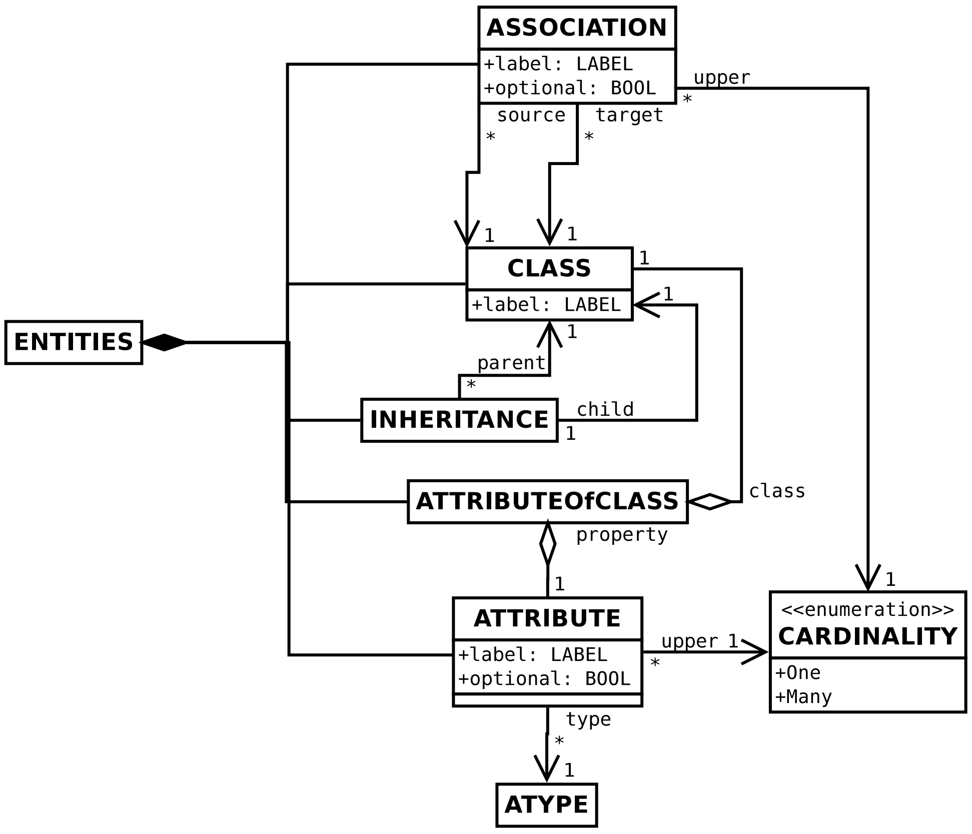


Figure 5.1: The meta-model of the layer of entities.

model simplicity we decide to use only one-to-one and one-to-many associations. Thus the cardinality is defined as follows:

$CARDINALITY ::= One \mid Many$

5.1.2 Types in Application

Application type (*ATYPE*) represents primitive types in the application. There are typically defined types such as String, Integer, Boolean etc. in a typed programming language. Type casting is not part of transformations defined in this thesis, because we focus on structural changes and their impact on data in the first place. Therefore we choose to define only one universal type in an application. However, types and type casting can be integrated into the described model and transformations. Each type is a member of the set:

$[ATYPE]$

Although we are not interested in type castings, we decide to make an attribute type part of our model for two reasons. First reason is it the difference between attributes of primitive types and from attributes of types based on other modeled classes, which are in our model represented as associations. Second reason is it helps us to keep in mind that application and database types are not the same.

5.1.3 Class

Class represents a basic organizational unit in the application model.

CLASS
label : *LABEL*

All classes creates a domain with the bottom defined as:

NULLCLASS : *CLASS*

5.1.4 Attribute

Attribute represents a feature of a class which is represented as a primitive type. An attribute can be optional and according to its cardinality, it can represent a single value or a collection of values. The label identifies the attribute in the context of the owning class.

ATTRIBUTE
optional : *BOOL*
upper : *CARDINALITY*
type : *ATYPE*
label : *LABEL*

The relation between an attribute and a class is represented by the *ATTRIBUTEOf-CLASS* schema. The class in the relation with an attribute is called owning class of the attribute.

ATTRIBUTEOfCLASS
class : *CLASS*
attribute : *ATTRIBUTE*

5.1.5 Association

Association represents a connection between two classes. The association is unidirectional - each transformation has a source class and its target class. The source class of the association is considered to be the owning class of the association. The associations allow to model relationships of cardinality one to one and one to many.

ASSOCIATION

$label : LABEL$
$upper : CARDINALITY$
$optional : BOOL$
$source : CLASS$
$target : CLASS$

A class in the application can be members of an inheritance hierarchy.

INHERITANCE

$parent : CLASS$
$child : CLASS$

5.1.6 Layer of Entities

An application is described as a set of classes, attributes, associations and its relationships. It creates the context for all structures used in the software persistent layer.

ENTITIES

$classes : \mathbb{P}\, CLASS$
$attributes : \mathbb{P}\, ATTRIBUTE$
$associations : \mathbb{P}\, ASSOCIATION$
$attributesOfClasses : \mathbb{P}\, ATTRIBUTEOfCLASS$
$inheritance : \mathbb{P}\, INHERITANCE$

A special kind of *ENTITIES* is used to denote an inconsistent layer of entities:

$ERRENTITIES : ENTITIES$

The *ERRENTITIES* is a bottom of the domain of all *ENTITIES*.

The transformations, which are capable to change the structure of entities are defined in Sec. A.2. Set of functions, which query the layer of entities and its parts is defined in appendix A.1.

5.1.7 Invariants Constraining Entites

The model itself provides only definitions of structural elements of the model. These definitions are extended by a set of invariants, which define essential features of models.

Label Uniqueness

The labels identify elements in the model, therefore all class' labels has to be unique in the model:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ c_1, c_2 : \mathit{CLASS} \bullet \\ \quad c_1 \in e.\mathit{classes} \land c_2 \in e.\mathit{classes} \land c_1.\mathit{label} = c_2.\mathit{label} \Rightarrow c_1 = c_2 \end{array}$$

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ \mathit{poc}_1, \mathit{poc}_2 : \mathit{ATTRIBUTEOfCLASS} \bullet \\ \quad \mathit{poc}_1 \in e.\mathit{attributesOfClasses} \land \mathit{poc}_2 \in e.\mathit{attributesOfClasses} \land \\ \quad \mathit{poc}_1.\mathit{class} = \mathit{poc}_2.\mathit{class} \land \mathit{poc}_1 \neq \mathit{poc}_2 \Rightarrow \\ \qquad (\mathit{poc}_1.\mathit{attribute}).\mathit{label} \neq (\mathit{poc}_2.\mathit{attribute}).\mathit{label} \end{array}$$

Labels of associations are unique not in context of the whole entities model, but in context of a single class only.

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ a_1, a_2 : \mathit{ASSOCIATION} \bullet \\ \quad a_1 \in e.\mathit{associations} \land a_2 \in e.\mathit{associations} \land \\ \quad a_1.\mathit{label} = a_2.\mathit{label} \Rightarrow a_1 = a_2 \lor a_1.\mathit{source} \neq a_2.\mathit{source} \end{array}$$

Label Uniqueness within Inheritance Hierarchy

The names of attributes and associations have to be unique not only within one class, but in the context of all parent classes in the hierarchy as well:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ c_1, c_2 : \mathit{CLASS};\ \mathit{par} : \mathbb{P}\,\mathit{CLASS};\ p_1 : \mathit{ATTRIBUTE} \bullet \\ \quad \mathit{par} = \mathit{parentOf}(c_1, e)^* \land c_2 \in \mathit{par} \land c_1 \in e.\mathit{classes} \Rightarrow \\ \qquad p_1 \in \mathit{attributesOf}(c_1, e) \land p_1 \notin \mathit{attributesOf}(c_2, e) \end{array}$$

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ c_1, c_2 : \mathit{CLASS};\ \mathit{par} : \mathbb{P}\,\mathit{CLASS};\ a_1 : \mathit{ASSOCIATION} \bullet \\ \quad \mathit{par} = \mathit{parentOf}(c_1, e)^* \wedge c_2 \in \mathit{par} \wedge c_1 \in e.\mathit{classes} \Rightarrow \\ \qquad a_1 \in \mathit{associationsOf}(c_1, e) \wedge a_1 \notin \mathit{associationsOf}(c_2, e) \end{array}$$

Attributes Are Owned by Classes

If there is attribute in the entities layer, then it has to be owned by a class. And each class which is in a relation to an attribute has to be member of the entities' classes:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ p : \mathit{ATTRIBUTE} \bullet \\ \quad p \in e.\mathit{attributes} \Leftrightarrow \exists\, \mathit{poc} : \mathit{ATTRIBUTEOfCLASS} \bullet \\ \qquad \mathit{poc} \in e.\mathit{attributesOfClasses} \wedge p = \mathit{poc}.\mathit{attribute} \\ \forall\, e : \mathit{ENTITIES};\ c : \mathit{CLASS};\ \mathit{poc} : \mathit{ATTRIBUTEOfCLASS} \bullet \\ \quad c = \mathit{poc}.\mathit{class} \Rightarrow c \in e.\mathit{classes} \end{array}$$

Associations Are between Entities' Classes

If there is an association in the entities layer then both source and target classes have to be in the entities' classes:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ a : \mathit{ASSOCIATION} \bullet \\ \quad a \in e.\mathit{associations} \Rightarrow \exists\, c_s, c_t : \mathit{CLASS} \bullet \\ \qquad c_s = a.\mathit{source} \wedge c_t = a.\mathit{target} \wedge c_s \in e.\mathit{classes} \wedge c_t \in e.\mathit{classes} \end{array}$$

Inheritance between Entities' Classes Only

The inheritance relationship can be defined only between classes which are part of the entities' classes:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ i : \mathit{INHERITANCE} \bullet \\ \quad i \in e.\mathit{inheritance} \Rightarrow \exists\, c_p, c_c : \mathit{CLASS} \bullet \\ \qquad c_p \in e.\mathit{classes} \wedge c_c \in e.\mathit{classes} \wedge c_p = i.\mathit{parent} \wedge c_c = i.\mathit{child} \end{array}$$

Only One Parent of Class

There is at most one parent for a class:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ i_1, i_2 : \mathit{INHERITANCE} \bullet \\ \quad i_1 \in e.\mathit{inheritance} \wedge i_2 \in e.\mathit{inheritance} \wedge i_1.\mathit{child} = i_2.\mathit{child} \\ \quad \wedge\, i_1.\mathit{parent} = i_2.\mathit{parent} \Rightarrow i_1 = i_2 \end{array}$$

No Cyclical Inheritance

The inheritance cannot be cyclical:

$$\begin{array}{l} \forall\, e : \mathit{ENTITIES};\ c : \mathit{CLASS} \bullet \\ \quad c \in e.\mathit{classes} \Rightarrow \mathit{isInheritanceCyclical}(c, e) = \mathit{False} \end{array}$$

5.1.8 Consistency of Entities

The invariants create a type system of entities layer. If all invariants are fulfilled, then a model is consistent otherwise we consider the model to be *ERRENTITIES*. The consistency or inconsistency of the entities' layer is important in order to the evolutionary transformations. An evolutionary transformation cannot be applied on an inconsistent layer of entities.

5.1.9 Transformations for Entities Manipulation

The transformations, which are able to change the layer of entities are used during the software evolution. The transformation we defined are introduced in this section informally and their formal definitions of selected transformations is in appendix A.2. The selected set does not represent all transformations for the entities' layer nor the minimal set of transformations. Selected are the transformation used during the evolution of the software as defined in Sec. 6. The transformation has name with suffix "-EL".

- **addEntityEL** The transformation adds a new entity into the entities' layer.
- **removeEntityEL** Removes an entity from the entities' layer.
- **addAttributeEL** Adds the given attribute into the given class.
- **removeAttributeEL** Removes the given attribute from the given class.
- **addAssociationEL** Adds a new association between two classes into the entities' layer.
- **removeAssociationEL** Removes an association between two classes in the entities' layer.
- **addEntityParentEL** Creates an inheritance (child – parent) relationship between two classes.

- **removeEntityParentEL** Destroys an inheritance (child – parent) relationship between two classes.
- **pushAttributeDownEL** Moves the selected attribute from parent to all its child classes.
- **pullAttributeUpEL** Moves the selected attribute from child to its parent.

5.2 Meta-Model of Database

A database consists of two parts - of a database schema, which defines the structure, and stored data. The database model uses terminology from the domain of relational databases. We use the relational database as it is the commonly used in software applications, whereas object or XML databases are not so widely used. To fit other domains the model has to be changed or we can use mapping between various database types [64] [65].

The meta-model is designed as the platform independent model. Therefore we try to avoid any elements specific for a concrete database.

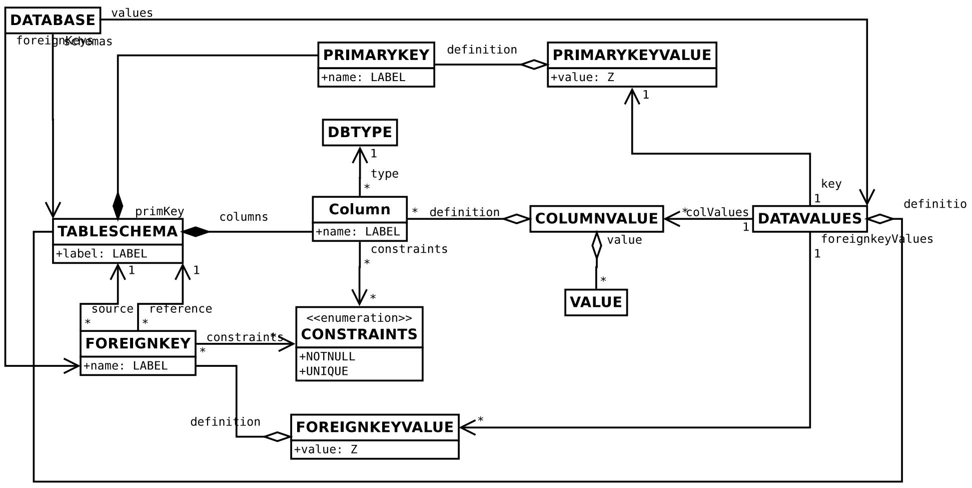


Figure 5.2: The meta-model of the database.

5.2.1 Data Types

Similar as in case of application there is only one type in the database, however more types can be added:

$[DTYPE]$

5.2.2 Values

A database consists not only of a schema but also of data which are represented as rows in a table. The concrete values are not important for our purposes. Therefore a value is defined only as a member of the set:

$$[\mathit{VALUE}]$$

5.2.3 Constraints

There are two types of constraints defined in the model. Both constraints are column constraints - first constraint *NOTNULL* defines non-empty columns. Second constraint *UNIQUE* defines there has to be unique records in a column or foreign key.

$$\mathit{CONSTRAINT} ::= \mathit{NOTNULL} \mid \mathit{UNIQUE}$$

5.2.4 Column

Column defines data values and types which can be a part of a table record. Each column can contain one value.

```
COLUMN
constraints : ℙ CONSTRAINT
type : DTYPE
label : LABEL
```

The column value is represented as:

```
COLUMNVALUE
definition : COLUMN
value : VALUE
```

5.2.5 Primary key

Primary key is unambiguous identifier of a record in a table. The primary key is always provided (automatically generated) by the database as a non-zero natural number. Primary key is always defined with constraints *NOTNULL* and *UNIQUE*.

PRIMARYKEY
$name : LABEL$

The value of a primary key:

PRIMARYKEYVALUE
$definition : PRIMARYKEY$
$value : \mathbb{Z}$

5.2.6 Table Schema

Table is a basic concept of the relational database schema. It has a name, one or more columns and it can be related to other tables in the schema by foreign keys.

TABLESCHEMA
$label : LABEL$
$primKey : PRIMARYKEY$
$columns : \mathbb{P}\ COLUMN$

5.2.7 Foreign key

Foreign key is a reference to another table's primary key. It has a unique name and it can be constrained.

FOREIGNKEY
$label : LABEL$
$constraints : \mathbb{P}\ CONSTRAINT$
$source : TABLESCHEMA$
$reference : TABLESCHEMA$

The value of a foreign key is 0 if there is no reference or a non-zero natural number.

FOREIGNKEYVALUE
$definition : FOREIGNKEY$
$value : \mathbb{Z}$

5.2.8 Data Values

Rows in the table represent stored data. The values are stored as:

$$\begin{array}{|l}
\textit{DATAVALUES} \\ \hline
\textit{definition} : \textit{TABLESCHEMA} \\
\textit{key} : \textit{PRIMARYKEYVALUE} \\
\textit{colValues} : \mathbb{P}\,\textit{COLUMNVALUE} \\
\textit{foreignkeyValues} : \mathbb{P}\,\textit{FOREIGNKEYVALUE} \\ \hline
\end{array}$$

All data values creates a domain with the bottom, defined as:

$$\mid \textit{NULLDATAVALUE} : \textit{DATAVALUES}$$

5.2.9 Sequence

The sequence provides a value for the primary key if a new data value is add into a table.

$$\begin{array}{|l}
\textit{SEQUENCE} \\ \hline
\textit{current} : \mathbb{Z} \\ \hline
\end{array}$$

5.2.10 Database

A database consists of database schemas and its values.

$$\begin{array}{|l}
\textit{DATABASE} \\ \hline
\textit{schemas} : \mathbb{P}\,\textit{TABLESCHEMA} \\
\textit{foreignKeys} : \mathbb{P}\,\textit{FOREIGNKEY} \\
\textit{values} : \mathbb{P}\,\textit{DATAVALUES} \\
\textit{sequence} : \textit{SEQUENCE} \\ \hline
\end{array}$$

A special kind of database is *ERRDATABASE*, which represents inconsistent database:

$$\mid \textit{ERRDATABASE} : \textit{DATABASE}$$

The transformations, which are capable to change the structure of database schema and data are defined in Sec. A.4. Set of functions, which query the layer of entities and its parts is defined in appendix A.3.

5.2.11 Database Invariants

The database model is constrained by a set of invariants as is the entities model.

Label Uniqueness

Labels in the database are unique in certain context - *TABLESCHEMA*s' labels are unique in context of the whole database, *COLUMN*s' labels are unique in the context of owning *TABLESCHEMA* etc.

$$\begin{array}{|l}
\forall\, ts_1, ts_2 : TABLESCHEMA;\ d : DATABASE \bullet \\
\quad ts_1.label = ts_2.label \land ts_1 \in d.schemas \land ts_2 \in d.schemas \Rightarrow \quad ts_1 = ts_2 \\
\forall\, col_1, col_2 : COLUMN;\ ts : TABLESCHEMA \bullet \\
\quad col_1.label = col_2.label \land col_1 \in ts.columns \land col_2 \in ts.columns \Rightarrow col_1 = col_2 \\
\forall\, fk_1, fk_2 : FOREIGNKEY;\ d : DATABASE \bullet \\
\quad fk_1.label = fk_2.label \land fk_1 \in d.foreignKeys \land fk_2 \in d.foreignKeys \Rightarrow fk_1 = fk_2
\end{array}$$

Unique Constraint Invariant

The invariant assures that the values constrained by the *UNIQUE* constraint are unique within the database's data.

$$\begin{array}{|l}
\forall\, cv_1, cv_2 : COLUMNVALUE;\ cd : COLUMN \bullet \\
\quad cv_1 \neq cv_2 \land cv_1.definition = cd \land cv_2.definition = cd \land \\
\quad UNIQUE \in cd.constraints \Rightarrow cv_1.value \neq cv_2.value
\end{array}$$

Notnull Constraint Invariant

The *NOTNULL* constraint assures that there is a *COLUMNVALUE* for each constrained *COLUMN* in the *TABLESCHEMA*:

$$\begin{array}{|l}
\forall\, d : DATABASE;\ ts : TABLESCHEMA;\ col : COLUMN;\ td : DATAVALUES \bullet \\
\quad ts \in d.schemas \land \\
\quad col \in ts.columns \land \\
\quad td.definition = ts \land \\
\quad td \in d.values \land \\
\quad NOTNULL \in col.constraints \Rightarrow \\
\qquad \exists\, cv : COLUMNVALUE \bullet cv \in td.colValues
\end{array}$$

Foreign Keys Reference Tables in Database

The *TABLESCHEMA* which is referenced by a foreign key has to be part of the database.

$$\begin{array}{l}
\forall\, d : \mathit{DATABASE};\ \mathit{fk} : \mathit{FOREIGNKEY} \bullet \\
\quad \mathit{fk} \in d.\mathit{foreignKeys} \Leftrightarrow \\
\quad \exists\, ds_s, ds_t : \mathit{TABLESCHEMA} \bullet \\
\qquad ds_s = \mathit{fk}.\mathit{source} \land \\
\qquad ds_t = \mathit{fk}.\mathit{reference} \land \\
\qquad ds_s \in d.\mathit{schemas} \land \\
\qquad ds_t \in d.\mathit{schemas}
\end{array}$$

Foreign Key References a Primary Key Value

The foreign key value can reference only a value of a primary key in our model.

$$\begin{array}{l}
\forall\, \mathit{fv} : \mathit{FOREIGNKEYVALUE};\ d : \mathit{DATABASE};\ dv : \mathit{DATAVALUES} \bullet \\
\quad \mathit{fv} \in dv.\mathit{foreignkeyValues} \land dv \in d.\mathit{values} \Rightarrow \\
\quad \exists\, dv_2 : \mathit{DATAVALUES} \bullet \\
\qquad dv_2.\mathit{key}.\mathit{value} = \mathit{fv}.\mathit{value} \land dv_2.\mathit{definition} = \mathit{fv}.\mathit{definition}.\mathit{reference}
\end{array}$$

Data Exists Only for Columns in Database

Each value which exists in the *DATAVALUES* has to refer to a member of *TABLESCHEMA* existing within the *DATABASE*.

$$\begin{array}{l}
\forall\, d : \mathit{DATABASE};\ dv : \mathit{DATAVALUES};\ cv : \mathit{COLUMNVALUE}; \\
\mathit{fk} : \mathit{FOREIGNKEYVALUE} \bullet \\
\quad (dv \in d.\mathit{values} \Leftrightarrow dv.\mathit{definition} \in d.\mathit{schemas}) \land \\
\quad (cv \in dv.\mathit{colValues} \Leftrightarrow cv.\mathit{definition} \in dv.\mathit{definition}.\mathit{columns}) \land \\
\quad (\mathit{fk} \in dv.\mathit{foreignkeyValues} \Leftrightarrow \mathit{fk}.\mathit{definition} \in d.\mathit{foreignKeys})
\end{array}$$

Primary Key Values Uniqueness

The primary key value is an unique identifier of a *DATAVALUE*.

$$\begin{array}{l}
\forall\, dv_1, dv_2 : \mathit{DATAVALUES};\ d : \mathit{DATABASE} \bullet \\
\quad dv_1 \in d.\mathit{values} \land dv_2 \in d.\mathit{values} \land dv_1.\mathit{key} = dv_2.\mathit{key} \land \\
\quad dv_1.\mathit{definition} = dv_2.\mathit{definition} \Rightarrow dv_1 = dv_2
\end{array}$$

5.2.12 Database Consistency

The invariants create a type system of the database layer. If all invariants are fulfilled, then a model is consistent otherwise we consider the model to be *ERRDATABASE*. The consistency or inconsistency of the database's layer is essential in order to the evolutionary transformations. An evolutionary transformation cannot be applied on an inconsistent database layer.

5.2.13 Transformations for Database Manipulation

The transformations for database manipulation are introduced informally in this section. The formal definitions of mentioned transformations is in appendix A.4. In contrast with the transformation for entities manipulation the database transformations are more complex because they handle both database schema and stored data. The selected set does not represent all transformations for the database layer nor the minimal set of transformations as in case of the transformation for entities' layer. The transformations for database manipulation has the suffix "-DB".

- **addTableDB** Adds a table schema into the database schema. The transformation has no impact on stored data.
- **dropTableDB** Removes a table from the database and removes all data stored in the given table.
- **dropEmptyTableDB** Removes a table schema from the database only if there are no stored data.
- **addColumnDB** Adds a column into the table schema in the database.
- **dropColumnDB** Removes a column from the table schema as well as all data stored in the column.
- **dropEmptyColumnDB** Removes a column from the table schema only if there are no data stored in the given column.
- **addForeignKeyDB** Adds a foreign key into the given table.
- **dropForeignKeyDB** Removes a foreign key from the table schema.
- **dropEmptyForeignKeyDB** Removes a foreign key from the table schema only if there are no data stored.
- **changeForeignKeyReference** Changes the table referenced by the given foreign key, according to the given mapping.

- **copyColumnDB** The transformations copies structure of the column from one table schema to another. The data are copied from the source to the target table according to the given mapping.
- **copyTableStructureDB** The transformations creates a copy of the given table. The new table (copy) has a new name defined by the given label. The data are copied as well.

5.3 Mapping between Data

A relation between data from different *DATAVALUES* needs to be known during execution of some transformations (e.g. *moveAttribute*). The relation is defined as a mapping between *DATAVALUES*. The mapping is defined using *MAPPINGPAIR* as follows:

$$
\begin{array}{|l}
\hline
\quad \mathit{MAPPINGPAIR} \\
\mathit{source} : \mathit{DATAVALUES} \\
\mathit{target} : \mathit{DATAVALUES} \\
\hline
\mathit{source} \neq \mathit{NULLDATAVALUE} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\quad \mathit{MAPPING} \\
\mathit{pairs} : \mathbb{P}\, \mathit{MAPPINGPAIR} \\
\hline
\forall\, p_1, p_2 : \mathit{MAPPINGPAIR} \bullet \\
\quad p_1.\mathit{source}.\mathit{definition} = p_2.\mathit{source}.\mathit{definition} \land \\
\quad p_2.\mathit{target}.\mathit{definition} = p_1.\mathit{target}.\mathit{definition} \\
\hline
\end{array}
$$

The definition of mapping provides an opportunity to define one-to-many and many-to-many relations between data. If there is no mapping (*target*) for a given source the bottom of the *DATAVALUES* domain can be used. However, such mapping can cause data lost between states. This issue is addressed in detail in the context of each transformation.

A special case of mapping is an empty mapping denoted as m_e, which is used when there are no *DATAVALUES* in the domain and in the range i.e. the transformation takes part on the structural level only.

