THESIS ON INFORMATICS AND SYSTEM ENGINEERING C32

Relational and Object-Relational Database Management Systems as Platforms for Managing Software Engineering Artifacts

ERKI EESSAAR

TUT
PRESS

Faculty of Information Technology

Department of Informatics

TALLINN UNIVERSITY OF TECHNOLOGY

Dissertation was accepted for the commencement of the degree of Doctor of Philosophy in Engineering on November 15, 2006.

Supervisor: Prof. Rein Kuusik, Faculty of Information Technology

Opponents: Prof. Bernhard Thalheim, Christian-Albrechts-University Kiel, Computer Science Institute, Germany

Prof. Jüri Kiho, University of Tartu, Estonia

Commencement: December 18, 2006

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any degree or examination.

/Erki Eessaar/

ISSN 1406-4731

ISBN 9985-59-671-4

Table of Contents

INTRODUCTION 6

OBJECTIVES 7
LIMITATIONS 10
OUTLINE OF THE DISSERTATION 11
ACKNOWLEDGEMENTS 13

LIST OF ABBREVIATIONS 15

1 DATA MODELS 16

1.1 IMPORTANT CONCEPTS OF DATA MODELS 16
1.2 COMPARISON METHODS OF DATA MODELS 19
1.3 COMPARISON OF THE DATA MODELS 21
1.3.1 Proposed Method for Comparing Data Models 24
1.3.2 Comparison of Data Types 27
1.3.2.1 Discussion 32
1.3.3 Comparison of Data Structures 35
1.3.3.1 Discussion 39
1.3.4 Comparison of Data Integrity Rules 42
1.3.4.1 Discussion 44
1.3.5 Comparison of Data Operators 46
1.3.5.1 Discussion 51
1.3.6 Observations 53
1.3.7 Metrics Values 55
1.3.8 Orthogonality Principle in Language Design 57
1.4 COMPARISON WITH THE EXISTING SATE OF THE ART 59
1.4.1 Object-Oriented Database Metamodel 59
1.4.2 CIM Database Model 60
1.4.3 Common Warehouse Metamodel 60
1.4.4 SQL:2003 Ontology 61
1.4.5 SQL Definition Schema 66
1.5 SUMMARY 67

2 DATABASE MANAGEMENT SYSTEMS IN EXISTING SOFTWARE ENGINEERING SYSTEMS 69

2.1 REQUIREMENTS TO THE REPOSITORY SYSTEMS 72
2.2 EXISTING SOFTWARE ENGINEERING SYSTEMS 74
2.2.1 Systems that do not Use a DBMS 75
2.2.2 Heterogeneous Systems 77
2.2.3 Systems that Use Only an Engineering DBMS 79
2.2.4 Systems that Use Only a General-purpose DBMS 81
2.2.4.1 Systems that Use a RDBMS 81
2.2.4.2 Systems that Use an OODBMS 84
2.2.4.3 Systems that Use an ORDBMS 84

2.3 PROBLEMS OF USING THE RELATIONAL MODEL AND RDBMS$_{SQL}$S IN ENGINEERING SYSTEMS 86
2.4 SUMMARY 90

3 REPOSITORY DATABASE DESIGN 92

3.1 DESIGN ALTERNATIVES OF DATABASE SCHEMA OF A SOFTWARE ENGINEERING SYSTEM 92
3.1.1 Encapsulated Artifact Types 93
3.1.2 Non-encapsulated Artifact Element Types 95
3.1.3 Encapsulated Artifact Element Types 96
3.1.4 Universal Data Model 97
3.2 CHECKING OF THE WELL-FORMEDNESS RULES 107
3.2.1 Ignore All Well-formedness Rules 108
3.2.2 Automatically Resolve Constraint Violations as they Arrive 108
3.2.3 Disallow an Operation and Inform a User 108
3.2.3.1 Example with SimpleM 109
3.2.3.2 Example with Use Cases 110
3.2.4 Allow Everything Initially and Search Errors Later 111
3.2.4.1 Integrated Approach with Versioning 112
3.3 PRESERVING THE SEMANTICS OF RELATIONSHIPS IN A DATABASE 116
3.3.1 Generalization Relationships 116
3.3.2 Whole-Part Relationships 117
3.3.2.1 Related Works 117
3.3.2.2 Possible Designs 118
3.3.2.3 Choosing Between the Designs 120
3.3.3 Advantages of Attributes with Complex Types in Real Relvars 126
3.3.4 Disadvantages of Attributes with Complex Types in Real Relvars 128
3.4 ADDITIONAL GUIDELINES FOR DATABASE DESIGN 128
3.4.1 The Principle of Orthogonal Design 129
3.4.2 The Extended Principle of Orthogonal Database Design ... 135
3.4.2.1 Discussion and Examples 136
3.4.3 Heuristic Rules for Reducing Data Redundancy within the Value of One Real Relvar 138
3.4.3.1 Discussion and Examples 139
3.4.4 An Example about the Suitability of Relation-Valued Attribute 142
3.5 VIEW TO ORDBMS$_{SQL}$S 145
3.5.1 SQL-specific Solutions 145
3.5.2 Usability of the Designs and Guidelines in ORDBMS$_{SQL}$s .. 145
3.6 SUMMARY 154