5.3.1 Inverse Mapping

The inverse mapping is defined as follows:

$$
\begin{array}{|l}
\mathit{inverse} : \mathit{MAPPING} \rightarrow \mathit{MAPPING} \\
\hline
\forall\, m, m^i : \mathit{MAPPING} \bullet \\
\quad \mathit{inverse}(m) = m^i \Leftrightarrow \\
\quad\quad (\forall\, p : \mathit{MAPPINGPAIR} \bullet p \in m.\mathit{pairs} \Rightarrow \\
\quad\quad\quad \exists\, p^i : \mathit{MAPPINGPAIR} \bullet \\
\quad\quad\quad\quad p^i \in m^i.\mathit{pairs} \wedge p.\mathit{source} = p^i.\mathit{target} \wedge p.\mathit{target} = p^i.\mathit{source}) \wedge \\
\quad\quad (\forall\, p^i : \mathit{MAPPINGPAIR} \bullet p^i \in m^i.\mathit{pairs} \Rightarrow \\
\quad\quad\quad \exists\, p : \mathit{MAPPINGPAIR} \bullet \\
\quad\quad\quad\quad p \in m.\mathit{pairs} \wedge p.\mathit{source} = p^i.\mathit{target} \wedge p.\mathit{target} = p^i.\mathit{source})
\end{array}
$$

5.3.2 Mapping Invariants

The mapping is defined in the context of existing data only i.e. no new data can be added into the database by mapping:

$$
\begin{array}{|l}
\hline
\forall\, d : \mathit{DATABASE};\ m : \mathit{MAPPING};\ p : \mathit{MAPPINGPAIR} \bullet \\
\quad p \in m.\mathit{pairs} \Rightarrow p.\mathit{source}.\mathit{definition} \in d.\mathit{schemas}
\end{array}
$$

Each mapping has to fulfill constraints given by the structural definition of its range *DATAVALUES*. Concretely uniqueness of column values:

$$
\begin{array}{|l}
\hline
\forall\, m : \mathit{MAPPING};\ p_1, p_2 : \mathit{MAPPINGPAIR};\ x_1, x_2, y_1, y_2 : \mathit{DATAVALUES}; \\
c_1, c_2 : \mathit{COLUMNVALUE} \bullet \\
\quad p_1 \in m.\mathit{pairs} \wedge p_2 \in m.\mathit{pairs} \wedge x_1 = p_1.\mathit{source} \wedge \\
\quad x_1 = p_1.\mathit{source} \wedge x_2 = p_2.\mathit{source} \wedge \\
\quad y_1 = p_1.\mathit{target} \wedge y_2 = p_2.\mathit{target} \wedge c_1 \in y_1.\mathit{colValues} \wedge \\
\quad c_2 \in y_2.\mathit{colValues} \wedge c_1.\mathit{definition} = c_2.\mathit{definition} \wedge \\
\quad \mathit{NOTNULL} \in c_1.\mathit{definition}.\mathit{constraints} \Rightarrow c_1.\mathit{value} \neq c_2.\mathit{value}
\end{array}
$$

if this principle is violated then use of such mapping leads to an inconsistent database. Similarly if there are *COLUMNVALUES* or *FOREIGNKEYVALUES* which violate the *NOTNULL* constraint then the result of a transformation can be inconsistent. Therefore another invariant for the mapping is defined:

$$
\begin{array}{l}
\forall m : MAPPING;\ p : MAPPINGPAIR;\ c : COLUMN \bullet \\
\quad (p \in m.pairs \land c \in p.source.definition.columns \land NOTNULL \in c.constraints \Rightarrow \\
\quad\quad \exists cv : COLUMNVALUE \bullet cv \in p.source.colValues \land cv.definition = c) \lor \\
\quad (c \in p.target.definition.columns \land NOTNULL \in c.constraints \Rightarrow \\
\quad\quad \exists cv : COLUMNVALUE \bullet cv \in p.target.colValues \land cv.definition = c) \\
\forall m : MAPPING;\ p : MAPPINGPAIR;\ f : FOREIGNKEY \bullet \\
\quad (p \in m.pairs \land f.source = p.source.definition \land NOTNULL \in f.constraints \Rightarrow \\
\quad\quad \exists fv : FOREIGNKEYVALUE \bullet fv \in p.source.foreignkeyValues \land \\
\quad\quad fv.definition = f) \lor \\
\quad (p \in m.pairs \land f.source = p.target.definition \land NOTNULL \in f.constraints \Rightarrow \\
\quad\quad \exists fv : FOREIGNKEYVALUE \bullet fv \in p.target.foreignkeyValues \land \\
\quad\quad fv.definition = f)
\end{array}
$$

5.3.3 Mapping Features

The mapping has crucial impact on feasibility of several transformations or data preservation during their execution, therefore we define two features of mapping - completeness and duplicity. This features helps us to describe easier the transformations.

Mapping Completeness

The completeness of mapping has crucial impact on data preservation in some transformations. We define three kinds on mapping completeness:

1. **Source completeness** A mapping is source complete, if there is a mapping pair for each instance of the source class. The instances of the target class can occur in the mapping once or multiple times or they are not part of the mapping.

$$
\begin{array}{l}
\textit{MappingSourceComplete} \\
\hline
\Xi DATABASE \\
map? : MAPPING \\
m : MAPPINGPAIR \\
\hline
m \in map?.pairs \\
\forall ds : DATAVALUES \bullet \\
\quad ds \in values \land ds.definition = m.source.definition \Leftrightarrow \\
\quad \exists mp : MAPPINGPAIR \bullet mp.source = ds \land mp \in map?.pairs \\
\hline
\end{array}
$$

2. **Target completeness** Is similar to the source completeness. A mapping is target complete, if there is a mapping pair for each instance of the target class.

MappingTargetComplete

$$
\begin{array}{l}
\Xi DATABASE \\
map? : MAPPING \\
m : MAPPINGPAIR \\
\hline
m \in map?.pairs \\
\forall ds : DATAVALUES \bullet \\
\quad ds \in values \land ds.definition = m.target.definition \Leftrightarrow \\
\quad \exists mp : MAPPINGPAIR \bullet mp.target = ds \land mp \in map?.pairs
\end{array}
$$

3. **Full completeness** A mapping is considered to be full complete, if there is a mapping pair for each instance of the source and each instance of the target.

MappingFullComplete

$$
\begin{array}{l}
\Xi DATABASE \\
map? : MAPPING \\
m : MAPPINGPAIR \\
\hline
MappingSourceComplete \\
MappingTargetComplete
\end{array}
$$

Duplicities in Mapping

The values in mapping has to follow the *MappingRespectsUniquenessInv* invariant. Nevertheless, in case there is no constraint on uniqueness than there can be more than one occurrence of the same instance in the mapping. We define four kinds of mapping based on the duplicities:

1. **Simple mapping** A mapping is called simple if there is no duplicate instance of target or source class.

MappingIsSimple

$$
\begin{array}{l}
\Xi DATABASE \\
map? : MAPPING \\
\hline
\forall dvs, dvt : DATAVALUES; \ m, m_2 : MAPPINGPAIR \bullet \\
\quad dvs \in values \land dvt \in values \land m \in map?.pairs \land m_2 \in map?.pairs \Rightarrow \\
\qquad (dvs = m.source \land dvs = m_2.source \Rightarrow m = m_2) \land \\
\qquad (dvt = m.target \land dvt = m_2.target \Rightarrow m = m_2)
\end{array}
$$

2. **Mapping with source duplicates** A mapping with source duplicates contains duplicates of source class instances, but it contains no duplicates of target class instances i.e. it represents the many-to-one relationship.

$$
\begin{array}{|l}
\textit{MappinNoTargetDuplicates} \\
\Xi \mathit{DATABASE} \\
\mathit{map?} : \mathit{MAPPING} \\
\hline
\exists\, \mathit{dvs} : \mathit{DATAVALUES};\ m, m_2 : \mathit{MAPPINGPAIR} \bullet \\
\quad \mathit{dvs} \in \mathit{values} \land m \in \mathit{map?.pairs} \land m_2 \in \mathit{map?.pairs} \land \\
\quad \mathit{dvs} = m.\mathit{source} \land \mathit{dvs} = m_2.\mathit{source} \land m \neq m_2 \\
\forall\, \mathit{dvt} : \mathit{DATAVALUES};\ m, m_2 : \mathit{MAPPINGPAIR} \bullet \\
\quad \mathit{dvt} \in \mathit{values} \land \mathit{dvt} \in \mathit{values} \land m \in \mathit{map?.pairs} \land m_2 \in \mathit{map?.pairs} \land \\
\quad \mathit{dvt} = m.\mathit{target} \land \mathit{dvt} = m_2.\mathit{target} \Rightarrow m = m_2 \\
\hline
\end{array}
$$

3. **Mapping with target duplicates** The mapping with target duplicates is similar to the previous case, but it represents one-to-many relationship. A mapping with target duplicates contains duplicates of target class instances, but it contains no duplicates of source class instances.

$$
\begin{array}{|l}
\textit{MappinNoSourceDuplicates} \\
\Xi \mathit{DATABASE} \\
\mathit{map?} : \mathit{MAPPING} \\
\hline
\exists\, \mathit{dvt} : \mathit{DATAVALUES};\ m, m_2 : \mathit{MAPPINGPAIR} \bullet \\
\quad \mathit{dvt} \in \mathit{values} \land m \in \mathit{map?.pairs} \land m_2 \in \mathit{map?.pairs} \land \\
\quad \mathit{dvt} = m.\mathit{target} \land \mathit{dvt} = m_2.\mathit{target} \land m \neq m_2 \\
\forall\, \mathit{dvs} : \mathit{DATAVALUES};\ m, m_2 : \mathit{MAPPINGPAIR} \bullet \\
\quad \mathit{dvs} \in \mathit{values} \land m \in \mathit{map?.pairs} \land \\
\quad \mathit{dvs} = m.\mathit{source} \land \mathit{dvs} = m_2.\mathit{source} \Rightarrow m = m_2 \\
\hline
\end{array}
$$

4. **Mapping with duplicates** Mapping with duplicates is the most general case of mapping. It can contain duplicate of both source and target instances i.e. it represents the many-to-many relationship.

MappinWithDuplicates

$\Xi DATABASE$

$map? : MAPPING$

$\exists\, dvt : DATAVALUES;\ m, m_2 : MAPPINGPAIR \bullet$
$\quad dvt \in values \wedge m \in map?.pairs \wedge m_2 \in map?.pairs \wedge$
$\quad dvt = m.target \wedge dvt = m_2.target \wedge m \neq m_2$

$\exists\, dvs : DATAVALUES;\ m, m_2 : MAPPINGPAIR \bullet$
$\quad dvs \in values \wedge m \in map?.pairs \wedge m_2 \in map?.pairs \wedge$
$\quad dvs = m.source \wedge dvs = m_2.source \wedge m \neq m_2$

5.4 Entites - Database Mapping

Whereas entities and database can evolve, the entities - data mapping i.e. object-relational mapping (ORM) is fixed during the software lifecycle. Its change causes that the transformations defined in this thesis have to be redefined or extended in the way they fit the new ORM. The mapping used in this thesis is similar to the Hibernate mapping [6], thus a lot of developers should be familiar with it. The main ideas are:

- Classes are mapped to tables.
- Attribute representing a single value is mapped to a column; if the attribute represents a collection then the attribute is mapped as a table with a foreign key, which references the table, which represents the owning class of the attribute.
- Associations are mapped to foreign keys or to tables if the association represents a many-to-many relationship similarly to the attributes.
- Primary keys are created automatically for each table.
- Inheritance is mapped into the database according to the single table principle. It means there is one table for all classes in the inheritance hierarchy. The table contains one special column called *INSTANCEDEF*[1], which value determines the class, which is represented by concrete instance.
- Labels used in the application are mapped into the labels of elements of database schema by the function *dbNameORM*, which assures that a unique name in application stays unique in database.

[1]It means that *INSTANCEDEF* $\in$ *LABEL* and that the *INSTANCEDEF* is a keyword in the model and cannot be used as a new of a class, attribute or association etc.

The whole ORM is defined in appendix C. Each schema which define the ORM is denoted by the '*-ORM*' postfix. The schemas in appendix C define how an entities layer can be mapped into a database structure. They do not define how instances are obtained in a working software. The relation between entities and database schema is given by the Z schema called *ORM*.

5.5 Software

The software contains the entities and database, which is defined as the Z schema:

$$\begin{array}{l} \textit{SOFTWARE} \\ \hline \textit{entities} : \textit{ENTITIES} \\ \textit{database} : \textit{DATABASE} \\ \hline \end{array}$$

The consistency of software depends on consistency of its parts - if both entities and database are consistent, then the software is consistent as well. Next condition of consistency is that the database schema is created according to the given ORM.

$$\begin{array}{l} \forall s : \textit{SOFTWARE};\ \textit{entities} : \textit{ENTITIES};\ \textit{database} : \textit{DATABASE} \bullet \\ \quad \textit{entities} = s.\textit{entities} \land \textit{database} = s.\textit{database} \land \\ \quad \textit{entities} \neq \textit{ERRENTITIES} \land \\ \quad \textit{database} \neq \textit{ERRDATABASE} \land \\ \quad \textit{ORM}[\textit{entities}/e?, \textit{database}/d?] \end{array}$$

The *SOFTWARE* represents a consistent software, which can be transformed by the transformations. A special kind of *SOFTWARE* is used in case the software is inconsistent or if the transformation fails. This special state of software is defined as:

$$\textit{ERRSOFTWARE} : \textit{SOFTWARE}$$

All possible softwares creates a domain, where the *ERRSOFTWARE* represents the bottom of the domain.

The transformations, which are able to change the structure of software are in the Sec. 6.3. These transformations are interpreted as transformations on the level of entities and database.

Chapter 6

Transformations for Application and Database Co-Evolution

The catalogue of transformations contains definitions of the basic set of transformation for data evolution. The transformations in the set are chosen because they represent the basic transformations needed for model and data manipulation or because they represent common refactorings [1] and their usage can speed up the process of software development. The group of common refactoring is chosen based on their popularity in community [66] [67].

Most of the transformation waere defined in the Alloy modeling tool [68] [69] first to prove transformation feasibility. Then they were rewriten in the Z language, because we believe that the Z notation is easier to understand for the reader.

The transformations respect the description of the model driven co-evolution framework architecture described in Sec. 3.1.3.

6.1 Transformation Definition

Each transformation definition consists of three parts, which defines the process of transformation execution. These parts are:

1. **Initial model verification** First the initial model is verified if it is consistent and contains no errors i.e. if it is not the $ERRSOFTWARE$.

2. **Preconditions verification** The preconditions of the transformation are verified in context of the input $SOFTWARE$. If they are verified successfully, then the state can be changed.

3. **State difference description** The last part of transformation definition describe the difference between the initial and the final state.

The processing of transformation is described by the following schema[1], where the symbol 'X' denotes the transformation:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{executeOnSoftware}[X]} \\
\hline
\Delta \mathit{SOFTWARE} \\
\mathit{decl}(X) \\
\hline
\theta \mathit{SOFTWARE} \neq \mathit{ERRSOFTWARE} \land \mathit{pre}(X) \Rightarrow \\
\quad X \lor \\
\theta \mathit{SOFTWARE} = \mathit{ERRSOFTWARE} \Rightarrow \\
\quad \theta \mathit{SOFTWARE}' = \mathit{ERRSOFTWARE} \lor \\
\theta \mathit{SOFTWARE} \neq \mathit{ERRSOFTWARE} \land \neg\, \mathit{pre}(X) \Rightarrow \\
\quad \theta \mathit{SOFTWARE}' = \mathit{ERRSOFTWARE} \\
\hline
\end{array}$$

Way transformations are executed assures the whole system is type-safe. The transformation has either valid software input and it is successful, then it produces a new software. Or the transformation's input is the *ERRSOFTWARE*, then a transformation cannot be applied and the situation results in to the *ERRSOFTWARE*. The last case is that the prerequisites of an transformations are not fulfilled in the context of the input software then the result is the *ERRSOFTWARE* again.

6.2 Composition of Transformations

The schema *executeOnSoftware* is used to describe how the transformations can be concatenated. To describe concatenation of transformations we use the possibilities of the Z-language.

The concatenation of transformations by logical symbol '$\land$' is used when the transformation has to be processed at the same time - i.e. preconditions of both has to be fulfilled and the difference between the initial and final state has to conform to both definitions.

The concatenation of transformations by pipelining is denoted '$A \gg B$' and it is used when the transformation A has to be executed before the transformation B is applied. There is an intermediate state arising as the result of the transformation A. This intermediate state is the input state for B as well as other outputs of A. These intermediate

[1]The schemas which describes the execution of a transformation is not syntactically correct Z schema. However, we decide to use the Z-like notation because the semantics of the schemas is clear enough. Moreover, these schemas shorten all transformations defined later in this theses.

state and outputs are not visible to the user of the framework. The user sees the output of B as the final state. The result of the composition is $ERRSOFTWARE$ if the intermediate state is equal to $ERRSOFTWARE$, which corresponds with the *executeOnSoftware* schema. This way of composition is used in the catalogue, if a transformation is processed in a sequence of smaller steps.

This techniques of composition are used in the thesis as the evolutionary transformation are composed from the transformations of entities and database.

The declaration from all composed transformations have to be presented in the schema which contains composition according to the Z definition [61]. We decide to not present the declaration from composed transformations for sake of length of this thesis, therefore only the declaration used directly in the schema, which contains compositions, are declared in a transformation.

6.3 Catalogue of Transformations

Evolutionary transformations, which creates the core of the framework, are described in this section. All transformations are interpreted on the entities layer and on the database layer of the software. The evolutionary transformations use the transformations for application and database manipulation, which are defined in Appendix A.2 and in Appendix A.4. The transformations, which change the layer of entities have postfix '*-EL*' and postfix '*-DB*' denote the transformations, which change the database. The postfix '*-ORM*' denotes schemas which define the entities - database mapping, which is defined in Appendix C. The evolutionary transformations of the whole software are without postfix.

Each transformation is a transaction i.e. both application and database has to be changed otherwise the transformation fails.

The transformation catalogue contains the definition of the transformation's preconditions and the difference between initial and final state only. The complete form of the transformation is insertion of the definition from catalogue into schema *executeOnSoftware* as the 'X'. All transformation presented in this section creates a set of all transformations:

$$[TRANSFORMATION]$$

Structure of Catalogue List

Each catalogue list in following section defines one transformation. The structure of the list consists of four parts, which are common for all transformations:

- Definition: The first part describes the purpose of the transformation as well as its definition in the Z language.
- Prerequisites: The second part defines the prerequisites of a transformation.The prerequisites are emphasizes in separate section because the definition of a transformation can consist of a set of smaller sub-transformations and therefore the prerequisites could be hidden in the high-level transformation.
- Impact on Data: The third part describes how the transformation affects stored instances.
- Inverse Transformation: The fourth part defines, which transformation can be used to revert the defined transformation. In some case there are more possibilities in dependency on concrete state of the software.

6.3.1 Add Class

Inserts a class into the application and corresponding table into the database and then it adds all attributes from the given set to the given class. The database is changed according to the ORM.

$$
\begin{array}{|l}
\textit{addClass} \\
\hline
\Delta \mathit{SOFTWARE} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{DATABASE} \\
c? : \mathit{CLASS} \\
att? : \mathbb{P}\, \mathit{ATTRIBUTE} \\
\hline
\mathit{addEntityEL} \gg \mathit{entityToTableORM} \gg \mathit{addTableDB} \gg \\
[| \; \forall p : \mathit{ATTRIBUTE} \bullet p \in atts \Rightarrow \mathit{addAttribute}[p/p?]] \\
\hline
\end{array}
$$

Prerequisites

The transformation is possible if there is no possible name collision. The precondition respects the invariants of the model.

$$
\mathit{prerequisites}(\mathit{addClass}) = \{(\forall c : \mathit{CLASS} \bullet c \in \mathit{classes} \Rightarrow c.\mathit{label} \neq c?.\mathit{label})\}
$$

Impact on Data

The transformation has no impact on stored data.

Inverse Transformation

The transformation can be reverted by the *removeClass* or by the *removeClassWithNoInstances* transformation. The first option does not keep stored data, whereas the second is possible only it there are no data stored in table representing the entity in the database.

6.3.2 Remove Class

The transformation removes the given class from the software as well as all attributes belonging to the class. The database is changed as well.

$$
\begin{array}{|l}
\hline
\textit{removeClass} \\
\Delta \mathit{SOFTWARE} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{DATABASE} \\
c? : \mathit{CLASS} \\
\hline
\forall\, p : \mathit{ATTRIBUTE} \bullet \\
\quad p \in \mathit{attributesOf}(c?, \theta(\mathit{ENTITIES})) \Rightarrow \mathit{removeAttribute}[p/p?] \gg \\
\mathit{removeEntityEL} \gg \mathit{entityToTableORM} \gg \mathit{dropTableDB} \\
\hline
\end{array}
$$

Prerequisites Remove an entity is possible only if there are no children of the given class and if there is no association, which targets the given class. Aim of these restriction is to create atomic transformations, which are easy to understand. A transformation which removes the entity and which is able to remove all references or handle the inheritance relationship can be created as a composition of atomic transformations by the user. Next condition is that the class is part of the entities' layer. This condition can be considered redundant, however its meaning is that if a user wants to delete a non-existing class there is some kind of inconsistency between the real state of the entities and user's understanding of the entities layer.

$$
\begin{array}{|l}
\hline
\mathit{prerequisetes}(\mathit{removeClass}) = \{(c? \in \mathit{classes}), \\
\quad (\mathit{children}(c?, \theta(\mathit{ENTITIES})) = \{\mathit{NULLCLASS}\}), \\
\quad (\mathit{associationsTargeting}(c?, \theta(\mathit{ENTITIES})) = \{\mathit{NULLASSOCIATION}\})\ \}
\end{array}
$$

Impact on Data The transformation can delete all data, which represents instances of the given class. If we want to preserve the stored data we can use the *removeClassWithNoInstances* transformation.

Inverse Transformation The transformation cannot be reverted in general case. Only in case, if there are no instances of the removed class the *removeClass* can be reverted by the *addClass* transformation.

Remove Class with no Instances

The transformation *removeClassWithNoInstances* removes the given class only if there are no instance of the class. The transformation is defined to provide a general reverse transformation for the *addClass* transformation.

$$
\begin{array}{|l}
\hline
\textit{removeClassWithNoInstances} \\
\Delta \textit{SOFTWARE} \\
\Delta \textit{ENTITIES} \\
\Delta \textit{DATABASE} \\
c? : \textit{CLASS} \\
\hline
\textit{entityToTableORM}[ts/ts!] \\
\textit{selectAllData}(ts, \theta \textit{DATABASE}) = \varnothing \\
\textit{removeEntityEL} \\
\textit{entityToTableORM} \gg \textit{dropEmptyTableDB} \\
\hline
\end{array}
$$

Prerequisites

$$
\begin{array}{|l}
\hline
\textit{prerequisites}(\textit{removeClassWithNoInstances}) = \\
\quad \{\textit{selectAllData}(ts, \theta \textit{DATABASE}) = \varnothing\} \cup \textit{prerequisites}(\textit{removeClass})
\end{array}
$$

Impact on Data The transformation has no impact on stored data.

Inverse Transformation The transformation can be reverted by the *addClass* transformation.

6.3.3 Add Attribute

Inserts an attribute into the given class in the application and corresponding column or table into the database.

$$
\begin{array}{|l}
\hline
\textit{addAttribute} \\
\Delta SOFTWARE \\
\Delta ENTITIES \\
\Delta DATABASE \\
c? : CLASS \\
p? : ATTRIBUTE \\
\hline
entityToTableORM[ts/ts!] \\
p?.optional = True \Rightarrow selectAllData(ts, \theta(DATABASE)) \neq \varnothing \\
addAttributeEL \\
(p?.upper = One \Rightarrow attributeToColumnORM \gg addColumnDB) \\
(p?.upper = Many \Rightarrow attributeToTableORM \gg addTableDB) \\
\hline
\end{array}
$$

Prerequisites The transformation is possible if the class exists in the context of entities' layers and if there is no possible name collision between attributes' label in the class. Next the transformation cannot be applied in case the attribute is optional and there are data stored in the table, which corresponds to the owning class. In such a case we are missing a value, which can be added as the new column value.

$$
\begin{array}{|l}
\hline
prerequisites(addAttribute) = \{(c? \in classes), \\
\quad (\{p : ATTRIBUTE \mid \\
\qquad p \in attributesOf(c?, \theta(ENTITIES)) \wedge p.label = p?.label\} = \varnothing), \\
\quad (p?.optional = True \Rightarrow \\
\qquad selectAllData(entityToTableORM, \theta(DATABASE)) \neq \varnothing)\}
\end{array}
$$

Impact on Data The transformation has no impact on stored data.

Inverse Transformation The transformation can be reverted by the *removeAttribute.* The revert can cause data loss. If the transformation should proceed only if there are no data values stored in the corresponding column or table in database the *removeAttribute-WithNoData* transformation should be used.

6.3.4 Remove Attribute

The transformation removes the given attribute from the given class and corresponding column (or table) is removed as well as stored data.

$$\begin{array}{|l}
\textit{removeAttribute} \\
\hline
\Delta \mathit{SOFTWARE} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{DATABASE} \\
\Delta \mathit{TABLESCHEMA} \\
c? : \mathit{CLASS} \\
p? : \mathit{ATTRIBUTE} \\
\hline
\mathit{removeAttributeEL} \\
(p?.\mathit{upper} = \mathit{One} \land \mathit{attributeToColumnORM} \gg \mathit{dropColumnDB}) \lor \\
(p?.\mathit{upper} = \mathit{Many} \land \mathit{attributeToTableORM} \gg \mathit{dropTableDB}) \\
\hline
\end{array}$$

Prerequisites The transformation is feasible if the attribute exists in the model.

$$\begin{array}{|l}
\hline
\mathit{prerequisites}(\mathit{removeAttribute}) = \{(c? \in \mathit{classes}), (p? \in \mathit{attributes}), \\
\quad (\exists\, \mathit{poc} : \mathit{ATTRIBUTEOfCLASS} \bullet \\
\qquad \mathit{poc} \in \mathit{attributesOfClasses} \land \mathit{poc.class} = c? \land \mathit{poc.attribute} = p?)\}
\end{array}$$

Impact on Data The transformation can remove data from the database, therefore it is not data safe.

Inverse Transformation In general case the transformation cannot be reverted. The transformation can be reverted by the *addAttribute* transformation, if there are no values in column (or table), which represents attribute in the database.

6.3.5 Remove Attribute with no Data

The evolutionary transformation which protects data in the database is called *removeAttributeWithNoData*.

$$\begin{array}{|l}
\textit{removeAttributeWithNoData} \\
\hline
\Delta \mathit{SOFTWARE} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{DATABASE} \\
\Delta \mathit{TABLESCHEMA} \\
c? : \mathit{CLASS} \\
p? : \mathit{ATTRIBUTE} \\
\hline
\mathit{removeAttributeEL} \gg \\
(p?.\mathit{upper} = \mathit{One} \land \mathit{attributeToColumnORM} \gg \mathit{dropEmptyColumnDB}) \lor \\
(p?.\mathit{upper} = \mathit{Many} \land \mathit{attributeToTableORM} \gg \mathit{dropEmptyTableDB}) \\
\hline
\end{array}$$

Prerequisites The prerequisites of the transformation are the same as in case of standard *removeAttribute* with additional verification that there are no data stored in the table or in the column (in dependency on the attribute cardinality).

$$\begin{array}{l}
prerequisites(removeAttributeWithNoData) = prerequisites(removeAttribute) \cup \\
\quad \{(\{d : DATAVALUES \mid \\
\qquad d \in values \wedge d.definition = attributeToTableORM\} = \varnothing \Leftrightarrow \\
\quad p?.upper = Many) \vee \\
(\{cv : COLUMNVALUE \mid \\
\qquad cv.definition = attributeToColumnORM\} = \varnothing \Leftrightarrow p?.upper = One)\}
\end{array}$$

Impact on Data The transformation has no impact on stored data.

Inverse Transformation The transformation can be reverted by the *addAttribute* transformation.

6.3.6 Add Association

Inserts a new association between two existing classes into the application and corresponding foreign key or mapping table into the database.

addAssociation

$$\begin{array}{l}
\Delta SOFTWARE \\
\Delta ENTITIES \\
\Delta DATABASE \\
a? : ASSOCIATION \\
c? : CLASS \\
\hline
entityToTableORM[ts/ts!] \\
a?.optional = True \Rightarrow selectAllData(ts, \theta(DATABASE)) \neq \varnothing \\
addAssociationEL \gg \\
[\| \; a?.upper = One \Rightarrow assocToFkORM \gg (addForeignKeyDB \gg \\
\quad insertDataToFKDB) \vee \\
a?.upper = Many \Rightarrow assocToTableORM \gg (addTableDB \gg \\
\quad insertDataToMapTableDB)]
\end{array}$$

Prerequisites The transformation is possible if there are no name collisions between the associations in the entities' layer and the new association and if the source and target of the association are in the entities' layer. The mapping has to be source complete if the association is mandatory.

$$\begin{array}{|l} prerequisites(addAssociation) = \{(a?.source \in classes), \\ \quad (a?.target \in classes), (a?.optional = False \Rightarrow MappingSourceComplete)\} \end{array}$$

Impact on Data The transformation has no impact on stored data as it changes only the structure of application and database.

Inverse Transformation The transformation can be reverted by the *removeAssociation* transformation, but this transformation is data loss. The data save inverse transformation is *removeAssociationWithNoData*.

6.3.7 Remove Association

The transformation removes the given association from the software and corresponding foreign key (or table) and its data as well. Two versions of the transformation are presented. The first version deletes stored data during its execution, whereas the second one is executable only if there are no stored data.

$$\begin{array}{|l} \textit{removeAssociation} \\ \hline \Delta SOFTWARE \\ \Delta ENTITIES \\ \Delta DATABASE \\ a? : ASSOCIATION \\ \hline removeAssociationEL \gg \\ [\,|\; a?.upper = One \Rightarrow assocToFkORM \gg dropForeignKeyDB \lor \\ a?.upper = Many \Rightarrow assocToTableORM \gg dropTableDB] \\ \hline \end{array}$$

Prerequisites The transformation is possible if the given association is in the entities' layer. If the user wants to remove non-existing associations it is interpreted as an inconsistency between the real state of the entities layer and the user's understanding of this layer.

$$\begin{array}{|l} prerequisites(removeAssociation) = \{(a? \in associations)\} \end{array}$$

Impact on Data The transformation can cause a data loss - concretely a connection between instances.

Inverse Transformation The transformation *removeAssociation* can be reverted by the *addAssociation* only in case if there are no data stored in the foreign key or mapping table.

6.3.8 Remove Association with no Data

The transformation implements removal of an attribute and corresponding database structure, which protects stored data.

$$
\begin{array}{|l}
\hline
\textit{removeAssociationWithNoData} \\
\Delta SOFTWARE \\
\Delta ENTITIES \\
\Delta DATABASE \\
a? : ASSOCIATION \\
\hline
removeAssociationEL \gg \\
[\,|\; a?.upper = One \Rightarrow assocToFkORM \gg dropEmptyForeignKeyDB \lor \\
a?.upper = Many \Rightarrow assocToTableORM \gg dropEmptyTableDB] \\
\hline
\end{array}
$$

Prerequisites The prerequisites of the *removeEmptyAssociation* are the same as in case of *removeAssociation* transformation with additional verification if there are no data stored in the database.

$$
\begin{array}{|l}
\hline
prerequisites(removeEmptyAssociation) = prerequisites(removeAssociation) \cup \\
\quad \{(\{d : DATAVALUES \mid \\
\quad\; d \in values \land d.definition = assocToTableORM\} = \varnothing \Leftrightarrow a?.upper = Many) \lor \\
\quad (\{fv : FOREIGNKEYVALUE \mid \\
\quad\; fv.definition = assocToFkORM\} = \varnothing \Leftrightarrow a?.upper = One)\}
\end{array}
$$

Impact on Data There is no impact on data.

Inverse Transformation The transformation can be reverted by the *addAssociation* transformation.

6.3.9 Move Attribute

The transformation moves the given attribute from one class to another including stored data. If the attribute represents a single value then the corresponding column is moved. If the attribute represents a collection of values then the foreign key in the table, which represents the attribute has to be redirected. If the mapping *map?* is the empty mapping, then the *moveAttribute* takes part only on the structural level.

moveAttribute

$$\begin{array}{l}
\Delta SOFTWARE \\
\Delta ENTITIES \\
\Delta DATABASE \\
c? : CLASS \\
d? : CLASS \\
p? : ATTRIBUTE \\
map? : MAPPING \\
\hline
l = p?.label \\
c? \neq d? \\
d? \notin \operatorname{ran}(childParentRelation(c?, \theta ENTITIES)^{+}) \\
c? \notin \operatorname{ran}(childParentRelation(d?, \theta ENTITIES)^{+}) \\
entityToTableORM[ts/ts!] \\
entityToTableORM[d?/c?, to/ts!] \\
(p?.upper = One \Rightarrow \\
\quad addAttribute \gg attributeToColumnORM[col/col!] \gg \\
\quad (copyColumnDB[col/col?, ts/sourceSchema?, to/targetSchema?] \gg \\
\quad dropColumnDB[col/col?])) \vee \\
(p?.upper = Many \Rightarrow \\
\quad addAttribute \gg attributeToTableORM \gg \\
\quad changeAllReferencesInTable[to/target?] \gg removeAttribute[d?/c?])
\end{array}$$

Prerequisites The transformation is possible if there are no possible name collisions in the target class, the source and target class are not the same and if the source and target class are not in the same inheritance hierarchy. If they are in the same inheritance hierarchy the transformation *pullUp* should be used. The duplicates in mapping affects the feasibility of the *moveAttribute* transformation as well. The transformation is feasible if the mapping is simple or with no target duplicates. In other cases the transformation is consider unfeasible.

$$\begin{array}{l}
prerequisites(moveAttribute) = \{\{(c?, d? \in classes), (c? \neq d?) \\
\quad (\{p : ATTRIBUTE \mid \\
\qquad p \in attributesOf(c?, \theta(ENTITIES)) \wedge p.label = p?.label\} = \varnothing), \\
\quad (d? \notin \operatorname{ran}(childParentRelation(c?, \theta ENTITIES)^{+})), \\
\quad (c? \notin \operatorname{ran}(childParentRelation(d?, \theta ENTITIES)^{+})), \\
\quad (MappingIsSimple \vee MappinNoTargetDuplicates)\}
\end{array}$$

Impact on Data The values are moved from one instance to another during the *moveAttribute* transformation. The exact effect depends on the mapping *map?*. A solution for a mapping with target duplicates could be creation of new instances. These instances contain the original target value extended by the moved attribute and its value. An issue can occur in such a situation - possible associations referencing the original target instance - should they be duplicated as well or only part of them or none of them? We consider such a big change in instances to be undesirable because it is hard to predict the impact of such a big change without detailed knowledge of the database and the domain. Therefore we decide to limit the transformation *moveAttribute* to use mappings with no target duplicates.

There are no data loss if the mapping is full complete. If the mapping is target complete there can be some instances from the source, which values are not moved and are deleted, this results in a data loss.

The attribute's data stored in the source class's instances is removed, but they are still reachable in the software according to the mapping given in the transformation.

Inverse Transformation The feasibility of the *moveAttribute* transformation revert depends on the mapping's features. The mapping has to be simple and full or source complete if the revert should be feasible. Then the transformation can be reverted by the *moveAttribute* transformation with the inverse mapping.

The *moveAttribute* cannot be reverted in case its mapping contains source duplicates, because we cannot decide which of many values has to be moved to the original class.

The mapping is problem during a revert in general, because the values which are used in the mapping during the forward transformation can changed during application lifecycle by an update or insert database query and therefore the inverse mapping can be unfeasible.

6.3.10 Inline Class

The transformation *inlineClass* inlines two different classes into one. The transformation is used when all features (attributes) have to be moved into another class and the original owning class of moved attributes is going to be removed afterwards. The transformation moves all attributes from inlined class, then all references in database are redirect (i.e. attributes and associations are redirected from inlined to the target class). Finally the inlined class is removed.

```
inlineClass
  $\Delta ENTITIES$
  $\Delta DATABASE$
  $\Delta SOFTWARE$
  $\Delta ASSOCIATION$
  $c? : CLASS$
  $d? : CLASS$
  $map? : MAPPING$
  ----
  $c? \in classes \land d? \in classes \land c? \neq d?$
  $parentOf(c?, \theta ENTITIES) = NULLCLASS$
  $children(c?, \theta ENTITIES) = \varnothing$
  $\forall q, r : ATTRIBUTE \bullet$
    $q \in attributesOf(c?, \theta(ENTITIES)) \land$
    $r \in attributesOf(d?, \theta(ENTITIES)) \Rightarrow q.label \neq r.label$
  $\forall a, b : ASSOCIATION \bullet$
    $a.source = c? \land$
    $b.source = d? \Rightarrow a.label \neq b.label$
  $[| \forall p : ATTRIBUTE \bullet$
    $p \in attributesOf(c?, \theta(ENTITIES)) \Rightarrow$
      $moveAttribute[p/p?]] \gg$
  $[| \forall a : ASSOCIATION \bullet$
    $a.target = c? \Rightarrow changeAssociationDirectionEL[d?/target?, a/a?] \land$
      $changeReferenceInDB[a/a?, d?/target?]$
  $removeClass]$
```

Prerequisites The transformation is feasible only if the class to be inlined is not part of an inheritance hierarchy and if there are no possible name collisions between attributes and associations labels. Next we assume no data loss is possible during the *inlineClass* transformation, therefore the mapping has to be full complete and simple. We limit the *inlineClass* transformation in a way it can be used only in case if there is only one attribute in the inlined class. This restriction help us to create atomic transformations and to create a reverse transformation for the *inlineClass*.

$$\begin{aligned}
&\mathit{prerequisites}(\mathit{inlineClass}) = \{(c? \in \mathit{classes}), (d? \in \mathit{classes}), (c? \neq d?),\\
&\quad (\mathit{parentOf}(c?, \theta \mathit{ENTITIES}) = \mathit{NULLCLASS}),\\
&\quad (\mathit{children}(c?, \theta \mathit{ENTITIES}) = \varnothing),\\
&\quad (\forall\, q, r : \mathit{ATTRIBUTE} \bullet\\
&\qquad q \in \mathit{attributesOf}(c?, \theta(\mathit{ENTITIES})) \land\\
&\qquad r \in \mathit{attributesOf}(d?, \theta(\mathit{ENTITIES})) \Rightarrow q.\mathit{label} \neq r.\mathit{label}),\\
&\quad (\forall\, a, b : \mathit{ASSOCIATION} \bullet\\
&\qquad a.\mathit{source} = c? \land b.\mathit{source} = b? \Rightarrow a.\mathit{label} \neq b.\mathit{label}),\\
&\quad (\mathit{MappingIsComplete} \land \mathit{MappingIsSimple})\}
\end{aligned}$$

Impact on Data The transformation extends instances of the class, which is the target of the transformation, whereas instances of the inlined class are removed from the software. Because the mapping is complete and simple there is no data loss during the transformation.

Inverse Transformation The transformation can be reverted by the *splitClass* transformation with usage of inverse mapping if there are no associations targeting the inlined class. There are no special issue during the revert, because the mapping *map?* is complete and simple.

6.3.11 Split Class

The transformation creates a new class and then moves an attribute from the source class into the new class.

splitClass

$$\begin{aligned}
&\Delta \mathit{ENTITIES}\\
&\Delta \mathit{DATABASE}\\
&\Delta \mathit{TABLESCHEMA}\\
&\Delta \mathit{SOFTWARE}\\
&l? : \mathit{LABEL}\\
&ps? : \mathbb{P}\, \mathit{ATTRIBUTE}\\
&\mathit{toSplit}? : \mathit{CLASS}
\end{aligned}$$

$$\begin{aligned}
&\mathit{att} = \varnothing\\
&\mathit{initEntity}[g/c!]\\
&\mathit{addClass}[g/c?, \mathit{att}/\mathit{att}?] \gg \mathit{initMappingForSplit}[\mathit{toSplit}?/\mathit{old}?, g/\mathit{new}?] \gg\\
&\mathit{moveAttribute}[\mathit{toSplit}?/d?, g/c?]
\end{aligned}$$

Prerequisites The prerequisites are the same as in case of *addClass* transformation.

$prerequisites(splitClass) = prerequisites(addClass)$

Impact on Data The transformation creates new instances (new stored data). These data have the same primary key value as it use to have in the source table. The mapping created during the transformation is simple and full complete so no data loss occurs during the transformation.

Inverse Transformation The transformation can be reverted by the *inlineClass* transformation. The input mapping for the reverting transformation is based on equality of primary keys' values. The mapping has to be simple and full complete otherwise the *inlineClass* does not revert the *splitClass* transformation.

6.3.12 Extract Class

The transformation is similar to the *splitClass* transformation with the difference it adds an association between the original and the new class. Inspiration for this transformation is the refactoring "Replace Data Value with Object" [1]. The transformation proceeds in following steps: a class is extracted, then a new association is initialized and added into the application. All these steps are processed for the database level at the same time.

extractClass

$\Delta SOFTWARE$
$\Delta ENTITIES$
$\Delta DATABASE$
$c?, d : CLASS$
$p? : ATTRIBUTE$
$l? : LABEL$

$l = p?.label$
$u = p?.upper$
$o = p?.optional$
$splitClass[c?/toSplit?, d/g] \gg$
$initAssociation[l/label?, u/upper?, o/optional?, c?/source?, d/target?, a/a!] \gg$
$initMappingForSplit[c?/old?, g/new?] \gg$
$addAssociation[a/a?]$

Prerequisites The prerequisites of the *extractClass* transformation are the same as in case of *splitClass* transformation. Important is the first step - new class creation.

$prerequisites(extractClass) = prerequisites(addClass)$

Impact on Data The transformation changes the structure and stored instances stored in class, from which is extracted and its children (if exists). New instances occurs in the extracted class and the association between classes and their instances occurs.

Inverse Transformation The transformation has the same prerequisites and similar impact on data as the *splitClass* transformation. The only difference is that the *extract-Class* transformation creates a new foreign keys' values if there are instances in the source class.

The transformation can be reverted by the combination of *inlineClass* and *remove-Association* transformations. The mapping, which is used in *inlineClass* is based on the association between the two classes created during the *extractClass*.

6.3.13 Add Parent

Adds a parent - child relationship between the two given classes. The transformation creates a column which distinguish instances first, then it merges the tables which represents the parent and the child in one table. Finally all references are updated in the database.

$$
\begin{array}{l}
\textit{addParent} \\
\hline
\Delta \mathit{ENTITIES} \\
i? : \mathit{INHERITANCE} \\
\mathit{map}? : \mathit{MAPPING} \\
\Delta \mathit{SOFTWARE} \\
\Delta \mathit{DATABASE} \\
\hline
i?.\mathit{parent} = \mathit{parent} \\
i?.\mathit{child} = \mathit{child} \\
\mathit{children}(\mathit{child}, \theta \mathit{ENTITIES}) = \varnothing \\
\mathit{entityToTableORM}[\mathit{child}/c?, \mathit{cts}/\mathit{ts}!] \\
\mathit{entityToTableORM}[\mathit{parent}/c?, \mathit{pts}/\mathit{ts}!] \\
\mathit{children}(\mathit{parent}, \theta \mathit{ENTITIES}) = \varnothing \Rightarrow \\
\quad [\mid \mathit{constraints} = \{\mathit{NOTNULL}\} \wedge \\
\quad\quad \mathit{initColumn}[\mathit{INSTANCEDEF}/l?, \mathit{constraints}/\mathit{constraints}?] \gg \\
\quad\quad \mathit{addColumnDB}[\mathit{pts}/\mathit{ts}?]] \gg \\
\mathit{addEntityParentEL} \gg \\
[\mid \forall c : \mathit{COLUMN} \bullet c \in \mathit{cts}.\mathit{columns} \Rightarrow \\
\quad \mathit{copyColumnDB}[\mathit{col}/\mathit{col}?, \mathit{cts}/\mathit{sourceSchema}?, \mathit{pts}/\mathit{targetSchema}?] \gg \\
\quad \mathit{dropColumnDB}[\mathit{col}/\mathit{col}?, \mathit{cts}/\mathit{ts}?]] \gg \\
[\mid \forall \mathit{fk} : \mathit{FOREIGNKEY};\ \mathit{ts} : \mathit{TABLESCHEMA} \bullet \\
\quad \mathit{fk} \in \mathit{foreignKeys} \wedge \mathit{ts} = \mathit{fk}.\mathit{source} \Rightarrow \\
\quad \mathit{changeAllReferencesInTable}[\mathit{pts}/\mathit{target}?, \mathit{ts}/\mathit{ts}?] \gg \\
\mathit{dropTableDB}[\mathit{cts}/\mathit{ts}?]] \\
\hline
\end{array}
$$

Prerequisites The transformation is possible if the to-be-child class has no parents, because the multiple inheritance is not allowed in the model. Next, because we do not insert values into the child class instances during the creation of the inheritance relationship, we have to verify there are no optional values (attributes) in the parent class. The last prerequisite is that only a class without children can be added into inheritance hierarchy. This constraint is added for the simplicity of the model.

$$
\begin{array}{l}
\mathit{prerequisites}(\mathit{addParent}) = \{(\mathit{parentOf}(\mathit{child}, \theta(\mathit{ENTITIES})) = \mathit{NULLCLASS}), \\
\quad (\{p : \mathit{ATTRIBUTE} \mid p \in \mathit{attributesOf}(\mathit{parent}, \theta(\mathit{ENTITIES})) \wedge \\
\quad\quad \mathit{NOTNULL} \notin p.\mathit{constraints}\}), (\mathit{children}(\mathit{child}, \theta \mathit{ENTITIES}) = \varnothing), \\
\quad (\mathit{MappingSourceComplete} \vee \mathit{MappingIsSimple})\}
\end{array}
$$

Impact on Data The transformation extends structure of instances stored in child class by the structure of the parent class. There can occur instances of the to-be-parent class, which are not extended by a child. Therefore the mapping is source complete.

The mapping is simple to assure a direct reverting operation exists. A parent is added only to one instance of a child class and a instance cannot correspond with two possible parent's instances.

Inverse Transformation The transformation can be reverted by the *removeParent* transformation.

6.3.14 Remove Parent

Destroys the parent-child relationship between two classes. The transformation affects the database significantly - a new table is created, columns are moved and references are updated. Finally if the parent has no more children the column, which distinguish kind of instances is removed from the parent.

```
removeParent
  $\Delta SOFTWARE$
  $\Delta ENTITIES$
  $\Delta DATABASE$
  $c? : CLASS$
  ----
  $children(c?, \theta ENTITIES) = \varnothing$
  $parent = parentOf(c?, \theta ENTITIES)$
  $removeEntityParentEL \gg$
  $entityToTableNoAttributesORM[cts/ts!] \gg$
  $addTableDB[cts/ts?] \gg$
  $entityToTableORM[parent/c?, pts/ts!] \gg$
  $initMappingForRemoveParent[map/map!, cts/cts?] \gg$
  $[\| \forall c : COLUMN \bullet (c \in pts.columns \land \exists p : ATTRIBUTE \bullet$
    $p \in attributesOf(c?, \theta ENTITIES) \land$
    $attributeToColumnORM[p/p?, c/col!]) \Rightarrow$
      $copyColumnDB[col/col?, pts/sourceSchema?, cts/targetSchema?,$
       $map/map?] \gg dropColumnDB[col/col?, pts/ts?]] \gg$
  $[\| \forall t : TABLESCHEMA \bullet \exists p : ATTRIBUTE \bullet$
    $p \in attributesOf(c?, \theta ENTITIES) \land attributeToTableORM[p/p?, t/ts!] \Rightarrow$
      $changeReferenceTableDB[t/ts?, c?/newTarget?, parent/oldTarget?,$
       $me/map?]] \gg$
  $[\| \forall a : ASSOCIATION \bullet a \in associationsTargeting(c?, \theta ENTITIES) \Rightarrow$
      $(a.upper = One \Rightarrow assocToFkORM[a/a?] \gg$
        $changeFKreferenceDB[cts/targetSchema?, me/map?]) \lor$
      $(a.upper = Many \Rightarrow assocToTableORM[a/a?] \gg$
       $changeReferenceTableDB[c?/newTarget?, parent/oldTarget?,$
       $me/map?])] \gg$
  $[col : COLUMN \mid children(parent, \theta ENTITIES) = \varnothing \Rightarrow$
    $col.label = INSTANCEDEF \land dropColumnDB[col/col?]]$
```

Prerequisites The transformation is possible if the inheritance is part of the entities' layer. Missing inheritance is considered to be a semantics inconsistency. To simplify the model we assume the child class is a leaf of the inheritance hierarchy.

$$\begin{aligned}&prerequisites(removeParent) = \{(children(c?, \theta ENTITIES) = \varnothing),\\&\quad(\exists i : INHERITANCE \bullet i \in inheritance \land i.child = c?)\}\end{aligned}$$

Impact on Data The instances of the child class looses information which was stored in their parent.

Inverse Transformation The transformation can be reverted by the *addParent* transformation, where the mapping is based on equality of primary keys' values.

6.3.15 Push Down

The transformations moves one attribute from the parent class to all child classes. The transformation takes part at the level o entities only, because the inheritance is mapped into the database as a single table by the ORM. Therefore no data moves are needed in the database.

pushDown
$\Delta ENTITIES$
$\Delta CLASS$
$p? : ATTRIBUTE$
---
pushAttributeDownEL

Prerequisites The transformation is possible if the given attribute is an attribute owned by the given class within the given entities.

$$prerequisites(pushDown) = \{(\theta(CLASS) \in classes), (p? \in attributesOf(\theta(CLASS), \theta(ENTITIES))\}$$

Impact on Data The transformation has impact on all instances of the parent class, which are shortened by the pushed attribute. The transformation causes therefore a data loss.

Inverse Transformation The transformation cannot be reverted by the simple *pullUp* transformation, because the *pullUp* has only one attribute and one class as its input. To revert the *pushDown* a new refactoring needs to be defined. This refactoring is called *pullCommonAttributeUp*.

6.3.16 Push Attribute Down to a Class

The transformation is similar to the *pushDown* transformation. In contrast it moves the attribute only to one child. Therefore the values in other children, which used to represent the moved attribute has to be removed from the table.

$$
\begin{array}{|l}
\textit{pushAttributeDownToClass} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{CLASS} \\
\Delta \mathit{CLASS} \\
p? : \mathit{ATTRIBUTE} \\
\hline
p? \in \mathit{attributesOf}(\mathit{parent}, \theta(\mathit{ENTITIES})) \\
\mathit{parent} = \mathit{parentOf}(\mathit{child}, \theta(\mathit{ENTITIES})) \\
\mathit{addAttributeEL}[\mathit{child}/c?] \gg \\
\mathit{removeAttributeEL}[\mathit{parent}/c?] \gg \\
[\,|\ (p?.\mathit{upper} = \mathit{One} \Rightarrow \quad \forall\, dv, dv' : \mathit{DATAVALUES} \bullet \\
\quad\quad \mathit{entityToTableORM}[\mathit{parent}/c?, \mathit{tab}/\mathit{ts}!] \wedge \\
\quad\quad \mathit{isInstanceOf}(dv, \mathit{child}) \neq \mathit{True} \Rightarrow \mathit{attributeToColumnORM}[\mathit{col}/\mathit{col}!] \wedge \\
\quad\quad\quad dv'.\mathit{colValues} = dv.\mathit{colValues} \setminus \{cv : \mathit{COLUMNVALUE} \mid \\
\quad\quad\quad\quad cv.\mathit{definition} = \mathit{col}\})]
\end{array}
$$

Prerequisites The prerequisites are the same as in case of the *pushDown* transformation.

$$
\begin{array}{|l}
\hline
\mathit{prerequisites}(\mathit{pushAttributeDownToClass}) = \mathit{prerequisites}(\mathit{pushDown})
\end{array}
$$

Impact on Data The transformation has impact on all instances of the parent class and on all instances of child classes (besides the target class of the push down), which are shortened by the pushed attribute. The transformation causes therefore a data loss.

Inverse Transformation The refactoring *pullUp* can be used to revert the *pushAttributeDownToClass*. The revert is possible only if there were no stored instances when pushing down, because the transformation *pushAttributeDownToClass* is data loss.

6.3.17 Pull Up

The transformation moves one attribute from the child class into the parent class.

$$
\begin{array}{|l}
\textit{pullUp} \\
\Delta \mathit{ENTITIES} \\
\Delta \mathit{CLASS} \\
\mathit{parent}? : \mathit{CLASS} \\
p? : \mathit{ATTRIBUTE} \\
\hline
p?.\mathit{optional} = \mathit{True} \Rightarrow \#\mathit{children}(\mathit{parent}?, \theta \mathit{ENTITIES}) = 1 \\
\mathit{pullAttributeUpEL}
\end{array}
$$

Similarly to the *pushDown* the transformation takes part at the level o entities only, because the inheritance is mapped into the database as a single table by the ORM. Therefore no data moves are needed in the database.

Prerequisites If there are more than one children class, then the attribute has not be optional, because we have no value, which can be used in instances of other children.

$$\begin{array}{l} \mathit{prerequisites}(\mathit{pullUp}) = \{(p?.\mathit{optional} = \mathit{True} \Rightarrow \\ \quad \#\mathit{children}(\mathit{parent}?, \theta \mathit{ENTITIES}) = 1)\} \end{array}$$

Impact on Data The transformation extends the structure of the instances of the parent and all its children.

Inverse Transformation The transformation can be reverted by the *pushAttribute-DownToClass* transformation.

6.3.18 Pull Common Attribute Up

The transformations pulls up a property only if it is defined in all child classes of a parent.

$$\begin{array}{l} \mathit{pullCommonAttributeUp} \\ \hline \Delta \mathit{ENTITIES} \\ c? : \mathit{CLASS} \\ p? : \mathit{ATTRIBUTE} \\ \hline c_p = \mathit{parentOf}(c?, \theta(\mathit{ENTITIES})) \\ cs = \{c : \mathit{CLASS} \mid p? \in \mathit{attributesOf}(c, \theta(\mathit{ENTITIES})) \wedge \\ \quad c \in \mathit{children}(c_p, \theta(\mathit{ENTITIES}))\} \\ \mathit{addAttributeEL}[c/c?] \\ \forall\, c : \mathit{CLASS} \bullet \\ \quad c \in cs \Rightarrow \mathit{removeAttributeEL}[c/c?] \\ \hline \end{array}$$

Prerequisites The prerequisite of the *pullCommonAttributeUp* is that the attribute, which is going to be pulled up, has to be present in all child classes of the parent.

$$\begin{array}{l} \mathit{prerequisites}(\mathit{pullCommonAttributeUp}) = \{\{c_p = \mathit{parentOf}(c?, \theta(\mathit{ENTITIES}))\}, \\ \{\forall\, c : \mathit{CLASS} \bullet c \in \mathit{children}(c_p, \theta(\mathit{ENTITIES})) \Leftrightarrow \\ \quad p? \in \mathit{attributesOf}(c, \theta(\mathit{ENTITIES}))\}\} \end{array}$$

Impact on Data The transformation has no impact on stored data because they are already stored in the appropriate table.

Inverse Transformation The transformation can be inverted by the *pushDown* transformation.

6.3.19 Extract Parent

The transformation creates a new class with the given attribute and creates the parent-child relationship between the new and the original class.

extractParent

$$\begin{array}{l}
\Delta SOFTWARE \\
\Delta ENTITIES \\
\Delta DATABASE \\
parent, child?, child : CLASS \\
l? : LABEL \\
p? : ATTRIBUTE \\
\hline
parentOf(child?, \theta(ENTITIES)) = NULLCLASS \\
p? \in attributesOf(child?, \theta(ENTITIES)) \\
att = \varnothing \\
initEntity[parent/c!] \\
addClass[parent/c?, att/att?] \gg initInheritance \gg \\
initMappingForExtractParent[parent/parent?] \gg \\
addParent[parent/parent?] \gg pullUp[parent/parent?]
\end{array}$$

Prerequisites The prerequisites of the *extractParent* are that there is no parent of the source class and that the attribute is owned by the to-be-child class.

$$\begin{array}{l}
prerequisites(extractParent) = \{(p? \in attributesOf(child?, \theta(ENTITIES))), \\
\quad (parentOf(child?, \theta(ENTITIES)) = NULLCLASS)\}
\end{array}$$

Impact on Data The transformation causes no data loss during its execution. The structure is changed only.

Inverse Transformation The transformation can be reverted by the combination of *pushDownToClass* and *removeParent*.

6.3.20 Merge Classes

The transformation merges two classes with the same structure into one class. The transformation looks similar as the *inlineClass* transformation, but its meaning is different. The *mergeClasses* transformation merges two classes with the same structure into one and similarly it merge instances from two tables into one table. Whereas the *inlineClass* transformation extends the structure of the target class (and table) and instances are not merged but extended. This transformation is not part of the common code refactorings, but we add into the framework to create possibility to reduce duplicities in code. As this need occurs during some testing scenarios.

```
mergeClasses
  $\Delta SOFTWARE$
  $\Delta ENTITIES$
  $\Delta DATABASE$
  $c?, c_2? : CLASS$
  ----------------
  $attributesOf(c?, \theta ENTITIES) = attributesOf(c_2?, \theta ENTITIES)$
  $associationsOf(c?, \theta ENTITIES) = associationsOf(c_2?, \theta ENTITIES)$
  $parentOf(c?, \theta ENTITIES) = parentOf(c_2?, \theta ENTITIES)$
  $parentOf(c?, \theta ENTITIES) = NULLCLASS$
  $children(c?, \theta ENTITIES) = children(c_2?, \theta ENTITIES)$
  $children(c?, \theta ENTITIES) = \varnothing$
  $\forall a : ATTRIBUTE \bullet$
    $a \in attributesOf(c?, \theta ENTITIES) \Leftrightarrow a.upper = One$
  $isReferenced(c?, \theta ENTITIES) = \varnothing$
  $isReferenced(c_2?, \theta ENTITIES) = \varnothing$
  $entityToTableORM[ts/ts!]$
  $entityToTableORM[c_2?/c?, ts2/ts!]$
  $[| \forall k, k_2, k_2' : DATAVALUES \bullet$
    $k.definition = ts \land k_2.definition = ts2 \land k \in values \land k_2 \in values$
    $\land k.key \neq k_2.key \Rightarrow (k_2'.definition = ts \land k_2' \notin values \land k_2' \in values' \land$
    $k \notin values') \lor$
  $(\forall k, k_2, k_2' : DATAVALUES \bullet$
    $k.definition = ts \land k_2.definition = ts2 \land k \in values \land k_2 \in values$
    $\land k.key = k_2.key \Rightarrow k_2'.definition = ts \land k_2'.key.value = next(sequence))] \gg$
  $removeClass[c_2?/c?]$
```

Prerequisites The transformation is possible only if the structure of both classes is the same and if the classes are not part of an inheritance hierarchy and are not referenced. Next, all the attributes has to be single values. The restriction are necessary to keep the transformation small enough to be easy to understand by the user. Next reason is that the transformation represents significant change (mostly) in the database. Therefore it is necessary to create conditions for safe application of the transformation.

$$\begin{array}{|l}
\hline
\mathit{prerequisites}(\mathit{mergeClasses}) = \{ \\
\quad (\mathit{attributesOf}(c?, \theta \mathit{ENTITIES}) = \mathit{attributesOf}(c_2?, \theta \mathit{ENTITIES})), \\
\quad (\mathit{associationsOf}(c?, \theta \mathit{ENTITIES}) = \mathit{associationsOf}(c_2?, \theta \mathit{ENTITIES})), \\
\quad (\mathit{parentOf}(c?, \theta \mathit{ENTITIES}) = \mathit{parentOf}(c_2?, \theta \mathit{ENTITIES})), \\
\quad (\mathit{parentOf}(c?, \theta \mathit{ENTITIES}) = \mathit{NULLCLASS}), \\
\quad (\mathit{children}(c?, \theta \mathit{ENTITIES}) = \mathit{children}(c_2?, \theta \mathit{ENTITIES})), \\
\quad (\mathit{children}(c?, \theta \mathit{ENTITIES}) = \varnothing), \\
\quad (\forall\, a : \mathit{ATTRIBUTE} \bullet \\
\quad a \in \mathit{attributesOf}(c?, \theta \mathit{ENTITIES}) \Leftrightarrow a.\mathit{upper} = \mathit{One}), \\
\quad (\mathit{isReferenced}(c?, \theta \mathit{ENTITIES}) = \varnothing), \\
\quad (\mathit{isReferenced}(c_2?, \theta \mathit{ENTITIES}) = \varnothing)\}
\end{array}$$

Impact on Data The transformation moves instances from one class to another with the same structure i.e. theirs definition name is changed. The primary key value of some stored data can change during the transformation. Therefore the associations' direction and foreign keys' values has to changed as well.

Inverse Transformation The transformation *mergeClasses* does not have a direct inverse transformation, because the transformation changes not only the structure, but the values as well. If there are no data in tables, the transformation can be reverted by the combination of *addClass*, *addAttribute* and *addAssociation*.

6.4 Transformation Set Completness

As discussed in Sec. 6 there are many transformations, which do not have an inverse transformation in general case. Thus the set *TRANSFORMATION* is not complete. The conditions for completeness is that for each transformation $t : \mathit{TRANSFORMATION}$, there is a transformation $t^{-1} : \mathit{TRANSFORMATION}$, which is an inverse function to t i.e. $t^{-1}(t(s)) = s$, where $s \in \mathit{SOFTWARE}$ and $t(s) \neq \mathit{ERRSOFTWARE}$.

We discuss alternative inverse transformations for transformations with no general inverse in Sec. 6 and some transformations can be inverted if their input mapping fulfill certain conditions or if given transformations' prerequisites are fulfilled. Therefore there is a subset of *TRANSFORMATION* set, which is complete. This subset is called:

[*CoTRANSFORMATION*]

and it contains following transformations, which can be reverted:

- **addClass** can be reverted by the *removeClassWithNoInstances* transformation.
- **removeClassWithNoInstances** can be reverted by the *addClass* transformation.
- **addAttribute** can be reverted by the *removeAttributeWithNoData* transformation.
- **removeAttributeWithNoData** can be reverted by the *addAttribute* transformation.
- **addAssociation** can be reverted by the *removeAssociationWithNoData* transformation.
- **removeAssociationWithNoData** can be reverted by the *addAssociation* transformation.
- **moveAttribute** the transformation can be reverted by the *moveAttribute* transformation with a inverse mapping.
- **inlineClass** the transformation can be reverted by the *splitClass* with a inverse mapping.
- **splitClass** the transformation can be reverted by the *inlineClass* with a inverse mapping.
- **addParent** the transformation can be reverted by the *removeParent* transformation.
- **removeParent** the transformation can be reverted by the *addParent* transformation.
- **pushDown** the transformation can be reverted by the *pullCommonAttributeUp* transformation.
- **pullCommonAttributeUp** the transformation can be reverted by the *pushDown* transformation.

On one side the set *CoTRANSFORMATION* creates a complete set, on the other side this set provides limited user experience during framework usage; especially in case of removing elements from the model.

Chapter 7

Model of Software Versioning

We have mentioned two approaches to software versioning in Sec. 2.2. First of them, state-based versioning, is based on a sequence of software states and the second, operation-based versioning, is based on a record of sequence of transformations, which produced current state. We decide to use the second approach combined with our framework for application and database co-evolution and create a model driven framework for application and database co-versioning.

The model driven operation-based VCS and its important features are defined in this section: first a model of operation-based VCS is introduced in Sec. 7.1, then creation and deletion of branches is discussed in Sec. 7.3, versioning (evolution and revert to a previous state) is introduced in Sec. 7.2 and finally merging of branches and collision solving in Sec. 7.4. The transformation defined earlier are updated in a way, that they respect the features in the VCS in Sec. 7.5. Parts of this section has been published in [42].

7.1 The Model of Operation-Based VCS

The set of transformations, which produce a new version in the operation-based VCS is the most important part of the whole system. The set of transformations, which is used in our model, contains all transformations defined in Sec. 6.3. In this section we show how an operation-based VCS can be built with use of the *TRANSFORMATION* set.

A general operation-based VCS is represented as follows:

GENERALOPVCS

$$\begin{array}{|l} initial : SOFTWARE \\ transformations : \mathrm{seq}\ TRANSFORMATION \\ \hline initial \neq ERRSOFTWARE \end{array}$$

The operation-based VCS contains only the initial state of the software and the sequence of transformations, which produces the current state from the initial one. Any consistent *SOFTWARE* can be used as the initial state of the operation-based VCS. In the forthcoming text we assume that the initial *SOFTWARE* is an empty one i.e. it does not contain any classes. Therefore we can simplify the definition by omitting the *initial* state of the *GENERALOPVCS*.

We are interested in the co-versioning of application and database from the developer's point of view. Therefore we are interested in current state of the database as well. Although we are able to reconstruct the structure of the *SOFTWARE* (definition of the entities and database schema) from the transformations stored in the *transformation* sequence, we are not able to reconstruct the stored data, because the *TRANSFORMATION* set contains only structural transformation and it does not contain transformations for inserting, deleting or updating of data, which are not needed by the developer, but whose results (i.e. stored data) can affect feasibility of the structural transformations.

The version is defined as follows in our case:

OPERATIONBASEDVCS

$$\begin{array}{|l} current : SOFTWARE \\ transformations : \mathrm{seq}\ TRANSFORMATION \end{array}$$

7.2 Revert a Transformation

The transformations for creation of a new version are identical with transformations for application and database co-evolution as defined in Sec. 6. Inverse transformations were discussed in the catalogue of evolutionary transformations as well. This section provides a deeper insight on the revert in context of the operation-based VCS.

A revert to a previous state is a must-have function for each VCS. The fact we would like to keep versions of application structure and stored data as well brings two challenges. First is that the set of transformations we defined is not complete. Second challenge is lack of information about changes of data, which are not caused by a structural data evolution.

7.2.1 Missing Inverse Transformation

The *TRANSFORMATION* set is not complete, therefore there are limitations in some cases of transformation's revert. The problem is to get all necessary information for inverse e.g. if we remove a class together with all its attributes and data by the transformation *removeClass* then the result of the inverse transformation *addClass* will not correspond to the original state because the attributes of the original class and its data will be missing.

To solve this problem we define the *CoTRANSFORMATION* set (see Sec. 6.4). On such restricted set of transformations, each adding transformation has a remove transformation as its reversal transformation and vice versa. On the other hand, this solution reduces users' experience with the *OPERATIONBASEDVCS*, because they have to prepare the software for each removal. Constraining of the transformation set is solution for the structural transformations, but it does not cover the change of data in the running software.

7.2.2 Change of the Data

Second problem of reverting a change is that the *TRANSFORMATION* set contains only structural transformations and refactorings. It means, that all the data changes (such as insert, delete or update) are not stored in the *transformations* history sequence and they are not stored in the operation-based VCS.

The change of data may cause problems with consistency of pairs in the mapping e.g. the mapping can refer to values, which no longer exist. Therefore data changes can cause a situation when application of inverse transformation is no longer possible.

Possible solution is to add the transformations affecting data into the *CoTRANSFORMATION* set. The transformations for insert and delete data are inverse transformations and update can be reverted by another update. All data changing transformations have to be in a form, which contains all information necessary for revert i.e. in update transformation the whole original data has to be presented in the definition:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{updateDataDB}} \\
\Delta \mathit{DATABASE} \\
\mathit{original?} : \mathbb{P}\, \mathit{DATAVALUES} \\
\mathit{new?} : \mathbb{P}\, \mathit{DATAVALUES} \\
\hline
\mathit{values}' = (\mathit{values} \setminus \mathit{original?}) \cup \mathit{new?} \\
\hline
\end{array}
$$

and the inverse transformation is defined as follows:

$$\begin{array}{|l}
\hline
\textit{updateDataDBInverse} \\
\Delta \mathit{DATABASE} \\
\mathit{original?} : \mathit{DATAVALUES} \\
\mathit{new?} : \mathit{DATAVALUES} \\
\hline
\mathit{values}' = \mathit{values} \setminus \{\mathit{new?}\} \cup \{\mathit{original?}\} \\
\hline
\end{array}$$

where

$$\mathit{updateDataDB.original?} = \mathit{updateDataDBInverse.original?} \wedge$$
$$\mathit{updateDataDB.new?} = \mathit{updateDataDBInverse.new?}$$

This assures that the data are versioned as well. However, this solution is outside the goal of this thesis, who is to speed up the work of developers, which are not interested in each data change in running software. Moreover, this approach can cause security vulnerability of the system as the data are stored outside database. A possible solution can use a database backups created at the time of transformation execution. The revert of a transformation is connected with database revert to corresponding backup. This solution can consume a lot of disk space in case of large database. Moreover, all data collected between transformation execution and its revert is lost. Finaly, the original purpose is that the framework is capable to produce changes, which can be later propagated to different databases, which is not possible if the changes of the data are stored in the VCS. Therefore another solution has to be found.

The solution we propose is that the reverted state is not expected to be identical to the original state on the data level, but on the structural level only. The idea is illustrated in Fig. 7.1. During the evolution, new version of the entities layer and the database is created. Whereas when reverting a transformation the structure of application and database schema reverts to a previous state, but a new version of data is created. This way even the data, which was added into the new version is preserved in the database; it is adapted to the new schema.

Disadvantage of this approach is that a transformation, which has a mapping between instances as its input, has to be updated in case of transformation revert, because some data can be added or deleted from the database. It means that there can occur situations when the revert may become impossible without data loss (e.g. if the mapping pairs for the new data cannot be created), but the data loss will become obvious due to the mapping redefinition.

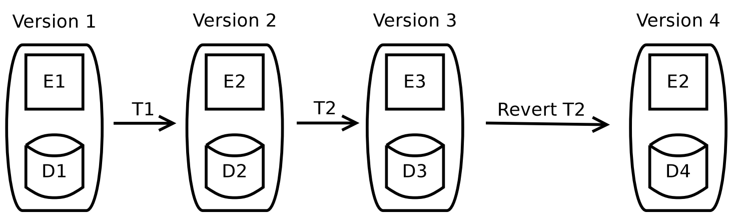


Figure 7.1: The approach for transformation reverting. An old version of entities (and database schema) is used, in contrast a new version of the database is created.

The proposed solution is not ideal as it represents a compromise between developer's code-first approach and data preservation in different software instances.

The process of change reverting is not as straightforward as the evolution of software. On the other hand it changes the structure into the desired former state and it preserves data stored in the database.

7.3 Branches

Creation of branches and their merging is an important feature of every VCS, whose goal is to provide the capability to develop several variants of final software or to separate the released product version from the development versions.

The model of operation-based VCS with branches looks as follows:

OPVCSWITHBRANCHES

$$branches : \mathbb{P}\, OPERATIONBASEDVCS$$

$$\begin{aligned}&\forall\, b_1, b_2 : OPERATIONBASEDVCS;\ tt, tt_1, tt_2 : \mathrm{seq}\, TRANSFORMATION \bullet\\&\quad b_1 \in branches \land b_2 \in branches \Rightarrow\\&\quad tt \frown tt_1 = b_1.transformations \land tt \frown tt_2 = b_2.transformations\end{aligned}$$

All branches in the *OPVCSWITHBRANCHES* have a common subsequence in the beginning, because they arise from the same initial branch. This feature is used later during merge of branches. The VCS is initialized with one empty initial branch, therefore the smallest common sequence is an empty sequence.

The creation of branches is simple. A new clone of a given branch is created and added into the set of branches:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{newBranch}} \\
\Delta OPVCSWITHBRANCHES \\
ovcs : OPERATIONBASEDVCS \\
\hline
branches' = branches \cup \{ovcs\} \\
\hline
\end{array}$$

The operation for branch removal just removes a branch from the set of branches:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{remBranch}} \\
\Delta OPVCSWITHBRANCHES \\
ovcs : OPERATIONBASEDVCS \\
\hline
branches' = branches \setminus \{ovcs\} \\
\hline
\end{array}$$

Branching in operation-based VCS is in general very similar to branching in the state-based VCS. However, when a new branch arises it is derived from an existing branch, which means it contains its data (if exists) as well. This can be used during the development when the test data adapts to the new versions of the software.

7.4 Merge of Branches

The most difficult problem related to branches is merging of two different branches. Merge workflow has following steps in case of state-based VCS [48]: i) detecting of collisions ii) solving collisions iii) creating of final consistent version. Similar approach is used in case of operation-based VCS as well. The difference is that the collision consists of different sequences of transformations (and therefore different current state of the entities' structure as well) or in difference of stored data, whereas in state-based VCS we are interested only in the difference between states. The collisions in an operation-based VCS are defined as violations of one of following principles of equivalency of operation-based VCS:

1. First kind of equivalency between two operation-based VCS is the equivalency of the transformation history records. Two history records are equivalent if they produce the same final structure of software.

$$\begin{array}{|l}
\mathit{areTransformationsEquivalent} : \\
\quad \mathit{OPERATIONBASEDVCS} \times \mathit{OPERATIONBASEDVCS} \rightarrow \mathit{BOOL} \\
\hline
\forall\, o_1, o_2 : \mathit{OPERATIONBASEDVCS} \bullet \\
\quad \mathit{areTransformationsEquivalent}(o_1, o_2) = \mathit{True} \Leftrightarrow \\
\qquad o_1.\mathit{current}.\mathit{entities} = o_2.\mathit{current}.\mathit{entities} \vee \\
\quad \mathit{areTransformationsEquivalent}(o_1, o_2) = \mathit{False} \Leftrightarrow \\
\qquad o_1.\mathit{current}.\mathit{entities} \neq o_2.\mathit{current}.\mathit{entities}
\end{array}$$

This kind of equivalency includes the equality of transformations as well as the equality of the entities' structure. (Database structure is derived from the entities by the ORM and thus it is considered as well.) Two *OPERATIONBASEDVCS* can be equivalent according to the definition even if they have different historical records. This kind of equivalency is called structural equivalency in the following text. A special case of equivalency is, when the historical records of both *OPERATIONBASEDVCS* are equal.

2. Second kind of equality is the equivalency of stored data. This kind of equivalency is important because even two *OPERATIONBASEDVCS*, which are equivalent according to the historical records could differ in stored data.

$$\begin{array}{|l}
\mathit{areDataEquivalent} : \\
\quad \mathit{OPERATIONBASEDVCS} \times \mathit{OPERATIONBASEDVCS} \rightarrow \mathit{BOOL} \\
\hline
\forall\, o_1, o_2 : \mathit{OPERATIONBASEDVCS} \bullet \\
\quad \mathit{areDataEquivalent}(o_1, o_2) = \mathit{True} \Leftrightarrow \\
\qquad o_1.\mathit{current}.\mathit{database}.\mathit{values} = o_2.\mathit{current}.\mathit{database}.\mathit{values} \vee \\
\quad \mathit{areDataEquivalent}(o_1, o_2) = \mathit{False} \Leftrightarrow \\
\qquad o_1.\mathit{current}.\mathit{database}.\mathit{values} \neq o_2.\mathit{current}.\mathit{database}.\mathit{values}
\end{array}$$

Two history records which are data equivalent are structural equivalent as well, whereas structurally equivalent history records could differ in stored data.

The merge process of two version proceeds in following two steps:

1. **Structural adaptation** The two merged versions are changed in a way they are structurally equivalent. This step contains collision detection and solving on the structural level.

2. **Data adaptation** The result of the step is a final version (product of the merge) which contains only desired data. The approach of data adaptation may differ according to the purpose of merge as discussed in detail later.

7.4.1 Structural Adaptation

The purpose of structural adaptation is to create two structurally equivalent versions, which has the required features. The structural adaptation prepares the versions for the next step - data adaptation.

Adaptation If One Branch has Final Structure

The adaptation with the know final structure is the most simple case of structural adaptation of two branches. This approach can be used if there is one version (branch), which represents the desired state and the other branch represents some older version of the software. This is a common situation when a new functionality is integrated with the previous version.

The most simple case is, when first version is direct predecessor of the second version, which represents desired structure. Direct predecessor means, that the whole history record of the predecessor's is a prefix of the ancestor's history record:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{isAncestor}} \\
\hline
\textit{pred}, \textit{anc} : \textit{OPERATIONBASEDVCS} \\
\hline
\textit{pred.transformations} \subseteq \textit{anc.transformations} \\
\hline
\end{array}$$

The predecessor has to be updated by applying the transformations, which creates the difference between predecessor's and ancestor's history record.

The second case is when one branch represents the desired final state, but there is no predecessor - ancestor relationship between versions. In such a case, the difference between current states has to be defined in the form of transformations' sequence. This sequence is then applied on the branch, which needs to be adapted. The transformation sequence can be obtained as a result of a model matching algorithm, applied on the models of current *SOFTWARE* states.

However, it means that this approach can produce multiple possible differences between branches [70] and a suitable one has to be chosen by the user.

Creation of the Final Structure from Both Branches

This section discuss the case desired features of the final version origins from both branches. This situation occurs e.g. in case when two branches containing newly developed functionality are going to be merged - both contain important information, which has to be propagated in to the final version. We can use similar approaches as in case of state-based VCS [48] in this case. The possible approaches are illustrated on the example of two history records:

| $t_1, t_2, t_3, t_4, t_5 : \mathit{TRANSFORMATION}$

$h_1 == \langle t_1, t_2, t_3, t_4 \rangle$
$h_2 == \langle t_1, t_2, t_3, t_5 \rangle$

The naive approach consists in sequential application of both history records on the last common version. The final sequence of transformations starts with the common parts of histories' records (in our case $h_c == \langle t_1, t_2, t_3 \rangle$) and then two possible variations arise from sequential application of h_1 and h_2. In our case we get two sequences $h_{1,2} == \langle t_1, t_2, t_3, t_4, t_5 \rangle$ and $h_{2,1} == \langle t_1, t_2, t_3, t_5, t_4 \rangle$, which both can result in a final state of the software and the user can select the sequence, which is most suitable for hers purposes. However, the choice can be limited by possible collisions. The collisions can arise because the application of transformations is not always commutative e.g. if $t_4 = \mathit{remClass}(\mathit{Person}, \mathit{Software})$ and $t_5 = \mathit{newAssociation}(\mathit{Address}, \mathit{Person}, \mathit{Software})$ then $h_{1,2}$ cause an error because for t_5 there is no class *Person*, whereas $h_{2,1}$ will create a new consistent version of the software. The naive approach considers the set of transformations only and it is based on sequential application of transformations in history records. A verification if the produced transformation sequence is well-formed (i.e. if it produce consistent final state) has to be processed by the VCS or by the user.

More sophisticated approach to creating a common final state uses information from both transformation sequences. Its application in operation-based VCS consists of searching for all possible well-formed sequences, which can produce a consistent final state of the software and which contains only the transformations, which are contained in one or both merged sequences. All such sequences are based on permutation of transformations from the sets of transformations' histories. To build all permutation is an operation with high time complexity. To reduce it we omit the initial common part of the history record and permute only the rest of history sequence (e.g. in our example the sequence h_c is not used for permutation and permuted is only following set of transformations: $\{t_4, t_5\}$). We do not assume there will be hundreds of transformations to permute in the real world scenarios. However, this assumption has to be verified on real projects. In case, this assumption is invalid a commutativity of transformations and partial order reduction could improve the time complexity of the merge process.

Structural Adaptation Summary

All mentioned approaches are suitable for situations when the requested final state of software can be obtained from historical records of merged branches. However, there are situations when the desired state has part of its features from one branch and the other features from the other branch. The VCS cannot help in such a case because user's intent cannot be derived automatically. If users want to use the features of the operation-based VCS, they should update one branch to the desired state and then use the first approach to structural merge of branches.

7.4.2 Data Adaptation

The data adaptation is a challenging process, because its failure can cause loss or corruption of stored data. We aim at software developers with our operation-based framework. Therefore we provide support for the data adaptation cases they can use during development.

First case represents the situation when a development branch is merged with the production branch. This case is simple, because we do not want to put our development (test) data in to the deployed software, we simply ignore the development data. The merge of branches then produce only a description of the structural change (i.e. adaptation of the deployed version to the newly developed structure) and the data stored in the deployed software's database changes accordingly to the structural changes.

Second case represents the situation when two development branches are merged. We can use the same approach as in the first case if the data in the source branch are not important, or we can use the *mergeClasses* transformation if the data from both branches has to be kept.

More complex solutions of collisions should be still solved manually and in cooperation with database administrator and domain specialist, because their meaning has to be interpreted in context of the whole domain.

7.5 The Extension of Transformations for VCS

The execution of each transformation, which has been defined by the schema *executeOn-Software* in Sec. 6.1 has to be updated, because the transformation and the new current state of the software has to be saved in the VCS.

First we define how a branch in the operation-based VCS is changed during the evolution:

$$
\begin{array}{|l}
\hline
\mathit{updateBranch}[X] \\
\Delta \mathit{OPVCSWITHBRANCHES} \\
\Delta \mathit{OPERATIONBASEDVCS} \\
\Delta \mathit{SOFTWARE} \\
X : \mathit{TRANSFORMATION} \\
\mathit{decl}(X) \\
\hline
\theta \mathit{OPERATIONBASEDVCS} \in \mathit{branches} \\
\theta \mathit{SOFTWARE} = \mathit{current} \\
\theta(\mathit{SOFTWARE})' = \mathit{current}' \\
\mathit{current}'! = \mathit{ERROSOFTWARE} \\
\mathit{transformations}' = \mathit{transformations} \frown \langle X \rangle \\
\mathit{branches}' = (\mathit{branches} \setminus \{\theta \mathit{OPERATIONBASEDVCS}\}) \cup \\
\quad \{\theta(\mathit{OPERATIONBASEDVCS})'\} \\
\hline
\end{array}
$$

A transformation is stored in the branch only if the transformation is successfully executed on the software i.e. it does not produce the *ERRSOFTWARE*.

The execution of a transformation introduced in Sec. 6.1 is suitable in the case of evolution. For the case of versioning the transformation has to be processed together with the branch update:

$$
\begin{array}{|l}
\hline
\mathit{executeOnOPERATIONBASEDVCS}[X] \\
\Delta \mathit{OPVCSWITHBRANCHES} \\
\Delta \mathit{OPERATIONBASEDVCS} \\
\Delta \mathit{SOFTWARE} \\
X : \mathit{TRANSFORMATION} \\
\mathit{decl}(X) \\
\hline
\mathit{executeOnSoftware} \land \mathit{updateBranch} \\
\hline
\end{array}
$$

7.6 Software Versioning Summary

The operation-based approach to software versioning can be used for co-versioning of entities and database, which preserves the semantics of the change and data stored in the application's database. However, there are limitations of the operation-based VCS e.g. the *CoTRANSFORMATION* set has to be used to allow reversal transformations, which reduces the users' experience.

It is obvious, that the operation-based VCS cannot solve all special cases during merge, however it can be useful in simple cases. The complex scenarios will be verified by future research and implementation.

Chapter 8

Implementation of Prototypes

The formal framework shows that the automatic co-evolution and even co-versioning of application and database is possible, but with many limitations. To verify if the formal model can be refined into a tool for developers we decide to implement such a tool. The framework, which was implemented to prove the model presented in this thesis, is called MigDb and it was implemented as the set of bachelor's thesis [71, 72, 73, 67, 74]. The source code is available online [75].

The architecture of the framework respect the structure presented in Fig. 3.2, which conform to the data evolution principle introduced in Sec. 2.1. The co-evolution is implemented as a set of model-to-model transformations and the final part consists in generation of an SQL script. The transformations defined in the framework slightly differs from transformations defined in Sec. 6, because different approach to inheritance mapping is used in the framework.

The SQL script produced by the framework contains queries for data definition and manipulation, which changes the structure of the database. However, it contains a queries, which verifies the state of instances in stored in the database, which is changed. The component of the framework, which executes the SQL script is capable to stop and rollback the database if it does not fulfill some preconditions.

MigDb framework implements the MDD approach to data evolution by using various tools from Eclipse Modeling Framework [76] (EMF) e.g. the technologies related to the MDD environment such as OCL [77] and QVT [78] are used.

The (meta)models are created using Ecore language and the transformations are implemented in the QVT language. A domain specific language is implemented on the top of the framework so the user can used for describing evolution instead of creating a sequence of QVT transformations. The process of co-evolution is orchestrated by the

Modeling Workflow Engine (MWE). The result of the evolution are evolved models and an SQL script for database migration. The metamodels of entities and database are created as platform independent. However, the database schema generators are created with respect to the PostgreSQL database. To improve the user experience with the framework we extend the metamodels so they are able to represent a real-world scenarios and more complex cases that the formal model. The evolution of the database schema model could be interpreted on a real database instance - it means the SQL scripts for schema alternation (re-generation) and for data migration are created and are executed. The SQL execution component of the framework is capable to verify the feasibility and consistency of the database and rollback the evolution if the evolution could not be processed safely.

The framework is able to co-evolve entities and database, however, the versioning - i.e. branching and revert - is not yet implemented.

8.1 Transformations Implemented in the MigDb Framework

The transformation are created according to the formal model to cover the basic cases of evolution - creating, updating and deleting parts of the application metamodel.

Each operation implementation consists of two parts - first there is a query which verifies the feasibility of the operation; the second part of the operation implementation is mapping from the old to the new model. The example of it is the code for adding a new class into the model:

Listing 8.1: Implementation of the *addClass* transformation in the MigDb framework.

```
query AddStandardClass::isValid(gen:Model):Boolean {
      return not gen.isEntityInModel(self.name)
  }

mapping AddStandardClass::apply(inout gen:Model) {
     gen.entities += gen.AddStandardClass(self);
  }
```

8.2 Case Study

Usage of the framework is presented on examples which are small, but they have their base in a real world problems. First we introduce the case, than we introduce how it can be solved with the defined framework.

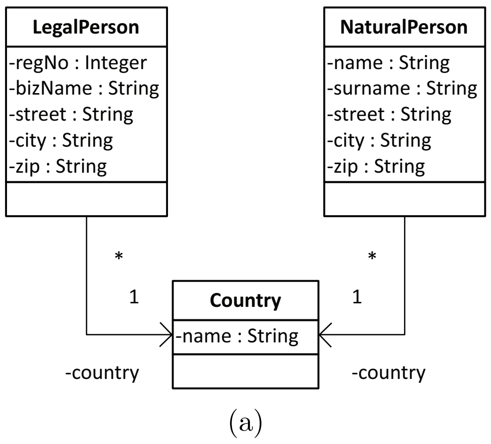
8.2.1 Description of the Case

A company implements the customer relationship management system (CRM). Part of its class model is in Fig. 8.1a. There are two types of customers - individuals (represented by a class *NaturalPerson*) and companies (represented by a class *LegalPerson*). Both classes provide information on the address, thus there are duplicities in the code (e.g. ZIP validation). Moreover more addresses of a customer cannot be stored. It means a new class *Address* should be created. The change of the entity's code structure is quite straightforward as well as the automatic generation of the new schema by a ORM framework. In contrast the evolution of stored data is more difficult, because we have to 1) merge part of the information stored in two different tables (*NaturalPerson* and *LegalPerson*) into the new one (*Address*) and preserve information about the address of each customer 2) create a new connection between the customer and the address 3) preserve connection between the address and the country.

The data migration has to solve following questions: "How to merge data from the two different sources?" and "How to preserve the information (data with their relationships) when extracting a part of a table from a table and how to verify no data were lost during the transformation?" These questions have to be solved in migration scripts and the feasibility of the solution needs to be verified for each of the deployed application, because of the stored data.

8.2.2 First Iteration

During the first iteration three entities are created. The entity *Country* represents a list of all countries used in the software; in addition there are two entities representing a natural person and a legal person. Both the *NaturalPerson* and the *LegalPerson* classes have a set of attributes, and as you can see some of them are the same (*street*, *city*, *zip*), and both entities are associated with the entity *Country*. The model is in the Fig. 8.1a. The database schema corresponding to this application model is in Fig. 8.1b.



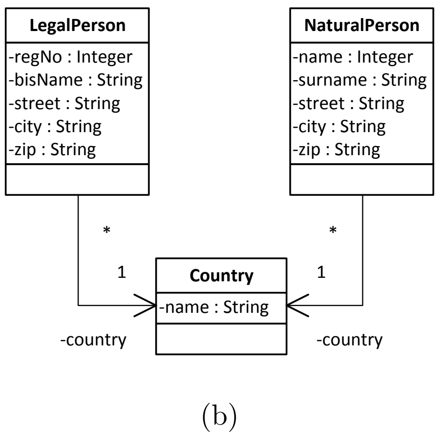


Figure 8.1: The initial state of the case study example - the model of entities in Fig. a and there is the database schema in Fig. b.

Listing 8.2: The transformations which produce the state of the software after in the first iteration.

```
addClass("Country", false, APP::InheritanceType::joined);
addProperty("Country", "name", "String");

addClass("LegalPerson", false, APP::InheritanceType::joined);
addProperty("LegalPerson", "regNo", "Integer");
addProperty("LegalPerson", "bizName", "String");
addProperty("LegalPerson", "street", "String");
addProperty("LegalPerson", "city", "String");
addProperty("LegalPerson", "zip", "String");
addProperty("LegalPerson", "country", "Country");

addClass("NaturalPerson", false, APP::InheritanceType::joined);
addProperty("NaturalPerson", "name", "String");
addProperty("NaturalPerson", "surname", "String");
addProperty("NaturalPerson", "street", "String");
addProperty("NaturalPerson", "city", "String");
addProperty("NaturalPerson", "zip", "String");
addProperty("NaturalPerson", "country", "Country");
```

8.2.3 Second Iteration

It is obvious that the software design from the first iteration suffers from duplication of code (the address has to be handled in two different entities); moreover it is impossible to have a person with two or more addresses (e.g. residential and contact). To improve the design the developers decided to evolve the software. Unfortunately there are already stored data in the database, thus the developers have to evolve the entities, database schema and migrate data.

The developers decide to do the second iteration. In this iteration they decide to solve the issue of code duplication - the aim of the code evolution is to concentrate all the address information in one class, where all the necessary application logic will be located according to the Don't Repeat Yourself (DRY) principle. This new class is called *Party* and it is a generalization of both *NaturalPerson* and *LegalPerson*.

Listing 8.3: Transformations which extracts a common parent class and moves attributes up.

```
addClass("Party", false, InheritanceType::joined);
addProperty("Party", "street", "String");
addProperty("Party", "city", "String");
addProperty("Party", "zip", "String");
addProperty("Party", "country", "Country");
setParent("LegalPerson", "Party",
      OrderedSet{"street", "city", "zip", "country"});
setParent("NaturalPerson", "Party",
     OrderedSet{"street", "city", "zip", "country"});
```

8.2.4 Third Iteration

After the second iteration the business analysts find out, they need more than one address for a customer. Therefore the the developers implement the third iteration, which adds multiple addresses per person - a new entity *Address* is extracted from the *Party* class and then two associations between entities are established - *residentialAdress* and *contactAddress*. The final state of the software is in Fig. 8.2.

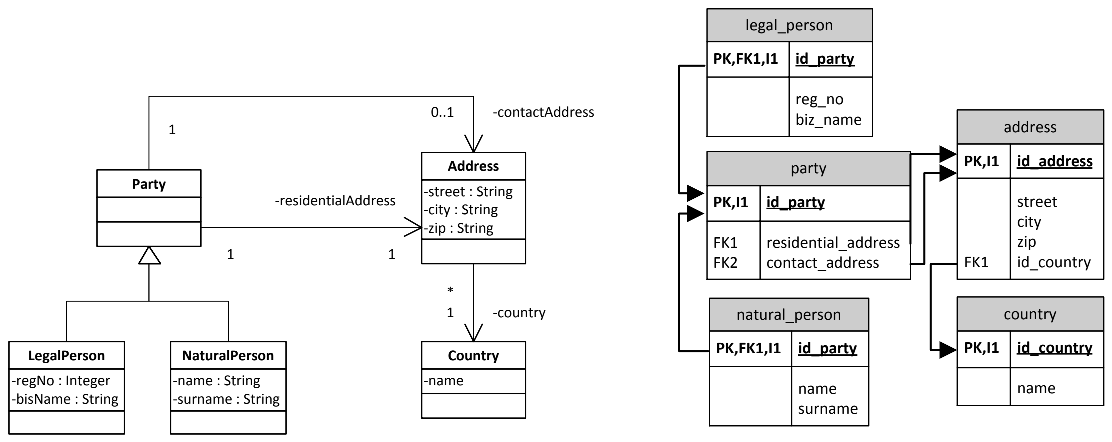


Figure 8.2: The final state of the entities model and database schema after the data evolution - the model of entities a and the database schema b.

Listing 8.4: Extracting class Address from the class Party.

```
extractClass(”Party”,
    OrderedSet{”street”, ”city”,”zip”, ”country”}, ”Address”);
renameProperty(”Party”, ”address”, ”residentialAddress”);
addProperty(”Party”, ”contactAddress”, ”Country”);
```

8.3 Case Study Summary

The complete SQL script which is generated by the MigDb framework is in the Appendix D. There is twenty-six transformations defined as the input of MigDb framework in the case study and there is about sixty-five SQL commands in the generated script. It is obvious that a user can create better - more compact - code, but the primary source is the short sequence of transformations. The main contribution of the MigDb framework is not the reduction of code, but the fact that it can generate script for data manipulation, which preserve data in the database.

8.4 Lessons Learned from the MigDb Implementation

During the implementation of the MigDb framework we found out that the MDD approach based on models and their transformations is on one hand very good tool to solve this issue on the other hand it brings a lot of problems. The biggest problem is, that change of a metamodel causes changes in all other components. Due to the immaturity of some used technologies it is hard to process these changes effectively. Moreover, it shows that the MDD approach for co-evolution and co-versioning is suitable mostly for environment where there are no big changes of the metamodels. It means that the domain has to be well known and the company has to implement a lot of projects in this domain to cover the expenses related to the creation and maintenance of proposed MDD system.

Next lesson learned from the MigDb implementation is that the mapping, which is defined in Sect. 5.3 as a general function, can be defined more specifically in the real-world scenarios. There is usually already existing connection between classes in case a *copyAttribute* or *moveAttribute* transformation is used. This allows us to implement the transformation *copyAttribute*, which uses an existing association.

Next lesson learned is that users sometimes need to specify more information for the ORM during the creation of application entity or attribute. The information, which are needed to be specified concern the database types (e.g. the representation of a number or a length of a VARCHAR column). This capability has to be added to the framework.

We verify that the framework is capable to solve common refactoring cases in context of entities and database co-evolution. However there are some known issues during transformations. Next limitation of the framework are technologies, which are used. These technologies limit the group of framework's potential users to the people with knowledge of MDD and EMF i.e. to the MDD-enthusiasts. Therefore we tried to implement two frameworks, which are using a more common technologies - Java and .NET.

The framework for Java [79] uses capabilities of the Spring Roo [80] and Liquibase. The Java framework allows to add more specific information about the ORM to the transformation. The example implementing the case study can be seen in Appendix E, where the case study example is described in form of the Spring Roo commands. From the example it is obvious that adding database specific information into the transformation definitions produce script which is about the same size as the pure SQL script.

The framework for the .NET platform [81] extends the set of Entity framework migrations and offers the capabilities of the framework defined in this thesis. Instead of using a DSL the transformations are described in form of a class, which defines the entity framework migration.

We hope at least one of the prototypes will find its users and become a valuable developer tool.

Chapter 9

Conclusion

The phenomenon of application and database co-evolution and co-versioning is examined in the thesis. We find out that the MDD approach to the co-evolution can speed up the software development, because it provides a good framework for automatic code-first co-evolution of the application and the database, which preserves stored data.

We provide a formal definition of the framework which is capable to co-evolve the application entities and the database (including data) at the same time. The definition of transformations, which represent the most common refactorings is provided in the thesis.

Next we discuss how the co-evolution can be used in the context of software versioning and how a model of operation-based VCS can be implemented based on the MDD framework for co-evolution.

Finally the implementation of the MigDb prototype is introduced. The prototype proves the idea of the proposed framework. Based on our experience from the building of the prototype the MDD approach is an efficient solution, but implementation and maintenance of such a tool is challenging.

Our future research will aim at co-evolution and co-versioning of other software layers such as GUI and security.

9.1 Results and Contribution

The thesis provides the model driven framework for application and database co-evolution and co-versioning. There are several similar projects, but this thesis still provides unique contribution to the topic. The main contributions are:

1. The architecture of the framework for application and database co-evolution is provided and discussed in context of application and database co-versioning.
2. The formal model in Z-language specifies the most common evolutionary cases (refactorings) and their impact on application and database. This formal framework can be used as an entry point for an implementation of a MDD tool for co-evolution.
3. The formal model describes the basic cases of application and database co-versioning such as branching, merging of the repository and reverting of a transformation.
4. The formal framework is verified by the prototype called MigDb, which provides feedback on the formal framework in real-world scenarios.

The results of the thesis improve the understanding of the area of model driven application and database co-evolution and co-versioning.

Bibliography

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[2] Oracle. *java.com: Java + you.* [Accessed 29. November 2013]. 2013. URL: `http://java.com/en/`.

[3] Microsoft. *Visual C#.* [Accessed 29. November 2013]. 2013. URL: `http://msdn.microsoft.com/en-us/library/vstudio/kx37x362.aspx`.

[4] Oracle. *MySQL::The world's most popular open source database.* [Accessed 29. November 2013]. 2013. URL: `http://www.mysql.com`.

[5] The PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source database.* [Accessed 29. November 2013]. 2013. URL: `http://www.postgresql.org`.

[6] JBoss Community. *Hibernate.* [Accessed 6. September 2012]. 2011. URL: `http://www.hibernate.org/`.

[7] D. H. Hansson and J. Kemper. *RubyForge:ActiveRecord: Project Info.* [Accessed 6. September 2012]. 2009. URL: `http://rubyforge.org/projects/activerecord`.

[8] D. H. Hansson and J. Kemper. *ActiveRecord::Migration.* [Accessed 6. September 2012]. 2009. URL: `http://api.rubyonrails.org/classes/ActiveRecord/Migration.html`.

[9] Microsoft. *ADO.NET Enitity Framework.* [Accessed 6. September 2012]. 2011. URL: `http://msdn.microsoft.com/en-us/library/bb399572.aspx`.

[10] N. Voxland. *Liquibase.* [Accessed 19. March 2013]. Jan. 2013. URL: `http://www.liquibase.org`.

[11] Eladio Domínguez et al. “MeDEA: A Database Evolution Architecture with Traceability”. In: *Data Knowl. Eng.* 65.3 (June 2008), pp. 419–441. ISSN: 0169-023X. DOI: 10.1016/j.datak.2007.12.001. URL: http://dx.doi.org/10.1016/j.datak.2007.12.001.

[12] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. “Graceful Database Schema Evolution: The PRISM Workbench”. In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 761–772. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1453856.1453939.

[13] Jean-Marc Hick and Jean-Luc Hainaut. “Database Application Evolution: A Transformational Approach”. In: *Data Knowl. Eng.* 59.3 (Dec. 2006), pp. 534–558. ISSN: 0169-023X. DOI: 10.1016/j.datak.2005.10.003. URL: http://dx.doi.org/10.1016/j.datak.2005.10.003.

[14] Behzad Bordbar et al. “Integrated Model-based Software Development, Data Access, and Data Migration”. In: *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems.* MoDELS’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 382–396. ISBN: 3-540-29010-9, 978-3-540-29010-0. DOI: 10.1007/11557432_28. URL: http://dx.doi.org/10.1007/11557432_28.

[15] A. Cleve and J. Hainaut. “Co-transformations in Database Applications Evolution”. In: *GTTSE.* 2005, pp. 409–421. DOI: http://dx.doi.org/10.1007/11877028_17.

[16] Enrico Franconi, Fabio Grandi, and Federica Mandreoli. “A general framework for evolving schemata support”. In: *SEBD.* 2000, pp. 371–384.

[17] Shi-Kuo Chang et al. “A Logic Framework to Support Database Refactoring”. English. In: *Database and Expert Systems Applications.* Ed. by Roland Wagner, Norman Revell, and Günther Pernul. Vol. 4653. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 509–518. ISBN: 978-3-540-74467-2. DOI: 10.1007/978-3-540-74469-6_50. URL: http://dx.doi.org/10.1007/978-3-540-74469-6_50.

[18] Peter McBrien and Alexandra Poulovassilis. “Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach”. English. In: *Advanced Information Systems Engineering.* Ed. by AnneBanks Pidduck et al. Vol. 2348. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 484–499. ISBN: 978-3-540-43738-3. DOI: 10.1007/3-540-47961-9_34. URL: http://dx.doi.org/10.1007/3-540-47961-9_34.

[19] E. McKenzie and R. Snodgrass. “Schema evolution and the relational algebra”. In: *Information Systems* 15.2 (1990), pp. 207–232.

[20] John F. Roddick, Noel G. Craske, and Thomas J. Richards. “A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models”. In: *Proceedings of the 12th International Conference on the Entity-Relationship Approach: Entity-Relationship Approach.* ER ’93. London, UK, UK: Springer-Verlag, 1994, pp. 137–148. ISBN: 3-540-58217-7. URL: `http://dl.acm.org/citation.cfm?id=647515.727030`.

[21] Lex Wedemeijer. “Semantical Change Patterns in the Conceptual Schema”. In: *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling.* ER ’99. London, UK, UK: Springer-Verlag, 1999, pp. 122–133. ISBN: 3-540-66653-2. URL: `http://dl.acm.org/citation.cfm?id=647523.728210`.

[22] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley Professional, 2006. ISBN: 0321293533.

[23] Alexandra Poulovassilis and Peter Mc. Brien. “A General Formal Framework for Schema Transformation”. In: *Data Knowl. Eng.* 28.1 (Oct. 1998), pp. 47–71. ISSN: 0169-023X. DOI: `10.1016/S0169-023X(98)00013-5`. URL: `http://dx.doi.org/10.1016/S0169-023X(98)00013-5`.

[24] Christoph Schulz, Michael Löwe, and Harald König. “A categorical framework for the transformation of object-oriented systems: Models and data”. In: *J. Symb. Comput.* 46.3 (Mar. 2011), pp. 316–337. ISSN: 0747-7171. DOI: `10.1016/j.jsc.2010.09.010`. URL: `http://dx.doi.org/10.1016/j.jsc.2010.09.010`.

[25] Ellen Van Paesschen, Wolfgang De Meuter, and Maja D’Hondt. “SelfSync: A Dynamic Round-trip Engineering Environment”. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA ’05. New York, NY, USA: ACM, 2005, pp. 146–147. ISBN: 1-59593-193-7. DOI: `10.1145/1094855.1094906`. URL: `http://doi.acm.org/10.1145/1094855.1094906`.

[26] Mohammed A. Aboulsamh and Jim Davies. “A Metamodel-Based Approach to Information Systems Evolution and Data Migration”. In: *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances.* ICSEA ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 155–161. ISBN: 978-0-7695-4144-

0. DOI: `10.1109/ICSEA.2010.31`. URL: `http://dx.doi.org/10.1109/ICSEA.2010.31`.

[27] Jay Banerjee et al. *Semantics and Implementation of Schema Evolution in Object-oriented Databases*. New York, NY, USA, Dec. 1987, pp. 311–322. DOI: `10.1145/38714.38748`. URL: `http://doi.acm.org/10.1145/38714.38748`.

[28] Randel J. Peters and M. Tamer Özsu. "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems". In: *ACM Trans. Database Syst.* 22.1 (Mar. 1997), pp. 75–114. ISSN: 0362-5915. DOI: `10.1145/244810.244813`. URL: `http://doi.acm.org/10.1145/244810.244813`.

[29] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. "SERF: Schema Evolution Through an Extensible, Re-usable and Flexible Framework". In: *Proceedings of the Seventh International Conference on Information and Knowledge Management*. CIKM '98. New York, NY, USA: ACM, 1998, pp. 314–321. ISBN: 1-58113-061-9. DOI: `10.1145/288627.288672`. URL: `http://doi.acm.org/10.1145/288627.288672`.

[30] Kajal T. Claypool, Elke A. Rundensteiner, and George T. Heineman. "Evolving the Software of a Schema Evolution System". In: FoMLaDO/DEMM 2000 (2001), pp. 68–84. URL: `http://dl.acm.org/citation.cfm?id=646201.681981`.

[31] Martin Nečaský et al. "Evolution and change management of XML-based systems". In: *Journal of Systems and Software* 85.3 (2012). Novel approaches in the design and implementation of systems/software architecture, pp. 683 –707. ISSN: 0164-1212. DOI: `http://dx.doi.org/10.1016/j.jss.2011.09.038`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121211002524`.

[32] Barbara Staudt Lerner and A. Nico Habermann. "Beyond Schema Evolution to Database Reorganization". In: *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA/ECOOP '90. New York, NY, USA: ACM, 1990, pp. 67–76. ISBN: 0-89791-411-2. DOI: `10.1145/97945.97956`. URL: `http://doi.acm.org/10.1145/97945.97956`.

[33] M. Tresch. "A Framework for Schema Evolution by Meta Object Manipulation". In: *In Proceedings of the 3d International Workshop on Foundations of Models and Languages for Data and Objects, Aigen*. 1991, pp. 1–13.

[34] Tom Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28.5 (May 2002), pp. 449–462. ISSN: 0098-5589. DOI: `10.1109/TSE.2002.1000449`. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1000449`.

[35] Reidar Conradi and Bernhard Westfechtel. “Towards a Uniform Version Model for Software Configuration Management”. In: *Proceedings of the SCM-7 Workshop on System Configuration Management.* ICSE ’97. London, UK, UK: Springer-Verlag, 1997, pp. 1–17. ISBN: 3-540-63014-7. URL: `http://dl.acm.org/citation.cfm?id=647176.716423`.

[36] The CVS Team. *Concurrent Versions System - Summary.* [Accessed 3. December 2013]. 2013. URL: `http://savannah.nongnu.org/projects/cvs`.

[37] The Apache Software Foundation. *Apache Subversion.* [Accessed 3. December 2013]. 2013. URL: `http://subversion.apache.org`.

[38] *Mercurial SCM.* [Accessed 6. September 2012]. 2014. URL: `http://mercurial.selenic.com/`.

[39] *Git.* [Accessed 3. December 2013]. 2013. URL: `http://git-scm.com`.

[40] Ernst Lippe and Norbert van Oosterom. “Operation-based merging”. In: *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments - SDE 5.* New York, New York, USA: ACM Press, 1992, pp. 78–87. ISBN: 0897915542. DOI: `10.1145/142868.143753`. URL: `http://portal.acm.org/citation.cfm?doid=142868.143753`.

[41] M Koegel et al. “Comparing State- and Operation-Based Change Tracking on Models”. In: *Enterprise Distributed Object Computing Conference EDOC 2010 14th IEEE International.* Vol. 0. Ieee, 2010, pp. 163–172. ISBN: 9781424479665. DOI: `10.1109/EDOC.2010.15`. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5630213`.

[42] Ondrej Macek. “Model Driven Approach to Software Versioning”. English. In: *International Journal on Information Technologies and Security* 6.1 (Mar. 2014), pp. 27–38. ISSN: 1313-8251.

[43] Martin Mazanec and Ondrej Macek. “On General-purpose Textual Modeling Languages”. English. In: *DATESO 2012.* Prague: MATFYSPRESS, 2012, pp. 1–12. ISBN: 978-80-7378-171-2. URL: `http://ceur-ws.org/Vol-837/paper10.pdf`.

[44] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. “COPE - Automating Coupled Evolution of Metamodels and Models”. In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 52–76. ISBN: 978-3-642-03012-3. DOI: `10.1007/978-3-642-03013-0_4`. URL: `http://dx.doi.org/10.1007/978-3-642-03013-0_4`.

[45] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. “Rondo: A Programming Platform for Generic Model Management”. In: *SIGMOD Conference*. 2003, pp. 193–204. URL: `http://doi.acm.org/10.1145/872757.872782`.

[46] Spark Systems. *Enterprise Architect*. [Accessed 13. April 2014]. 2014. URL: `http://www.sparxsystems.com/products/ea/index.html`.

[47] Kerstin Altmanninger et al. “Why Model Versioning Research is Needed!? An Experience Report”. In: *Proceedings of the Joint MoDSE-MCCM 2009 Workshop*. 2009. URL: `http://publik.tuwien.ac.at/files/PubDat_177761.pdf`.

[48] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. “A survey on model versioning approaches”. In: *IJWIS* 5.3 (2009), pp. 271–304. URL: `http://dx.doi.org/10.1108/17440080910983556`.

[49] Amanuel Koshima, Vincent Englebert, and Philippe Thiran. “Distributed Collaborative Model Editing Framework for Domain Specific Modeling Tools”. In: ICGSE ’11 (2011), pp. 113–118. DOI: `10.1109/ICGSE.2011.18`. URL: `http://dx.doi.org/10.1109/ICGSE.2011.18`.

[50] Bernhard Westfechtel. “A Formal Approach to Three-way Merging of EMF Models”. In: IWMCP ’10 (2010), pp. 31–41. DOI: `10.1145/1826147.1826155`. URL: `http://doi.acm.org/10.1145/1826147.1826155`.

[51] Klaus-D. Engel, Richard F. Paige, and Dimitrios S. Kolovos. “Using a Model Merging Language for Reconciling Model Versions”. In: ECMDA-FA’06 (2006), pp. 143–157. DOI: `10.1007/11787044_12`. URL: `http://dx.doi.org/10.1007/11787044_12`.

[52] Eclipse Foundation. *EMF Models*. [Accessed 13. April 2014]. 2014. URL: `http://www.eclipse.org/emf/compare/l`.

[53] Yuehua Lin, Jing Zhang, and Jeff Gray. “Model comparison: A key challenge for transformation testing and version control in model driven software development”. In: (2004), pp. 219–236.

[54] Tom Mens, Ragnhild Van Der Straeten, and Maja D&#39;Hondt. "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis". In: MoDELS'06 (2006), pp. 200–214. DOI: `10.1007/11880240_15`. URL: `http://dx.doi.org/10.1007/11880240_15`.

[55] Xavier Blanc et al. "Detecting Model Inconsistency Through Operation-based Model Construction". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. New York, NY, USA: ACM, 2008, pp. 511–520. ISBN: 978-1-60558-079-1. DOI: `10.1145/1368088.1368158`. URL: `http://doi.acm.org/10.1145/1368088.1368158`.

[56] Jernej Kovse and Theo Härder. *Model-Driven Development of Versioning Systems: An Evaluation of Different Approaches*. Tech. rep. University of Kaiserslautern, Kaiserslautern, 2005.

[57] Object Management Group. *OMG Model Driven Architecture*. [Accessed 19. March 2013]. 2012. URL: `http://www.omg.org/mda/`.

[58] Pavel Moravec et al. "An practical approach to dealing with evolving models and persisted data". In: *Code Generation*. 2012.

[59] Ondrej Macek and Karel Richta. "Application and Relational Database Co-Refactoring". English. In: *Computer Science and Information Systems* (). forthcoming. ISSN: 1820-0214.

[60] Ondrej Macek. *Thesis Model in the Z language*. [Accessed 7. July 2014]. 2014. URL: `https://github.com/macekond/thesis_z_model`.

[61] Mike J. Spivey. *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN: 0-13-983768-X.

[62] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-948472-8.

[63] J. P. Bowen. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press, 1996. ISBN: 1-85032-230-9. URL: `http://www.afm.sbu.ac.uk/zbook/`.

[64] Dongwon Lee, Murali Mani, and Wesley W. Chu. "Effective Schema Conversions between XML and Relational Models". In: *IN EUROPEAN CONF. ON ARTIFICIAL INTELLIGENCE (ECAI), KNOWLEDGE TRANSFORMATION WORKSHOP (ECAI-OT*. 2002, pp. 3–11.

[65] Pavel Strnad, Oondrej Macek, and P. Jíra. "Mapping XML to Key-Value Database". English. In: *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications.* Vol. 1. Red Hook: Curran Associates, Inc., 2013, pp. 121–127. ISBN: 978-1-61208-247-9. URL: `http://www.thinkmind.org/index.php?view=article\&articleid=dbkda_2013_5_30_30098`.

[66] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. "How We Refactor, and How We Know It". In: ICSE '09 (2009), pp. 287–297. DOI: `10.1109/ICSE.2009.5070529`. URL: `http://dx.doi.org/10.1109/ICSE.2009.5070529`.

[67] David Luksch. *Katalog refaktoringů frameworku MigDb.* CVUT FEL, bachelor thesis. 2013.

[68] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (Apr. 2002), pp. 256–290. ISSN: 1049-331X. DOI: `10.1145/505145.505149`. URL: `http://doi.acm.org/10.1145/505145.505149`.

[69] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006. ISBN: 0262101149.

[70] D.S. Kolovos et al. "Different models for model matching: An analysis of approaches to support model differencing". In: *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on.* 2009, pp. 1–6. DOI: `10.1109/CVSM.2009.5071714`.

[71] Martin Lukeš. *Transformace objektových modelů.* CVUT FEL, bachelor thesis. 2011.

[72] Jiří Ježek. *Modelem řízená evoluce aplikace.* CVUT FEL, bachelor thesis. 2012.

[73] Petr Tarant. *Modelem řízená evoluce databáze.* CVUT FEL, bachelor thesis. 2012.

[74] Martin Mazanec. *Domain Specific language for MigDb.* CVUT FEL, master thesis. 2014.

[75] Macek, O. et. all. *MigDb - Database Migration Framework.* [26. March 2014]. 2014. URL: `https://github.com/migdb/migdb`.

[76] Eclipse Foundation. *Eclipse Modeling - EMF.* [Accessed 26. February 2014]. 2012. URL: `http://www.eclipse.org/modeling/emf/`.

[77] Object Management Group. *OCL 2.2 Specification.* Feb. 2010. URL: `http://www.omg.org/spec/OCL/2.2`.

[78] Object Management Group. *Query/View/Transformation, v 1.1.* 2011. URL: `http://www.omg.org/spec/QVT/1.1/`.

[79] Petr Valeš. *Framework for automatic evolution of data model and database*. CVUT FIT, master thesis (in Czech Language). 2014.

[80] GoPivotal. *Spring Roo*. [Accessed 17. May 2014]. 2014. URL: `http://projects.spring.io/spring-roo/`.

[81] Filip Číž. *The framework for evolution and refactoring of domain model in the .NET environment with the use of ORM tool Entity Framework*. CVUT FIT, master thesis. 2014.

Publications by Ondřej Macek

Referred Publications Relevant for the Thesis

Peer Reviewed Journal Papers with Impact Factor

[A.1] Macek O. et al., Application and Relational Database Co-Refactoring, Computer Science and Information Systems ISSN: 1820-0214 (forthcoming). (WOS)

Peer Reviewed Journal Papers

[A.2] Macek, O. *Model Driven Approach to Software Versioning* In: International Journal on Information Technologies and Security. 2014, vol. 6, no. 1, p. 27-38. ISSN 1313-8251. (Google Scholar)

Peer Reviewed Proceedings Papers

[A.3] Strnad, P. - Macek, O. - Jíra, P. *Mapping XML to Key-Value Database* In: DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications. Red Hook: Curran Associates, Inc., 2013, vol. 1, p. 121-127. ISBN 978-1-61208-247-9. *(Google Scholar)*

[A.4] Mazanec, M. - Macek, O. *On General-purpose Textual Modeling Languages* In: DATESO 2012. Prague: MATFYSPRESS, 2012, p. 1-12. ISSN 1613-0073.ISBN 978-80-7378-171-2.*(SCOPUS)*

The paper has been cited in (autocitations excluded):

– da Silva, Mário Sergio, Valéria Cesário Times, and Michael Mireku Kwakye. "A Framework for ETL Systems Development." Journal of Information and Data Management 3.3 (2012): 300.

[A.5] Macek, O. - Richta, K. *The BPM to UML Activity Diagram Transformation Using XSLT* In: Databases, Texts, Specifications, and Objects. Praha: ČVUT v Praze, 2009, p. 119-129. ISBN 978-80-01-04323-3.*(WOS)*

The paper has been cited in (autocitations excluded):

- Rodríguez, Alfonso, et al. "Semi-formal transformation of secure business processes into analysis class and use case models: An MDA approach." Information and Software Technology 52.9 (2010): 945-971. *(WOS)*
- Nečaský, Martin, and Irena Mlýnková. "A Framework for Efficient Design, Maintaining, and Evolution of a System of XML Applications." DATESO. 2010. *(SCOPUS)*
- Górski, Tomasz. "Transformacje do automatyzacji projektowania architektury platformy integracyjnej." Biuletyn Wojskowej Akademii Technicznej 62.2 (2013): 145-165. *(Google Scholar)*
- Conrad Bock, Raphael Barbau, Anantha Narayanan, "BPMN Profile for Operational Requirements ", Journal of Object Technology, Volume 13, no. 2 (June 2014), pp. 2:1-35 *(SCOPUS)*

Refered Publications with Partial Relevance to the Thesis

Peer Reviewed Proceedings Papers

[B.1] Macek, O. - Nečaský, M. *An Extension of Business Process Model for XML Schema Modeling* In: 2010 6th World Congress on Services. Los Alamitos: IEEE Computer Society Press, 2010, p. 383-390. ISBN 978-0-7695-4129-7.*(SCOPUS)*

Unrefered Publications

Peer Reviewed Proceedings Papers

[C.1] Loupal, P. - Kantor, A. - Macek, O. - Strnad, P. On Indexing in Native XML Database Systems In: DATESO 2012. Prague: MATFYSPRESS, 2012, p. 127-134. ISSN 1613-0073.ISBN 978-80-7378-171-2. *(SCOPUS)*

[C.2] Macek, O. - Komárek, M. Evaluation of Student Teamwork In: 2012 25th IEEE Conference on Software Engineering Education and Training. Los Alamitos, CA: IEEE Computer Soc., 2012, p. 130-134. ISSN 1093-0175.ISBN 978-0-7695-4693-3. *(WOS)*

The paper has been cited in (autocitations excluded):

- Eterovic, Yadran, Gemma Grau, and Jorge Bozo. "Teaching software processes to professionals: The approach taken by an evening master's degree program." Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference on. IEEE, 2013. *(WOS)*
- Ashraf, M.A., Shamail, S., Rana, Z.A., "Agile model adaptation for e-learning students' final-year project," Teaching, Assessment and Learning for Engineering (TALE), 2012 IEEE International Conference on, pp.T1C-18,T1C-21, 2012. *(SCOPUS)*

[C.3] Macek, O. - Komárek, M. The practical method of motivating students to iterative software development In: 24th IEEE-CS Conference on Software Engineering Education and Training, CSEE&T 2011. New Jersey: IEEE, 2011, p. 512-516. ISSN 1093-0175.ISBN 978-1-4577-0348-5. *(WOS)*

The paper has been cited in (autocitations excluded):

- Eterovic, Yadran, Gemma Grau, and Jorge Bozo. "Teaching software processes to professionals: The approach taken by an evening master's degree program." Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference on. IEEE, 2013. *(WOS)*

Appendix A

Queries Used in the Model

A.1 Queries for Entites Layer

The sections introduces queries, which are used to query the layer of entities.

A.1.1 associationsOf

The function returns all *ASSOCIATION*s which have the given class as the source (i.e. all associations, which leads from the class).

$$
\begin{array}{|l}
\mathit{associationsOf} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathbb{P}\,\mathit{ASSOCIATION} \\
\hline
\forall\, c : \mathit{CLASS};\ s : \mathit{ENTITIES};\ as : \mathbb{P}\,\mathit{ASSOCIATION} \bullet \\
\quad \mathit{associationsOf}(c, s) = \\
\qquad \{a : \mathit{ASSOCIATION} \mid a \in s.\mathit{associations} \land c = a.\mathit{source}\}
\end{array}
$$

A.1.2 associationsTargeting

The function returns all *ASSOCIATION*s in the entities layer, which reference the given class.

$$
\begin{array}{|l}
\mathit{associationsTargeting} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathbb{P}\,\mathit{ASSOCIATION} \\
\hline
\forall\, c : \mathit{CLASS};\ s : \mathit{ENTITIES};\ r : \mathbb{P}\,\mathit{ASSOCIATION} \bullet \\
\quad \mathit{associationsTargeting}(c, s) = \\
\qquad \{a : \mathit{ASSOCIATION} \mid a \in s.\mathit{associations} \land a.\mathit{target} = c\}
\end{array}
$$

A.1.3 attributesOf

The function returns all attributes, which belongs to the given class.

$$
\begin{array}{|l}
\mathit{attributesOf} : \mathit{CLASS} \times \mathit{ENTITIES} \to \mathbb{P}\,\mathit{ATTRIBUTE} \\
\hline
\forall c : \mathit{CLASS};\ e : \mathit{ENTITIES} \bullet \\
\quad \mathit{attributesOf}(c, e) = \{p : \mathit{ATTRIBUTE} \mid \\
\qquad p \in e.\mathit{attributes} \land \exists\, \mathit{poc} : \mathit{ATTRIBUTEOfCLASS} \bullet \\
\qquad\quad \mathit{poc}.\mathit{class} = c \land \mathit{poc}.\mathit{attribute} = p\}
\end{array}
$$

A.1.4 children

The function returns all direct children of the given class (i.e. children of children are not returned by the functions).

$$
\begin{array}{|l}
\mathit{children} : \mathit{CLASS} \times \mathit{ENTITIES} \to \mathbb{P}\,\mathit{CLASS} \\
\hline
\forall c_p : \mathit{CLASS};\ s : \mathit{ENTITIES} \bullet \\
\quad \mathit{children}(c_p, s) = \{c : \mathit{CLASS} \mid \\
\qquad \exists\, i : \mathit{INHERITANCE} \bullet i.\mathit{parent} = c_p \land i.\mathit{child} = c \land \\
\qquad\quad i \in s.\mathit{inheritance}\}
\end{array}
$$

A.1.5 childParentRelation

The functions provides a relation between a child and its parent. The relation is used to determine all predecessors in the inheritance hierarchy.

$$
\begin{array}{|l}
\mathit{childParentRelation} : \mathit{CLASS} \times \mathit{ENTITIES} \to \mathit{CLASS} \to \mathit{CLASS} \\
\hline
\forall c_c, c_p : \mathit{CLASS};\ s : \mathit{ENTITIES} \bullet \\
\mathit{childParentRelation}(c_c, s) = \{c_c \mapsto c_p\} \Leftrightarrow c_p = \mathit{parentOf}(c_c, s)
\end{array}
$$

A.1.6 initAssociation

The schema creates a new association according to the given information.

$$
\begin{array}{|l}
\textit{initAssociation} \\
\hline
label? : LABEL \\
upper? : CARDINALITY \\
optional? : BOOL \\
source?, target? : CLASS \\
a! : ASSOCIATION \\
\hline
a!.label = label? \\
a!.upper = upper? \\
a!.optional = optional? \\
a!.source = source? \\
a!.target = target? \\
\hline
\end{array}
$$

A.1.7 initAttribute

The schema creates a new attribute according to the given information.

$$
\begin{array}{|l}
\textit{initAttribute} \\
\hline
label? : LABEL \\
upper? : CARDINALITY \\
optional? : BOOL \\
p! : ATTRIBUTE \\
\hline
p!.label = label? \\
p!.upper = upper? \\
p!.optional = optional? \\
p!.type \in ATYPE \\
\hline
\end{array}
$$

A.1.8 initAttributeOfClass

The query creates a new instance of the *ATTRIBUTEOfCLASS* - i.e. a new relationships between a class and an attribute.

$$
\begin{array}{|l}
\textit{initAttributeOfClass} \\
\hline
c? : CLASS \\
p? : ATTRIBUTE \\
poc! : ATTRIBUTEOfCLASS \\
\hline
poc!.class = c? \\
poc!.attribute = p? \\
\hline
\end{array}
$$

A.1.9 initEntity

The schema creates a new entity according to the given information.

$$
\begin{array}{|l}
\textit{initEntity} \\
\hline
c! : \mathit{CLASS} \\
l? : \mathit{LABEL} \\
\hline
c!.\mathit{label} = l? \\
\hline
\end{array}
$$

A.1.10 initEntitites

The schema creates a new entities layer.

$$
\begin{array}{|l}
\textit{initEntities} \\
\hline
e? : \mathit{ENTITIES} \\
\hline
e?.\mathit{classes} = \varnothing \\
e?.\mathit{attributes} = \varnothing \\
e?.\mathit{associations} = \varnothing \\
e?.\mathit{attributesOfClasses} = \varnothing \\
e?.\mathit{inheritance} = \varnothing \\
\hline
\end{array}
$$

A.1.11 initInheritance

The schema creates a new parent - child relationship.

$$
\begin{array}{|l}
\textit{initInheritance} \\
\hline
i! : \mathit{INHERITANCE} \\
\mathit{parent}, \mathit{child} : \mathit{CLASS} \\
\hline
i!.\mathit{parent} = \mathit{parent} \\
i!.\mathit{child} = \mathit{child} \\
\hline
\end{array}
$$

A.1.12 isInheritanceCyclical

The function *isInheritanceCyclical* returns true if there is a cyclical hierarchy between entities (i.e. some class is its own parent) and false if there is no such cyclical relation.

$$
\begin{array}{|l}
\mathit{isInheritanceCyclical} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathit{BOOL} \\
\hline
\forall c : \mathit{CLASS};\ s : \mathit{ENTITIES} \bullet \\
\quad \mathit{isInheritanceCyclical}(c, s) = \mathit{True} \Leftrightarrow \\
\qquad \exists \mathit{par} == \mathit{childParentRelation}(c, s) \bullet c \in \operatorname{ran}(\mathit{par}^{+}) \vee \\
\quad \mathit{isInheritanceCyclical}(c, s) = \mathit{False} \Leftrightarrow \\
\qquad \forall \mathit{par} == \mathit{childParentRelation}(c, s) \bullet c \notin \operatorname{ran}(\mathit{par}^{+})
\end{array}
$$

A.1.13 isReferenced

The function returns all classes which references the given class.

$$
\begin{array}{|l}
\mathit{isReferenced} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathbb{P}\, \mathit{CLASS} \\
\hline
\forall c : \mathit{CLASS};\ e : \mathit{ENTITIES} \bullet \quad \mathit{isReferenced}(c, e) = \\
\qquad \{ \mathit{cr} : \mathit{CLASS} \mid \mathit{cr} \in e.\mathit{classes} \wedge \exists a : \mathit{ASSOCIATION} \bullet \\
\qquad\quad a.\mathit{source} = \mathit{cr} \wedge a.\mathit{target} = c \}
\end{array}
$$

A.1.14 parentOf

The function returns the parent of the given class if exists or *NULLCLASS* in case there is no parent of the given class.

$$
\begin{array}{|l}
\mathit{parentOf} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathit{CLASS} \\
\hline
\forall c_c, c_p : \mathit{CLASS};\ s : \mathit{ENTITIES} \bullet \\
\quad \mathit{parentOf}(c_c, s) = c_p \Leftrightarrow \exists i : \mathit{INHERITANCE} \bullet \\
\qquad i.\mathit{parent} = c_p \wedge i.\mathit{child} = c_c \vee \\
\quad \mathit{parentOf}(c_c, s) = \mathit{NULLCLASS} \Leftrightarrow \forall i : \mathit{INHERITANCE} \bullet \\
\qquad i.\mathit{parent} = c_p \wedge i.\mathit{child} \neq c_c
\end{array}
$$

A.1.15 parentChildRelation

The function provides a relation between a class and all its children.

$$
\begin{array}{|l}
\mathit{parentChildRelation} : \mathit{CLASS} \times \mathit{ENTITIES} \rightarrow \mathit{CLASS} \rightarrow \mathit{CLASS} \\
\hline
\forall c_c, c_p : \mathit{CLASS};\ s : \mathit{ENTITIES} \bullet \\
\quad c_c \mapsto c_p \in \mathit{parentChildRelation}(c_p, s) \Leftrightarrow c_p = \mathit{parentOf}(c_c, s)
\end{array}
$$

A.2 Transformations for Entites Manipulation

This section contains the transformations, which are able to change the layer of entities. We pick only the transformations, which are later used in the evolutionary transformations. The selected set does not represent all transformations for the entities' layer nor the minimal set of transformations.

The full definition of transformations is according to the schema, where the symbol 'X' denotes the transformation, which is executed in the context of entities:

$$\begin{array}{|l}
\textit{executeOnEntities}[X] \\
\hline
\Delta \textit{ENTITIES} \\
\textit{declarations}(X) \\
\hline
\theta \textit{ENTITIES} \neq \textit{ERRENTITIES} \land \textit{pre}(X) \Rightarrow \theta \textit{ENTITIES}' \neq \textit{ERRENTITIES} \lor \\
\theta \textit{ENTITIES} = \textit{ERRENTITIES} \Rightarrow \theta(\textit{ENTITIES}) = \theta \textit{ENTITIES}' \lor \\
\theta \textit{ENTITIES} \neq \textit{ERRENTITIES} \land \neg\, \textit{pre}(X) \Rightarrow \theta \textit{ENTITIES}' = \textit{ERRENTITIES} \\
\hline
\end{array}$$

This assures the type safety of transformations for entities manipulation.

A.2.1 addAssociationEL

Adds a new association between two classes into the entities' layer.

$$\begin{array}{|l}
\textit{addAssociationEL} \\
\hline
\Delta \textit{ENTITIES} \\
a? : \textit{ASSOCIATION} \\
\hline
\forall\, a : \textit{ASSOCIATION} \bullet \\
\quad a \in \textit{associations} \Rightarrow a.\textit{label} \neq a?.\textit{label} \\
a?.\textit{source} \in \textit{classes} \\
a?.\textit{target} \in \textit{classes} \\
\textit{associations}' = \textit{associations} \cup \{a?\} \\
\hline
\end{array}$$

A.2.2 addAttributeEL

Adds the given attribute into the given class.

addAttributeEL

$$
\begin{array}{l}
\Delta ENTITIES \\
c? : CLASS \\
p? : ATTRIBUTE \\
poc : ATTRIBUTEOfCLASS \\
\hline
c? \in classes \\
\{p : ATTRIBUTE \mid p \in attributesOf(c?, \theta(ENTITIES)) \wedge \\
\quad p.label = p?.label\} = \varnothing \\
attributes' = attributes \cup \{p?\} \\
initAttributeOfClass[poc/poc!] \\
attributesOfClasses' = attributesOfClasses \cup \{poc\}
\end{array}
$$

A.2.3 addEntityEL

The transformation adds a new entity into the entities' layer.

addEntityEL

$$
\begin{array}{l}
\Delta ENTITIES \\
c? : CLASS \\
\hline
\forall\, c : CLASS \bullet \\
\quad c \in classes \Rightarrow c.label \neq c?.label \\
attributesOf(c?, \theta(ENTITIES)) = \varnothing \\
classes' = classes \cup \{c?\}
\end{array}
$$

A.2.4 addEntityParentEL

Creates an inheritance (child – parent) relationship between two classes.

addEntityParentEL

$$
\begin{array}{l}
\Delta ENTITIES \\
i? : INHERITANCE \\
\hline
parentOf(i?.child, \theta(ENTITIES)) = NULLCLASS \\
children(i?.child, \theta(ENTITIES)) = \varnothing \\
\forall\, p_c, p_p : ATTRIBUTE \bullet \\
\quad p_c \in attributesOf(i?.child, \theta(ENTITIES)) \wedge \\
\quad p_p \in attributesOf(i?.parent, \theta(ENTITIES)) \Rightarrow \\
\quad p_p.label \neq p_c.label \\
\forall\, a_c, a_p : ASSOCIATION \bullet \\
\quad a_c.source = i?.child \wedge a_p.source = i?.parent \Rightarrow a_p.label \neq a_c.label \\
inheritance' = inheritance \cup \{i?\}
\end{array}
$$

A.2.5 changeAssociationDirectionEL

Changes the direction of the given association to the new target class.

$$\begin{array}{|l}
\hline \textit{changeAssociationDirectionEL} \\
\Delta ASSOCIATION \\
target? : CLASS \\
a? : ASSOCIATION \\
\hline
a? = \theta(ASSOCIATION) \\
target' = target? \\
\hline
\end{array}$$

A.2.6 pushAttributeDownEL

Moves the selected attribute from parent to all its child classes.

$$\begin{array}{|l}
\hline \textit{pushAttributeDownEL} \\
\Delta ENTITIES \\
\Delta CLASS \\
p? : ATTRIBUTE \\
poc : ATTRIBUTEOfCLASS \\
c : CLASS \\
\hline
c = \theta(CLASS) \\
c \in classes \\
p? \in attributesOf(c, \theta(ENTITIES)) \\
[| \forall c' : CLASS \bullet \\
\quad c' \in children(c, \theta(ENTITIES)) \Rightarrow addAttributeEL[c'/c?]] \gg \\
removeAttributeEL[c/c?] \\
\hline
\end{array}$$

A.2.7 pushAttributeDownToClassEL

To allow reverting of the *pullAttributeUpEL* transformation a new refactoring has to be defined, which pushes the attribute down to the specified class.

```
pushAttributeDownToClassEL
    $\Delta ENTITIES$
    $\Delta CLASS$
    $\Delta CLASS$
    $p? : ATTRIBUTE$
    $a, a', b, b' : CLASS$
    $poc : ATTRIBUTEOfCLASS$
    ----------------
    $a = \theta(CLASS)$
    $a' = \theta(CLASS)'$
    $a \in classes$
    $b = \theta(CLASS)$
    $b' = \theta(CLASS)'$
    $b \in classes$
    $p? \in attributesOf(a, \theta(ENTITIES))$
    $a = parentOf(b, \theta(ENTITIES))$
    $addAttributeEL[b/c?]$
    $removeAttributeEL[a/c?]$
```

A.2.8 pullAttributeUpEL

Moves the selected attribute from child to its parent.

```
pullAttributeUpEL
    $\Delta ENTITIES$
    $\Delta CLASS$
    $p? : ATTRIBUTE$
    $poc : ATTRIBUTEOfCLASS$
    $c, d : CLASS$
    ----------------
    $p? \in attributes$
    $d = parentOf(\theta(CLASS), \theta(ENTITIES))$
    $c = \theta(CLASS)$
    $addAttributeEL[d/c?] \gg removeAttributeEL[c/c?]$
```

A.2.9 pullCommonAttributeUpEL

The transformation is able to pull up an attribute, which is common for all children of the given class.

```
$pullCommonAttributeUpEL$
$\Delta ENTITIES$
$p? : ATTRIBUTE$
$c_c? : CLASS$
$poc : ATTRIBUTEOfCLASS$
$d : CLASS$
----------------
$\exists\, d == parentOf(c_c?, \theta(ENTITIES)) \bullet$
  $\exists\, cs == children(d, \theta(ENTITIES)) \bullet$
    $\forall\, c : CLASS \bullet$
      $c \in cs \Leftrightarrow c \in children(d, \theta(ENTITIES)) \wedge$
      $p? \in attributesOf(c, \theta(ENTITIES)) \wedge$
      $removeAttributeEL[c/c?]$
  $addAttributeEL[d/c?]$
```

A.2.10 removeAssociationEL

Removes an association between two classes in the entities' layer.

```
$removeAssociationEL$
$\Delta ENTITIES$
$a? : ASSOCIATION$
----------------
$a? \in associations$
$associations' = associations \setminus \{a?\}$
```

A.2.11 removeAttributeEL

Removes the given attribute from the given class.

```
$removeAttributeEL$
$\Delta ENTITIES$
$c? : CLASS$
$p? : ATTRIBUTE$
$poc : ATTRIBUTEOfCLASS$
----------------
$c? \in classes$
$p? \in attributes$
$poc \in attributesOfClasses$
$poc.class = c?$
$poc.attribute = p?$
$attributesOfClasses' = attributesOfClasses \setminus \{poc\}$
$attributes' = attributes \setminus \{p?\}$
```

A.2.12 removeEntityEL

Removes an entity from the entities' layer.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{removeEntityEL}} \\
\Delta \mathit{ENTITIES} \\
c? : \mathit{CLASS} \\
\hline
c? \in \mathit{classes} \\
\mathit{children}(c?, \theta(\mathit{ENTITIES})) = \varnothing \\
\mathit{associationsTargeting}(c?, \theta(\mathit{ENTITIES})) = \varnothing \\
\mathit{classes}' = \mathit{classes} \setminus \{c?\} \\
\hline
\end{array}
$$

A.2.13 removeEntityParentEL

Destroys an inheritance (child – parent) relationship between two classes.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{removeEntityParentEL}} \\
\Delta \mathit{ENTITIES} \\
c? : \mathit{CLASS} \\
\hline
\exists\, i : \mathit{INHERITANCE} \bullet \\
\quad i \in \mathit{inheritance} \land i.\mathit{child} = c? \land \\
\quad \mathit{inheritance}' = \mathit{inheritance}' \setminus \{i\} \\
\hline
\end{array}
$$

A.3 Queries for Database Layer

Transformations for database manipulation and initialization are defined in this section.

A.3.1 initColumn

The schema initialize a new column according to the given information.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{initColumn}} \\
\mathit{col}! : \mathit{COLUMN} \\
\mathit{constraints}? : \mathbb{P}\, \mathit{CONSTRAINT} \\
l? : \mathit{LABEL} \\
\hline
\mathit{col}!.\mathit{label} = l? \\
\mathit{col}!.\mathit{type} \in \mathit{DTYPE} \\
\mathit{col}!.\mathit{constraints} = \mathit{constraints}? \\
\hline
\end{array}
$$

A.3.2 initDatabase

The schema initialize a new empty database.

$$
\begin{array}{|l}
\textit{initDatabase} \\
\hline
d? : \textit{DATABASE} \\
\hline
d?.\textit{schemas} = \varnothing \\
d?.\textit{foreignKeys} = \varnothing \\
d?.\textit{values} = \varnothing \\
d?.\textit{sequence}.\textit{current} = 0 \\
\hline
\end{array}
$$

A.3.3 initForeignKey

The schema initialize a new foreign key according to the given information.

$$
\begin{array}{|l}
\textit{initForeignKey} \\
\hline
l? : \textit{LABEL} \\
\textit{constraints}? : \mathbb{P}\,\textit{CONSTRAINT} \\
\textit{source}? : \textit{TABLESCHEMA} \\
\textit{reference}? : \textit{TABLESCHEMA} \\
\textit{fk}! : \textit{FOREIGNKEY} \\
\hline
\textit{fk}!.\textit{label} = l? \\
\textit{fk}!.\textit{constraints} = \textit{constraints}? \\
\textit{fk}!.\textit{source} = \textit{source}? \\
\textit{fk}!.\textit{reference} = \textit{reference}? \\
\hline
\end{array}
$$

A.3.4 initPrimaryKey

The schema initialize a new foreign key according to the given information.

$$
\begin{array}{|l}
\textit{initPrimaryKey} \\
\hline
\textit{primKey}! : \textit{PRIMARYKEY} \\
l? : \textit{LABEL} \\
\hline
\textit{primKey}!.\textit{name} = l? \\
\hline
\end{array}
$$

A.3.5 initTableSchema

The schema initialize a new table schema according to the given information.

```
initTableSchema
  $ts! : TABLESCHEMA$
  $label? : LABEL$
  $primKey? : PRIMARYKEY$
  $columns? : \mathbb{P}\ COLUMN$
  ----
  $ts!.label = label?$
  $ts!.primKey = primKey?$
  $ts!.columns = columns?$
```

A.3.6 referringSchemas

The functions returns all foreign keys, which reference the given class.

```
  $referringSchemas : TABLESCHEMA \times DATABASE \to \mathbb{P}\ FOREIGNKEY$
  ----
  $\forall ts : TABLESCHEMA;\ d : DATABASE;\ fks : \mathbb{P}\ FOREIGNKEY \bullet$
    $referringSchemas(ts, d) =$
      $\{fk : FOREIGNKEY \mid fk \in d.foreignKeys \land fk.reference = ts\}$
```

A.3.7 selectAllData

The function returns all values in the table schema.

```
  $selectAllData : TABLESCHEMA \times DATABASE \to \mathbb{P}\ DATAVALUES$
  ----
  $\forall ts : TABLESCHEMA;\ d : DATABASE \bullet ts \in d.schemas \Rightarrow$
    $selectAllData(ts, d) = \{dv : DATAVALUES \mid$
    $dv \in d.values \land dv.definition = ts\}$
```

A.3.8 valueOfColumn

The functions returns the value of the given column.

```
  $valueOfColumn : COLUMN \times DATAVALUES \to \mathbb{P}\ COLUMNVALUE$
  ----
  $\forall c : COLUMN;\ d : DATAVALUES \bullet$
    $valueOfColumn(c, d) =$
      $\{cv : COLUMNVALUE \mid cv.definition = c \land cv \in d.colValues\}$
```

A.4 Transformations for Database Manipulation

The transformations for database manipulation are defined in this section. In contrast with the transformation for entities manipulation the database transformations are more complex because they handle both database schema and stored data. The selected set does not represent all transformations for the database layer nor the minimal set of transformations as in case of the transformation for entities' layer.

The full definition of transformations is according to the schema, where the symbol 'X' denotes the transformation, which is executed in the context of database:

$$
\begin{array}{l}
\textit{executeOnDatabase}[X] \\
\hline
\Delta \textit{DATABASE} \\
\textit{declarations}(X) \\
\hline
\theta \textit{DATABASE} \neq \textit{ERRDATABASE} \land \\
\quad \textit{pre}(X) \Rightarrow \theta \textit{DATABASE}' \neq \textit{ERRDATABASE} \lor \\
\theta \textit{DATABASE} = \textit{ERRDATABASE} \Rightarrow \theta \textit{DATABASE} = \theta \textit{DATABASE}' \lor \\
\theta \textit{DATABASE} \neq \textit{ERRDATABASE} \land \\
\quad \neg\, \textit{pre}(X) \Rightarrow \theta \textit{DATABASE}' = \textit{ERRDATABASE} \\
\hline
\end{array}
$$

This assures the type safety of transformations for database manipulation.

A.4.1 addColumnDB

Adds a column into the table schema in the database.

$$
\begin{array}{l}
\textit{addColumnDB} \\
\hline
\Delta \textit{DATABASE} \\
\Delta \textit{TABLESCHEMA} \\
\textit{col}? : \textit{COLUMN} \\
\hline
\forall\, \textit{col} : \textit{COLUMN} \bullet \\
\quad \textit{col} \in \textit{columns} \Rightarrow \textit{col.label} \neq \textit{col?.label} \\
\textit{columns}' = \textit{columns} \cup \{\textit{col}?\} \\
\hline
\end{array}
$$

A.4.2 addForeignKeyDB

Adds a foreign key into the given table.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{addForeignKeyDB}} \\
\Delta \mathit{DATABASE} \\
\mathit{fk}? : \mathit{FOREIGNKEY} \\
\hline
\mathit{fk}?.\mathit{source} \in \mathit{schemas} \\
\mathit{fk}?.\mathit{reference} \in \mathit{schemas} \\
\mathit{NOTNULL} \in \mathit{fk}?.\mathit{constraints} \Leftrightarrow \\
\{ \mathit{dv} : \mathit{DATAVALUES} \mid \mathit{dv}.\mathit{definition} = \mathit{fk}?.\mathit{source} \} = \varnothing \\
\mathit{foreignKeys}' = \mathit{foreignKeys} \cup \{ \mathit{fk}? \} \\
\hline
\end{array}$$

A.4.3 addTableDB

The schema adds a table into the database.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{addTableDB}} \\
\Delta \mathit{DATABASE} \\
\mathit{ts}? : \mathit{TABLESCHEMA} \\
\hline
\forall \mathit{ts} : \mathit{TABLESCHEMA} \bullet \\
\quad \mathit{ts} \in \mathit{schemas} \land \mathit{ts}.\mathit{label} \neq \mathit{ts}?.\mathit{label} \\
\mathit{schemas}' = \mathit{schemas} \cup \{ \mathit{ts}? \} \\
\hline
\end{array}$$

A.4.4 changeAllReferencesInTable

The schema changes all foreign keys in the given schema so it references a new target.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{changeAllReferencesInTable}} \\
\Delta \mathit{DATABASE} \\
\mathit{ts}? : \mathit{TABLESCHEMA} \\
\mathit{target}? : \mathit{TABLESCHEMA} \\
\mathit{map}? : \mathit{MAPPING} \\
\hline
\forall \mathit{fk} : \mathit{FOREIGNKEY} \bullet \\
\quad \mathit{fk} \in \mathit{foreignKeys} \land \mathit{fk}.\mathit{source} = \mathit{ts}? \Rightarrow \\
\quad \mathit{changeFKreferenceDB}[\mathit{fk}/\mathit{fk}?, \mathit{target}?/\mathit{targetSchema}?] \\
\hline
\end{array}$$

A.4.5 changeFKreferenceDB

Changes the referenced table in the foreign key definition. All values are updated according to the given mapping to respect the new referenced table.

changeFKreferenceDB

$$
\begin{array}{l}
\Delta DATABASE \\
fk? : FOREIGNKEY \\
targetSchema? : TABLESCHEMA \\
map? : MAPPING \\
fk : FOREIGNKEY \\
\hline
fk.label = fk?.label \\
fk.constraints = fk?.constraints \\
fk.source = fk?.source \\
fk.reference = targetSchema? \\
foreignKeys' = foreignKeys \setminus \{fk?\} \cup \{fk\} \\
\forall\, dv, dv' : DATAVALUES;\ fkv, fkv' : FOREIGNKEYVALUE; \\
\quad\quad p : MAPPINGPAIR \bullet \\
\quad dv.definition = fk.source \land \\
\quad fkv.definition = fk \land \\
\quad fkv \in dv.foreignkeyValues \land dv = p.source \Rightarrow \\
\quad fkv'.definition = fk? \land \\
\quad fkv'.value = p.target.key.value \land \\
\quad dv'.foreignkeyValues = dv.foreignkeyValues \setminus \{fkv\} \cup \{fkv'\}
\end{array}
$$

A.4.6 changeForeignKeyReferenceDB

Change references (foreign keys) in all tables in the database. Data are not affected by this transformation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{changeForeignKeyReferenceDB}} \\
\Delta \mathit{DATABASE} \\
\mathit{newReference?}, \mathit{oldReference?} : \mathit{TABLESCHEMA} \\
\mathit{label} : \mathit{LABEL} \\
\mathit{constraints} : \mathbb{P}\, \mathit{CONSTRAINT} \\
\mathit{source} : \mathit{TABLESCHEMA} \\
\mathit{fk1}, \mathit{fk2} : \mathit{FOREIGNKEY} \\
\mathit{fkv}, \mathit{fkv}' : \mathit{FOREIGNKEYVALUE} \\
\hline
\forall\, \mathit{ts} : \mathit{TABLESCHEMA};\ \mathit{fk} : \mathit{FOREIGNKEY} \bullet \\
\mathit{ts} \in \mathit{schemas} \land \mathit{fk.source} = \mathit{ts} \land \\
\mathit{fk.reference} = \mathit{oldReference?} \land \\
\mathit{label} = \mathit{fk.label} \land \\
\mathit{constraints} = \mathit{fk.constraints} \land \\
\mathit{source} = \mathit{fk.source} \land \\
\mathit{initForeignKey}[\mathit{label}/\mathit{l?}, \mathit{constraints}/\mathit{constraints?}, \mathit{source}/\mathit{source?}, \\
\quad \mathit{newReference?}/\mathit{reference?}, \mathit{fk1}/\mathit{fk!}] \gg \\
\mathit{addForeignKeyDB}[\mathit{fk1}/\mathit{fk?}] \gg \\
\mathit{changeReferenceValueInForeignKeyValueDB}[\mathit{fk}/\mathit{old?}, \mathit{fk1}/\mathit{new?}] \gg \\
\mathit{dropForeignKeyDB}[\mathit{fk}/\mathit{fk?}] \\
\hline
\end{array}
$$

A.4.7 changeReferenceInDB

The transformation changes the reference in the database if the association changes.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{changeReferenceInDB}} \\
\hline
\Xi ENTITIES \\
\Delta DATABASE \\
a? : ASSOCIATION \\
target? : CLASS \\
map? : MAPPING \\
label : LABEL \\
primKey : PRIMARYKEY \\
atts : \mathbb{P}\, ATTRIBUTE \\
schemas, schemas', tables : \mathbb{P}\, TABLESCHEMA \\
foreignKeys, foreignKeys' : \mathbb{P}\, FOREIGNKEY \\
values, values' : \mathbb{P}\, DATAVALUES \\
sequence, sequence' : SEQUENCE \\
columns : \mathbb{P}\, COLUMN \\
col : COLUMN \\
tso, ts, ts!, targetSchema, sourceSchema, table : TABLESCHEMA \\
l, name : LABEL \\
constraints : \mathbb{P}\, CONSTRAINT \\
fk, fk1, fk2 : FOREIGNKEY \\
c, d, sourceCLASS, targetCLASS : CLASS \\
\hline
entityToTableORM[target?/c?, targetSchema/ts!] \\
a?.upper = One \Rightarrow assocToFkORM \gg \\
\quad changeFKreferenceDB[targetSchema/targetSchema?] \\
a?.upper = Many \Rightarrow \exists\, c == a?.source \bullet assocToTableORM \gg \\
\quad changeReferenceTableDB[target?/newTarget?, c/oldTarget?] \\
\hline
\end{array}
$$

A.4.8 changeReferenceTableDB

The transformation changes the foreign key in a mapping table in a way it references a new table.

```
changeReferenceTableDB
  $\Xi ENTITIES$
  $\Delta DATABASE$
  $ts? : TABLESCHEMA$
  $newTarget?, oldTarget? : CLASS$
  $map? : MAPPING$
  $label : LABEL$
  $primKey : PRIMARYKEY$
  $atts : \mathbb{P}\, ATTRIBUTE$
  $schemas, schemas', tables : \mathbb{P}\, TABLESCHEMA$
  $foreignKeys, foreignKeys' : \mathbb{P}\, FOREIGNKEY$
  $values, values' : \mathbb{P}\, DATAVALUES$
  $sequence, sequence' : SEQUENCE$
  $columns : \mathbb{P}\, COLUMN$
  $col : COLUMN$
  $tso, ts, ts! : TABLESCHEMA$
  $l : LABEL$
  $constraints : \mathbb{P}\, CONSTRAINT$
  $fk : FOREIGNKEY$
  ----
  $entityToTableORM[newTarget?/c?, ts/ts!]$
  $entityToTableORM[oldTarget?/c?, tso/ts!]$
  $\forall fk : FOREIGNKEY \bullet$
    $fk.source = ts? \wedge fk.reference = tso$
  $changeFKreferenceDB[fk/fk?, ts/targetSchema?]$
```

A.4.9 changeReferenceValueInForeignKeyValueDB

Changes the table referenced by the given foreign key, according to the given mapping.

```
changeReferenceValueInForeignKeyValueDB
  $\Delta DATABASE$
  $old?, new? : FOREIGNKEY$
  $fkv, fkv' : FOREIGNKEYVALUE$
  ----
  $\forall dv, dv' : DATAVALUES;\ fkv : FOREIGNKEYVALUE \bullet$
    $dv \in values \wedge fkv.definition = old? \Rightarrow$
    $fkv'.definition = new? \wedge$
    $fkv'.value = fkv.value \wedge$
    $dv'.foreignkeyValues = dv.foreignkeyValues \setminus \{fkv\} \cup \{fkv'\} \wedge$
    $values' = values \setminus \{dv\} \cup \{dv'\}$
```

A.4.10 copyColumnDB

The transformations copies structure of the column from one table schema to another. Data are copied from the source to the target table according to the given mapping. The transformation can be used for creating a structural copy of a column as well, if the empty mapping m_e is used.

```
copyColumnDB
  $\Delta DATABASE$
  $\Delta TABLESCHEMA$
  $col? : COLUMN$
  $sourceSchema? : TABLESCHEMA$
  $targetSchema? : TABLESCHEMA$
  $map? : MAPPING$
  $targetSchema' : TABLESCHEMA$
  ----
  $targetSchema? = \theta TABLESCHEMA$
  $targetSchema' = \theta(TABLESCHEMA)'$
  $col? \in sourceSchema?.columns$
  $sourceSchema? \in schemas$
  $targetSchema? \in schemas$
  $targetSchema'.columns = targetSchema?.columns \cup \{col?\}$
  $\forall m : MAPPINGPAIR;\ cval, dval : DATAVALUES;$
    $colval : COLUMNVALUE \bullet$
      $m.source.definition = sourceSchema? \land m.target.definition = targetSchema? \land$
      $cval = m.source \land dval = m.target \land colval.definition = col? \Rightarrow$
      $values' = values \setminus \{dval\} \cup \{dval' : DATAVALUES \mid$
        $dval'.colValues = dval.colValues \cup \{colval\} \land$
        $dval'.key = dval.key \land dval'.definition = dval.definition \land$
        $dval'.foreignkeyValues = dval.foreignkeyValues\}$
```

A.4.11 copyTableDB

The transformation creates a copy of the given table with a new name.

```
copyTableDB
  $\Delta DATABASE$
  $ts? : TABLESCHEMA$
  $l? : LABEL$
  $ts : TABLESCHEMA$
  ----
  $copyTableStructureDB \land$
  $\forall dv : DATAVALUES \bullet$
    $dv \in values \land dv.definition = ts? \Rightarrow$
    $values' = values \cup \{dv' : DATAVALUES \mid$
      $dv'.definition = dv.definition \land dv'.colValues = dv.colValues \land$
      $dv'.key = dv'.key \land dv'.foreignkeyValues = dv.foreignkeyValues\}$
```

A.4.12 copyTableStructureDB

The transformations creates a copy of the given table. The new table (copy) has a new name defined by *label*. The data are copied as well.

```
copyTableStructureDB
  $\Delta DATABASE$
  $ts? : TABLESCHEMA$
  $l? : LABEL$
  $ts : TABLESCHEMA$
  ----
  $ts? \in schemas$
  $\forall t : TABLESCHEMA \bullet$
    $t \in schemas \Rightarrow t.label \neq l?$
  $ts.label = l?$
  $ts.columns = ts?.columns$
  $ts.primKey = ts?.primKey$
  $schemas' = schemas \cup \{ts\}$
  $\forall fk : FOREIGNKEY \bullet$
    $fk \in foreignKeys \land fk.source = ts? \Rightarrow$
    $foreignKeys' = foreignKeys \cup \{fk' : FOREIGNKEY \mid$
      $fk'.source = ts \land fk'.reference = fk.reference \land$
     $fk'.constraints = fk.constraints \land fk'.label = fk.label\}$
```

A.4.13 dropColumnDB

Removes a column from the table schema as well as all data stored in the column.

```
dropColumnDB
    $\Delta DATABASE$
    $\Delta TABLESCHEMA$
    $col? : COLUMN$
    ----------
    $col? \in columns$
    $columns' = columns \setminus \{col?\}$
    $\forall dv, dv' : DATAVALUES;\ cv : COLUMNVALUE \bullet$
      $dv.definition = \theta TABLESCHEMA \land cv.definition = col? \land$
      $cv \in dv.colValues \Rightarrow$
        $dv'.definition = dv.definition \land dv'.key = dv.key \land$
        $dv'.foreignkeyValues = dv.foreignkeyValues \land$
        $dv'.colValues = dv.colValues \setminus \{cv\} \land$
        $values' = (values \setminus \{dv\}) \cup \{dv'\}$
```

A.4.14 dropEmptyForeignKeyDB

Removes a foreign key from the table schema only if there are no data stored.

```
dropEmptyForeignKeyDB
    $\Delta DATABASE$
    $fk? : FOREIGNKEY$
    ----------
    $\{fkv : FOREIGNKEYVALUE \mid fkv.definition = fk?\} = \varnothing$
    $dropForeignKeyDB$
```

A.4.15 dropEmptyTableDB

Removes a table schema from the database only if there are no stored data.

```
dropEmptyTableDB
    $\Delta DATABASE$
    $ts? : TABLESCHEMA$
    ----------
    $\{d : DATAVALUES \mid d \in values \land d.definition = ts?\} = \varnothing$
    $dropTableDB$
```

A.4.16 dropEmptyColumnDB

Removes a column from the table schema only if there are no data stored in the given column.

$$
\begin{array}{|l}
\textit{dropEmptyColumnDB} \\
\hline
\Delta DATABASE \\
\Delta TABLESCHEMA \\
col? : COLUMN \\
\hline
\{cv : COLUMNVALUE \mid cv.definition = col?\} = \varnothing \\
dropColumnDB \\
\hline
\end{array}
$$

A.4.17 dropForeignKeyDB

Removes a foreign key from the table schema.

$$
\begin{array}{|l}
\textit{dropForeignKeyDB} \\
\hline
\Delta DATABASE \\
fk? : FOREIGNKEY \\
\hline
fk? \in foreignKeys \\
\forall\, dv, dv' : DATAVALUES;\ fv : FOREIGNKEYVALUE \bullet \\
\quad dv.definition = fk?.source \land fv.definition = fk? \land \\
\quad fv \in dv.foreignkeyValues \Rightarrow \\
\qquad dv'.definition = dv.definition \land dv'.key = dv.key \land \\
\qquad dv'.colValues = dv.colValues \land \\
\qquad dv'.foreignkeyValues = dv.foreignkeyValues \setminus \{fv\} \land \\
\qquad values' = (values \setminus \{dv\}) \cup \{dv'\} \\
foreignKeys' = foreignKeys \setminus \{fk?\} \\
\hline
\end{array}
$$

A.4.18 dropTableDB

Removes a table from the database and removes all data stored in the given table.

$$
\begin{array}{|l}
\textit{dropTableDB} \\
\hline
\Delta DATABASE \\
ts? : TABLESCHEMA \\
\hline
ts? \in schemas \\
referringSchemas(ts?, \theta DATABASE) = \varnothing \\
schemas' = schemas \setminus \{ts?\} \\
values' = values \setminus \{val : DATAVALUES \mid \\
\quad val \in values \land val.definition = ts?\} \\
\hline
\end{array}
$$

A.4.19 insertDataToFKDB

The schema inserts data into a foreign key according to the given mapping.

$$
\begin{array}{|l}
\textit{insertDataToFKDB} \\
\hline
\Delta \mathit{DATABASE} \\
\mathit{fk?} : \mathit{FOREIGNKEY} \\
\mathit{map?} : \mathit{MAPPING} \\
\hline
\forall\, dv, dv' : \mathit{DATAVALUES};\ p : \mathit{MAPPINGPAIR}; \\
\quad fkv : \mathit{FOREIGNKEYVALUE} \bullet p \in map?.pairs \land dv = p.target \land \\
\qquad fkv.value = p.source.key.value \land fkv.definition = fk? \Rightarrow \\
\qquad dv'.foreignkeyValues = dv.foreignkeyValues \cup \{fkv\} \\
\hline
\end{array}
$$

A.4.20 insertDataToMapTableDB

The schema inserts data into a mapping table according to the given mapping.

$$
\begin{array}{|l}
\textit{insertDataToMapTableDB} \\
\hline
\Delta \mathit{DATABASE} \\
\mathit{ts?} : \mathit{TABLESCHEMA} \\
\mathit{map?} : \mathit{MAPPING} \\
\hline
\forall\, dv1, dv2, dv' : \mathit{DATAVALUES};\ p : \mathit{MAPPINGPAIR}; \\
\quad fkv, fkv2 : \mathit{FOREIGNKEYVALUE};\ fk, fk2 : \mathit{FOREIGNKEY} \bullet \\
\quad\; p \in map?.pairs \land dv1 = p.target \land dv2 = p.source \land fk.source = ts? \land \\
\quad\; fk.reference = dv1.definition \land fk2.source = ts? \land \\
\quad\; fk2.reference = dv2.definition \land fkv.definition = fk \land \\
\quad\; fkv2.definition = fk2 \Rightarrow \\
\quad\; fkv.value = dv1.key.value \land fkv2.value = dv2.key.value \\
\hline
\end{array}
$$

A.4.21 next

The function generates a new primary key's value according the *SEQUENCE*.

$$
\begin{array}{|l}
next : \mathit{SEQUENCE} \rightarrow \mathbb{N}
\end{array}
$$

A.5 initSoftware

Creates a new software with initialized application and database.

$$
\begin{array}{|l}
\textit{initSoftware} \\
\hline
\mathit{SOFTWARE} \\
\hline
newDatabase[database/d?] \\
newEntities[entities/e?] \\
\hline
\end{array}
$$

A.6 Transformation Helpers

The section contains queries, transformations and predicates, which are often repeated in the transformations or which are so long that they can obfuscate the meaning of the transformation in which they are used..

A.6.1 initMappingForExtractParent

The schema initialize mapping in the extract parent transformation.

$$
\begin{array}{|l}
\textit{initMappingForExtractParent} \\
\hline
\Xi ENTITIES \\
map! : MAPPING \\
parent?, child? : CLASS \\
tsc, tsp : TABLESCHEMA \\
label : LABEL \\
primKey : PRIMARYKEY \\
atts : \mathbb{P}\, ATTRIBUTE \\
schemas, schemas', tables : \mathbb{P}\, TABLESCHEMA \\
foreignKeys, foreignKeys' : \mathbb{P}\, FOREIGNKEY \\
values, values' : \mathbb{P}\, DATAVALUES \\
sequence, sequence' : SEQUENCE \\
columns : \mathbb{P}\, COLUMN \\
col : COLUMN \\
ts, ts! : TABLESCHEMA \\
l : LABEL \\
constraints : \mathbb{P}\, CONSTRAINT \\
\hline
entityToTableORM[child?/c?, tsc/ts!] \\
entityToTableORM[parent?/c?, tsp/ts!] \\
MappingIsSimple[map!/map?] \land MappingFullComplete[map!/map?] \\
\forall\, dvc : DATAVALUES;\ m : MAPPINGPAIR \bullet \\
\quad dvc.definition = tsc \land m \in map!.pairs \Leftrightarrow \\
\quad m.source = dvc \land \exists\, dvp : DATAVALUES \bullet dvp.key = dvc.key \\
\qquad \land\, dvp.definition = tsp \land m.target = dvp \\
\hline
\end{array}
$$

A.6.2 initMappingForRemoveParent

The schema initialize mapping in the transformation, which removes a parent from a class.

$$
\begin{array}{l}
\textit{initMappingForRemoveParent} \\
\hline
\Xi ENTITIES \\
\Xi DATABASE \\
map? : MAPPING \\
cts? : TABLESCHEMA \\
c? : CLASS \\
name, l, label : LABEL \\
c, d : CLASS \\
ts, sourceSchema, targetSchema : TABLESCHEMA \\
constraints : \mathbb{P}\, CONSTRAINT \\
primKey : PRIMARYKEY \\
columns : \mathbb{P}\, COLUMN \\
tables : \mathbb{P}\, TABLESCHEMA \\
col : COLUMN \\
atts : \mathbb{P}\, ATTRIBUTE \\
\hline
\forall\, mp : MAPPINGPAIR;\ dv : DATAVALUES \bullet mp \in map?.pairs \Leftrightarrow \\
\quad mp.source.definition = dv.definition \land mp.target.key = mp.source.key \land \\
\quad \forall\, cv : COLUMNVALUE;\ fkv : FOREIGNKEYVALUE \bullet \\
\qquad cv \in mp.target.colValues \Leftrightarrow cv.definition \in cts?.columns \land \\
\qquad fkv \in mp.target.foreignkeyValues \Leftrightarrow \\
\qquad\quad \exists\, a : ASSOCIATION;\ fk : FOREIGNKEY \bullet \\
\qquad\qquad a \in associationsOf(c?, \theta ENTITIES) \land assocToFkORM[a/a?, fk/fk!] \land \\
\qquad\qquad fk = fkv.definition \\
\hline
\end{array}
$$

A.6.3 initMappingForSplit

The schema initialize mapping in the split class transformation.

$$
\begin{array}{|l}
\textit{initMappingForSplit} \\
\Xi ENTITIES \\
\Xi DATABASE \\
map! : MAPPING \\
old?, new? : CLASS \\
label : LABEL \\
primKey : PRIMARYKEY \\
atts : \mathbb{P}\, ATTRIBUTE \\
schemas, schemas', tables : \mathbb{P}\, TABLESCHEMA \\
foreignKeys, foreignKeys' : \mathbb{P}\, FOREIGNKEY \\
values, values' : \mathbb{P}\, DATAVALUES \\
sequence, sequence' : SEQUENCE \\
columns : \mathbb{P}\, COLUMN \\
col : COLUMN \\
ts, ts!, oldSchema, newSchema : TABLESCHEMA \\
l : LABEL \\
constraints : \mathbb{P}\, CONSTRAINT \\
\hline
entityToTableORM[old?/c?, oldSchema/ts!] \\
entityToTableORM[old?/c?, newSchema/ts!] \\
\forall m : MAPPINGPAIR \bullet \\
\quad m \in map!.pairs \Rightarrow \\
\quad m.source \in selectAllData(oldSchema, \theta(DATABASE)) \Leftrightarrow \\
\quad m.target.definition = newSchema \land \\
\quad m.target.key = m.source.key \land \\
\quad \forall cv : COLUMNVALUE;\; c : COLUMN \bullet \\
\qquad cv \in m.target.colValues \Leftrightarrow \\
\qquad cv.definition = c \land c \in newSchema.columns \\
\hline
\end{array}
$$

A.6.4 isInstanceOf

The function verifies if a value is an instance of a class.

$$
\begin{array}{|l}
isInstanceOf : DATAVALUES \times CLASS \rightarrow BOOL \\
\hline
\forall dv : DATAVALUES;\; c : CLASS \bullet \\
\quad (isInstanceOf(dv, c) = True \Leftrightarrow \exists cv : COLUMNVALUE \bullet \\
\qquad cv.definition.label = INSTANCEDEF \land cv.value = c.label) \lor \\
\quad (isInstanceOf(dv, c) = False \Leftrightarrow \forall cv : COLUMNVALUE \bullet \\
\qquad cv.definition.label = INSTANCEDEF \land cv.value \neq c.label)
\end{array}
$$

A.6.5 moveAttributes

The transformation moves multiple attributes from a class to its parent.

moveAttributes

$c? : CLASS$
$\Delta SOFTWARE$
$\Delta ENTITIES$
$\Delta DATABASE$
$d : CLASS$
$map : MAPPING$
$poc : ATTRIBUTEOfCLASS$
$to, pts : TABLESCHEMA$
$atts : \mathbb{P}\, ATTRIBUTE$

$d = parentOf(c?, entities)$
$entityToTableORM[d/c?, pts/ts!]$
$\forall\, p : MAPPINGPAIR \bullet$
$\quad p \in map.pairs \Leftrightarrow p.source = p.target \wedge p.source.definition = pts$
$\forall\, p : ATTRIBUTE \bullet$
$\quad p \in attributesOf(c?, entities) \Rightarrow moveAttribute[d/d?, p/p?, map/map?]$

Appendix B

Queries for VCS Manipulation

B.1 initOPERATIONBASEDVCS

The transformation initialize a new operation-based VCS.

$$\begin{array}{|l}
\textit{initOPVCSWITHBRANCHES} \\
\hline
\textit{OPVCSWITHBRANCHES} \\
\textit{ovcs} : \textit{OPERATIONBASEDVCS} \\
\hline
\textit{ovcs.transformations} = \langle\rangle \\
\textit{branches} = \{\textit{ovcs}\} \\
\hline
\end{array}$$

Appendix C

Object-Relational Mapping

C.1 assocToFkORM

The schema maps an association to a foreign key.

$$
\begin{array}{|l}
\textit{assocToFkORM} \\
\hline
\Xi \mathit{ENTITIES} \\
\Xi \mathit{DATABASE} \\
a? : \mathit{ASSOCIATION} \\
\mathit{fk}! : \mathit{FOREIGNKEY} \\
\mathit{name}, l, \mathit{label} : \mathit{LABEL} \\
c, d : \mathit{CLASS} \\
\mathit{sourceSchema}, \mathit{targetSchema}, \mathit{ts} : \mathit{TABLESCHEMA} \\
\mathit{constraints} : \mathbb{P}\, \mathit{CONSTRAINT} \\
\mathit{primKey} : \mathit{PRIMARYKEY} \\
\mathit{columns} : \mathbb{P}\, \mathit{COLUMN} \\
\mathit{tables} : \mathbb{P}\, \mathit{TABLESCHEMA} \\
\mathit{col} : \mathit{COLUMN} \\
\mathit{atts} : \mathbb{P}\, \mathit{ATTRIBUTE} \\
\hline
\mathit{name} = a?.\mathit{label} \\
a?.\mathit{optional} = \mathit{True} \Rightarrow \mathit{constraints} = \{\mathit{NOTNULL}\} \\
a?.\mathit{optional} = \mathit{False} \Rightarrow \mathit{constraints} = \varnothing \\
c = a?.\mathit{source} \\
\mathit{entityToTableORM}[c/c?, \mathit{sourceSchema}/\mathit{ts}!] \\
d = a?.\mathit{target} \\
\mathit{entityToTableORM}[d/c?, \mathit{targetSchema}/\mathit{ts}!] \\
\mathit{initForeignKey}[\mathit{name}/l?, \mathit{constraints}/\mathit{constraints}?, \mathit{sourceSchema}/\mathit{source}?, \\
\quad \mathit{targetSchema}/\mathit{reference}?] \\
\hline
\end{array}
$$

C.2 assocToTableORM

The association maps a a table.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{assocToTableORM}} \\
\Xi DATABASE \\
\Xi ENTITIES \\
a? : ASSOCIATION \\
ts! : TABLESCHEMA \\
fk1, fk2 : FOREIGNKEY \\
l : LABEL \\
primKey : PRIMARYKEY \\
constraints : \mathbb{P}\, CONSTRAINT \\
columns : \mathbb{P}\, COLUMN \\
sourceCLASS, targetCLASS : CLASS \\
sourceSchema, targetSchema : TABLESCHEMA \\
label : LABEL \\
ts, table : TABLESCHEMA \\
tables : \mathbb{P}\, TABLESCHEMA \\
col : COLUMN \\
atts : \mathbb{P}\, ATTRIBUTE \\
\hline
l = a?.label \\
initPrimaryKey[l/l?, primKey/primKey!] \\
a?.optional = True \Rightarrow constraints = \{NOTNULL\} \\
a?.optional = False \Rightarrow constraints = \varnothing \\
columns = \varnothing \\
initTableSchema[l/label?, primKey/primKey?, \\
columns/columns?, table/ts!] \\
sourceCLASS = a?.source \\
targetCLASS = a?.target \\
entityToTableORM[sourceCLASS/c?, sourceSchema/ts!] \\
entityToTableORM[targetCLASS/c?, targetSchema/ts!] \\
initForeignKey[table/source?, sourceSchema/reference?, \\
\quad constraints/constraints?, label/l?, fk1/fk!] \\
initForeignKey[table/source?, targetSchema/reference?, \\
\quad constraints/constraints?, label/l?, fk2/fk!] \\
\hline
\end{array}
$$

C.3 attributeToColumnORM

The schema maps an attribute to column.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{attributeToColumnORM}} \\
p? : ATTRIBUTE \\
col! : COLUMN \\
\hline
col!.label = p?.label \\
col!.type \in DTYPE \\
p?.optional = True \Rightarrow NOTNULL \in col!.constraints \\
\hline
\end{array}
$$

C.4 attributesToDbORM

The schema maps an attribute to a database.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{attributesToDbORM}} \\
\Xi DATABASE \\
columns! : \mathbb{P}\ COLUMN \\
tables! : \mathbb{P}\ TABLESCHEMA \\
attributes? : \mathbb{P}\ ATTRIBUTE \\
col : COLUMN \\
ts : TABLESCHEMA \\
l, label : LABEL \\
primKey : PRIMARYKEY \\
constraints : \mathbb{P}\ CONSTRAINT \\
columns : \mathbb{P}\ COLUMN \\
\hline
\forall p : ATTRIBUTE \bullet (p \in attributes? \land p.upper = One \Rightarrow \\
\quad attributeToColumnORM[p/p?, col/col!] \land col \in columns!) \lor \\
(p \in attributes? \land p.upper = Many \Rightarrow attributeToTableORM[p/p?, ts/ts!] \land \\
ts \in tables!) \\
\hline
\end{array}
$$

C.5 attributeToTableORM

The schema maps an attribute to a table in the database.

$$
\begin{array}{|l}
\textit{attributeToTableORM} \\
\hline
\Xi DATABASE \\
p? : ATTRIBUTE \\
ts! : TABLESCHEMA \\
label : LABEL \\
primKey : PRIMARYKEY \\
constraints : \mathbb{P}\, CONSTRAINT \\
col : COLUMN \\
columns : \mathbb{P}\, COLUMN \\
\hline
label = p?.label \\
initPrimaryKey[label/l?, primKey/primKey!] \\
p?.optional = True \Rightarrow constraints = \{NOTNULL\} \\
p?.optional = False \Rightarrow constraints = \varnothing \\
initColumn[col/col!, constraints/constraints?, label/l?] \\
columns = \{col\} \\
initTableSchema[label/label?, primKey/primKey?, columns/columns?] \\
\hline
\end{array}
$$

C.6 dbNameORM

For each name on the level of entities the functions creates a unique name on the level of database.

$$
\begin{array}{|l}
dbNameORM : LABEL \rightarrow LABEL
\end{array}
$$

C.7 entityToTableNoAttributesORM

The schema maps an entity to a table with no attribute.

$$
\begin{array}{|l}
\textit{entityToTableNoAttributesORM} \\
\hline
\Xi ENTITIES \\
c? : CLASS \\
ts! : TABLESCHEMA \\
label : LABEL \\
columns : \mathbb{P}\, COLUMN \\
primKey : PRIMARYKEY \\
\hline
label = c?.label \\
columns = \varnothing \\
initPrimaryKey[label/l?, primKey/primKey!] \\
initTableSchema[label/label?, primKey/primKey?, columns/columns?] \\
\hline
\end{array}
$$

C.8 entityOutsideHierarchyToTableORM

The schema maps a class which is not in an inheritance hierarchy into a table.

$$
\begin{array}{l}
\textit{entityOutsideHierarchyToTableORM} \\
\hline
\Xi \mathit{ENTITIES} \\
c? : \mathit{CLASS} \\
ts! : \mathit{TABLESCHEMA} \\
label : \mathit{LABEL} \\
atts : \mathbb{P}\ \mathit{ATTRIBUTE} \\
primKey : \mathit{PRIMARYKEY} \\
schemas, schemas', tables : \mathbb{P}\ \mathit{TABLESCHEMA} \\
foreignKeys, foreignKeys' : \mathbb{P}\ \mathit{FOREIGNKEY} \\
values, values' : \mathbb{P}\ \mathit{DATAVALUES} \\
sequence, sequence' : \mathit{SEQUENCE} \\
columns : \mathbb{P}\ \mathit{COLUMN} \\
col : \mathit{COLUMN} \\
ts, ts! : \mathit{TABLESCHEMA} \\
l : \mathit{LABEL} \\
constraints : \mathbb{P}\ \mathit{CONSTRAINT} \\
\hline
(parentOf(c?, \theta(\mathit{ENTITIES})) = \mathit{NULLCLASS} \land \\
children(c?, \theta(\mathit{ENTITIES})) = \varnothing) \Rightarrow \\
\quad label = c?.label \land \\
\quad atts = attributesOf(c?, \theta(\mathit{ENTITIES})) \land \\
\quad initPrimaryKey[label/l?, primKey/primKey!] \land \\
\quad attributesToDbORM[columns/columns!, attributes/attributes?, \\
\quad tables/tables!] \land \\
\quad initTableSchema[label/label?, primKey/primKey?, columns/columns?] \\
\hline
\end{array}
$$

C.9 entityToTableORM

The schema maps an entity to a database table.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{entityToTableORM}} \\
\hline
\Xi \mathit{ENTITIES} \\
c? : \mathit{CLASS} \\
ts! : \mathit{TABLESCHEMA} \\
\mathit{label} : \mathit{LABEL} \\
\mathit{primKey} : \mathit{PRIMARYKEY} \\
\mathit{atts} : \mathbb{P}\, \mathit{ATTRIBUTE} \\
\mathit{schemas}, \mathit{schemas}', \mathit{tables} : \mathbb{P}\, \mathit{TABLESCHEMA} \\
\mathit{foreignKeys}, \mathit{foreignKeys}' : \mathbb{P}\, \mathit{FOREIGNKEY} \\
\mathit{values}, \mathit{values}' : \mathbb{P}\, \mathit{DATAVALUES} \\
\mathit{sequence}, \mathit{sequence}' : \mathit{SEQUENCE} \\
\mathit{columns} : \mathbb{P}\, \mathit{COLUMN} \\
\mathit{col} : \mathit{COLUMN} \\
ts, ts! : \mathit{TABLESCHEMA} \\
l : \mathit{LABEL} \\
\mathit{constraints} : \mathbb{P}\, \mathit{CONSTRAINT} \\
\hline
\mathit{parentEntityToTableORM} \lor \\
\mathit{entityOutsideHierarchyToTableORM} \\
\hline
\end{array}
$$

C.10 ORM

The schema defines how entities are mapped into a database.

$$
\begin{array}{|l}
\textit{ORM} \\
\hline
\Xi \textit{ENTITIES} \\
\Xi \textit{DATABASE} \\
e? : \textit{ENTITIES} \\
d? : \textit{DATABASE} \\
\textit{label}, l, \textit{name} : \textit{LABEL} \\
\textit{primKey} : \textit{PRIMARYKEY} \\
\textit{columns} : \mathbb{P}\ \textit{COLUMN} \\
\textit{tables} : \mathbb{P}\ \textit{TABLESCHEMA} \\
\textit{col} : \textit{COLUMN} \\
\textit{sourceSchema}, \textit{targetSchema}, \textit{table}, \textit{ts} : \textit{TABLESCHEMA} \\
\textit{constraints} : \mathbb{P}\ \textit{CONSTRAINT} \\
c, d, \textit{sourceCLASS}, \textit{targetCLASS} : \textit{CLASS} \\
\textit{fk}1, \textit{fk}2 : \textit{FOREIGNKEY} \\
\textit{atts} : \mathbb{P}\ \textit{ATTRIBUTE} \\
\hline
\forall\, c : \textit{CLASS} \bullet \\
\quad c \in e?.\textit{classes} \Leftrightarrow \\
\quad \exists\, \textit{td}, \textit{td}2 : \textit{TABLESCHEMA} \bullet \\
\quad\quad \textit{td} \in d?.\textit{schemas} \land \textit{entityToTableORM}[c/c?, \textit{td}2/\textit{ts}!] \land \textit{td}2 = \textit{td} \\
\forall\, a : \textit{ASSOCIATION} \bullet \\
\quad a \in e?.\textit{associations} \land a.\textit{upper} = \textit{Many} \Leftrightarrow \\
\quad \exists\, \textit{fk}, \textit{fk}2 : \textit{FOREIGNKEY} \bullet \\
\quad\quad \textit{fk} \in d?.\textit{foreignKeys} \land \textit{assocToFkORM}[a/a?, \textit{fk}2/\textit{fk}!] \land \textit{fk}2 = \textit{fk} \\
\forall\, a : \textit{ASSOCIATION} \bullet \\
\quad a \in e?.\textit{associations} \land a.\textit{upper} = \textit{One} \Leftrightarrow \\
\quad \exists\, \textit{td}, \textit{td}2 : \textit{TABLESCHEMA} \bullet \\
\quad\quad \textit{td} \in d?.\textit{schemas} \land \textit{assocToTableORM}[a/a?, \textit{td}2/\textit{ts}!] \land \textit{td}2 = \textit{td} \\
\hline
\end{array}
$$

C.11 parentEntityToTableORM

The schema maps a top class in an inheritance hierarchy to a table. The child classes are mapped into database as well.

parentEntityToTableORM

$$
\begin{array}{|l}
\Xi ENTITIES \\
c? : CLASS \\
ts! : TABLESCHEMA \\
label : LABEL \\
primKey : PRIMARYKEY \\
atts : \mathbb{P}\, ATTRIBUTE \\
schemas, schemas', tables : \mathbb{P}\, TABLESCHEMA \\
foreignKeys, foreignKeys' : \mathbb{P}\, FOREIGNKEY \\
values, values' : \mathbb{P}\, DATAVALUES \\
sequence, sequence' : SEQUENCE \\
columns : \mathbb{P}\, COLUMN \\
col, col2 : COLUMN \\
ts, ts! : TABLESCHEMA \\
l : LABEL \\
constraints : \mathbb{P}\, CONSTRAINT \\
type : DTYPE \\
\hline
parentOf(c?, \theta(ENTITIES)) = NULLCLASS \\
children(c?, \theta(ENTITIES)) \neq \varnothing \\
classes = \{c : CLASS \mid c \in \operatorname{dom}(childParentRelation(c?, \theta(ENTITIES))^{+})\} \\
label = dbNameORM(c?.label) \\
initPrimaryKey[label/l?, primKey/primKey!] \\
\forall\, a : ATTRIBUTE \bullet \\
\quad a \in atts \Leftrightarrow \exists\, c : CLASS \bullet c \in classes \land a \in attributesOf(c, \theta(ENTITIES)) \\
attributesToDbORM[columns/columns!, atts/attributes?, tables/tables!] \gg \\
[\mid constraints = \{NOTNULL\} \land \\
\quad initColumn[INSTANCEDEF/l?, constraints/constraints?, col2/col!] \land \\
\qquad columns \cup \{col2\}] \gg \\
initTableSchema[label/label?, primKey/primKey?, columns/columns?] \\
\hline
\end{array}
$$

Appendix D

SQL Generated by the MigDb Framework

```
CREATE SEQUENCE public.seq_global START 1;
CREATE TABLE public.country ();
ALTER TABLE public.country
        ADD COLUMN id_country int;
CREATE INDEX IX_country_id_country
        ON public.country (id_country);
ALTER TABLE public.country
        ADD CONSTRAINT PK_country
        PRIMARY KEY (id_country);
ALTER TABLE public.country
        ADD COLUMN name character(30) ;
CREATE TABLE public.legalperson ();
ALTER TABLE public.legalperson
        ADD COLUMN id_legalperson int;
CREATE INDEX IX_legalperson_id_legalperson
        ON public.legalperson (id_legalperson);
ALTER TABLE public.legalperson
        ADD CONSTRAINT PK_legalperson
        PRIMARY KEY (id_legalperson);
ALTER TABLE public.legalperson
        ADD COLUMN regno int;
ALTER TABLE public.legalperson
        ADD COLUMN bizname character(30) ;
```

```
ALTER TABLE public.legalperson
        ADD COLUMN street character(30) ;
ALTER TABLE public.legalperson
        ADD COLUMN city character(30) ;
ALTER TABLE public.legalperson
        ADD COLUMN zip character(30) ;
ALTER TABLE public.legalperson
        ADD COLUMN country int;
ALTER TABLE public.legalperson
        ADD CONSTRAINT FK_country_id_country
        FOREIGN KEY (country) REFERENCES public.country (id_country);
CREATE TABLE public.naturalperson ();
ALTER TABLE public.naturalperson
        ADD COLUMN id_naturalperson int;
CREATE INDEX IX_naturalperson_id_naturalperson
        ON public.naturalperson (id_naturalperson);
ALTER TABLE public.naturalperson
        ADD CONSTRAINT PK_naturalperson
        PRIMARY KEY (id_naturalperson);
ALTER TABLE public.naturalperson
        ADD COLUMN name character(30) ;
ALTER TABLE public.naturalperson
        ADD COLUMN surname character(30) ;
ALTER TABLE public.naturalperson
        ADD COLUMN street character(30) ;
ALTER TABLE public.naturalperson
        ADD COLUMN city character(30) ;
ALTER TABLE public.naturalperson
        ADD COLUMN zip character(30) ;
ALTER TABLE public.naturalperson
        ADD COLUMN country int;
ALTER TABLE public.naturalperson
        ADD CONSTRAINT FK_country_id_country
        FOREIGN KEY (country) REFERENCES public.country (id_country);
CREATE TABLE public.party ();
ALTER TABLE public.party
        ADD COLUMN id_party int;
CREATE INDEX IX_party_id_party
```

```
        ON public.party (id_party);
ALTER TABLE public.party
        ADD CONSTRAINT PK_party
        PRIMARY KEY (id_party);
ALTER TABLE public.party
        ADD COLUMN street character(30) ;
ALTER TABLE public.party
        ADD COLUMN city character(30) ;
ALTER TABLE public.party
        ADD COLUMN zip character(30) ;
ALTER TABLE public.party
        ADD COLUMN country int;
ALTER TABLE public.party
        ADD CONSTRAINT FK_country_id_country
        FOREIGN KEY (country) REFERENCES public.country (id_country);
INSERT INTO public.party (id_party,street,city,zip,country)
        SELECT id_legalperson,street,city,zip,country
            FROM legalperson;
ALTER TABLE public.legalperson
        DROP COLUMN street;
ALTER TABLE public.legalperson
        DROP COLUMN city;
ALTER TABLE public.legalperson
        DROP COLUMN zip;
ALTER TABLE public.legalperson
        DROP COLUMN country;
ALTER TABLE public.legalperson
        RENAME COLUMN id_legalperson TO id_party;
INSERT INTO public.party (id_party,street,city,zip,country)
        SELECT id_naturalperson,street,city,zip,country
            FROM naturalperson;
ALTER TABLE public.naturalperson
        DROP COLUMN street;
ALTER TABLE public.naturalperson
        DROP COLUMN city;
ALTER TABLE public.naturalperson
        DROP COLUMN zip;
ALTER TABLE public.naturalperson
```

```
        DROP COLUMN country;
ALTER TABLE public.naturalperson
        RENAME COLUMN id_naturalperson TO id_party;
CREATE TABLE public.address ();
ALTER TABLE public.address
        ADD COLUMN id_address int;
CREATE INDEX IX_address_id_address
        ON public.address (id_address);
ALTER TABLE public.address
        ADD CONSTRAINT PK_address
        PRIMARY KEY (id_address);
ALTER TABLE public.party
        ADD COLUMN address int;
UPDATE public.party SET address = nextval('seq_global');
ALTER TABLE public.address
        ADD COLUMN street character(30) ;
ALTER TABLE public.address
        ADD COLUMN city character(30) ;
ALTER TABLE public.address
        ADD COLUMN zip character(30) ;
ALTER TABLE public.address
        ADD COLUMN country int;
ALTER TABLE public.address
        ADD CONSTRAINT FK_country_id_country
        FOREIGN KEY (country) REFERENCES public.country (id_country);
INSERT INTO public.address (id_address,street,city,zip,country)
        SELECT address,street,city,zip,country
            FROM party;
ALTER TABLE public.party
        DROP COLUMN street;
ALTER TABLE public.party
        DROP COLUMN city;
ALTER TABLE public.party
        DROP COLUMN zip;
ALTER TABLE public.party
        DROP COLUMN country;
ALTER TABLE public.party
        ADD CONSTRAINT FK_address_id_address
        FOREIGN KEY (address) REFERENCES public.address (id_address);
ALTER TABLE public.party
        RENAME COLUMN address TO residentialaddress;
ALTER TABLE public.party
        ADD COLUMN contactaddress int;
ALTER TABLE public.party
        ADD CONSTRAINT FK_contactaddress_id_country
        FOREIGN KEY (contactaddress)
        REFERENCES public.country (id_country);
```

Appendix E

Case Study in the Java Framework

The case study presented in Sect. 8.2 solved by the implemented Java prototype:

```
migrate new class ~.model.Country --table country
migrate new class ~.model.NaturalPerson --table natural_person
migrate new class ~.model.LegalPerson --table legal_person

// Create Country properties
migrate add id --class ~.model.Country
migrate new property name --propertyType java.lang.String
    --class ~.model.Country --column name
    --columnType varchar2(255)

// Create NaturalPerson properties
migrate add id --class ~.model.NaturalPerson
migrate new property name --propertyType java.lang.String
    --class ~.model.NaturalPerson --column name
    --columnType varchar2(255)
migrate new property surname --propertyType java.lang.String
    --class ~.model.NaturalPerson --column surname
    --columnType varchar2(255)
migrate new property street --propertyType java.lang.String -
    -class ~.model.NaturalPerson --column street
    --columnType varchar2(255)
migrate new property city --propertyType java.lang.String
    --class ~.model.NaturalPerson --column city
    --columnType varchar2(255)
migrate new property zip --propertyType java.lang.String
```

```
    --class ~.model.NaturalPerson --column zip
    --columnType varchar2(255)

// Create LegalPerson properties
migrate add id --class ~.model.LegalPerson
migrate new property regNo --propertyType java.lang.Integer
    --class ~.model.LegalPerson  --column reg_no
    --columnType integer
migrate new property bisName --propertyType java.lang.String
    --class ~.model.LegalPerson --column bis_name
    --columnType varchar2(255)
migrate new property street --propertyType java.lang.String
    --class ~.model.LegalPerson --column street
    --columnType varchar2(255)
migrate new property city --propertyType java.lang.String
    --class ~.model.LegalPerson --column city
    --columnType varchar2(255)
migrate new property zip --propertyType java.lang.String
    --class ~.model.LegalPerson --column zip
    --columnType varchar2(255)

// Split LegalPerson into AddressLegalPerson and LegalPersonParty
migrate split class --class ~.model.LegalPerson
    --classA ~.model.LegalPersonParty --tableA legal_person_party
    --propertiesA id,regNo,bisName
    --classB ~.model.AddressLegalPerson
    --tableB address_legal_person
    --propertiesB id,street,city,zip,country
    --queryA "1 = 1" --queryB "1 = 1"

// Extract  AddressNaturalPerson from NaturalPerson
migrate split class --class ~.model.NaturalPerson
    --classA ~.model.NaturalPersonParty --tableA natural_person_party
    --propertiesA id,name,surname
    --classB ~.model.AddressNaturalPerson
    --tableB address_natural_person
    --propertiesB id,street,city,zip,country
    --queryA "1 = 1" --queryB "1 = 1"
```

```

// Merge AddressLegalPerson and AddressNaturalPerson into Address
migrate merge class --class ~.model.Address --table address
    --classA ~.model.AddressNaturalPerson
    --classB ~.model.AddressLegalPerson
    --query "1 = 1"
migrate make pk --class ~.model.Address --property id

// Introduce parent Party to LegalPerson an NuturalPerson
migrate introduce parent --class ~.model.NaturalPersonParty
    --parent ~.model.Party --parentTable parent
migrate introduce parent --class ~.model.LegalPersonParty
    --parent ~.model.Party

// Pull up common properties of LegalPersonParty
migrate pull up --class ~.model.LegalPersonParty
    --property id --query "1 = 1" --skipDrop

// Pull up common properties of NaturalPersonParty
migrate pull up --class ~.model.NaturalPersonParty
    --property id --query "1 = 1" --skipDrop

// Add address fields
migrate new property residentialAddress
    --propertyType ~.model.Address
    --class ~.model.Party --column residential_address
    --columnType bigint --oneToOne  --refColumn id
migrate new property contactAddress
    --propertyType ~.model.Address
    --class ~.model.Party --column contact_address
    --columnType bigint --oneToOne  --refColumn id

// Add new fields
migrate new property phone --propertyType java.lang.String
    --class ~.model.Party --column phone --columnType varchar2(255)
migrate new property email --propertyType java.lang.String
    --class ~.model.Party --column email --columnType varchar2(255)
```