4 REPOSITORY SYSTEM WITH A FIXED DATABASE SCHEMA 156

4.1 RELATED WORKS 158
4.2 DESCRIPTION OF THE MODELING LANGUAGE 160
4.3 QUERIES 164
4.3.1 Consistency and Completeness Checks 165
4.4 DISCUSSION AND COMPARISONS 167
4.5 SUMMARY 171

CONCLUSIONS 172

SUMMARY OF CONTRIBUTIONS 172
DIRECTIONS FOR FURTHER RESEARCH 178

KOKKUVÕTE 181

REFERENCES 185

APPENDIX A: SOME PROPERTIES OF EXISTING SOFTWARE ENGINEERING SYSTEMS THAT USE THE HELP OF A DBMS 205

APPENDIX B: SOME SECONDARY CHARACTERISTICS OF WHOLE-PART RELATIONSHIPS 207

APPENDIX C: COMPARISON OF SOME SYSTEMS THAT RECORD MODELS IN A DATABASE 208

APPENDIX D: THE LOCATION OF PROTOTYPE SYSTEM 210

APPENDIX E: CURRICULUM VITAE 211

APPENDIX F: ELULOOKIRJELDUS (CV IN ESTONIAN) 214

INTRODUCTION

The main areas of this study are relational and object-relational data models and their suitability in the systems that help to manage software engineering artifacts. The concept "data model" has two different meanings (it is "construct overload"):

- **Meaning 1**: "An abstract, self-contained, logical definition of the data structures, data operators, and so forth, that together make up the abstract machine with which users interact." (Date, 2003, p.15, 16) Some authors use in this context the concept "database model".
- **Meaning 2**: "A model of persistent data of some particular enterprise." (Date, 2003, p. 16)

An informal explanation is that a data model (meaning 1) specifies the building blocks of databases, the rules how to assemble these blocks and operations that can be performed based on the built-up structures. These blocks, rules and operations do not depend on the enterprises that create and maintain the databases. A data model (meaning 2) specifies a structure and constraints of a database of a particular enterprise. In this work, we use the concept "data model" in the sense of *meaning 1,* if not explicitly stated otherwise.

We also note that similarly to the book of Date (2003) we treat the terms "data" and "information" as synonyms in this work.

Edgar F. Codd is the author of the seminal work (Codd, 1970) about the principles of relational data model. Nowadays Relational Database Management System (RDBMS) is a popular type of DBMSs. These systems use language that conforms more or less to the SQL standard. SQL and systems that use it apply many (but not all) ideas of E. F Codd and others about the relational model.

Many researchers and developers claim, despite the success of RDBMSs, that these systems are not suitable for some types of applications. These applications use data that has a complex structure. An example of such a system is a repository system which supports a software development process by helping to record, retrieve, check and reuse different kinds of software engineering artifacts. Repository system is a kind of a software engineering system (SES). Repository manager that is a component of a repository system, provides services for recording, retrieving, and managing objects in a repository and therefore must offer functions of a DBMS and additional functions according to Bernstein and Dayal (1994). A DBMS has an underlying data model that determines how easy it is to create and extend a system that uses a database. For example, a software engineering system (like any other system) can take advantage of a data model that allows creation of new data types, handles missing information properly and permits creation of complex queries and declarative constraints that implement well-formedness rules.

The *motivation* of this dissertation is rooted in the widespread opinion that the relational data model is not powerful enough to be used in software engineering systems or other systems that have to perform sophisticated operations with complex data. Is the relational model useful, but outdated model, or is it still relevant and provides basis for creating complex systems now and in the future? The motivation of this work is to find answers to these questions. In the latter case, it is time to rediscover the relational model.

Objectives

It is a widely accepted position that the underlying data model of SQL:1992 or earlier versions of the SQL standard *are* the relational model and therefore shortcomings and inefficiencies of SQL and DBMSs that use it are actually shortcomings and inefficiencies of the relational model (Eessaar, 2006c). It currently leads to a widespread opinion that the relational data model is not powerful enough in order to build software engineering systems on top of a DBMS that uses this model. For example, Halpin (2001, p. 709) writes: "Relational DBMSs are suitable for about 90 percent of business applications, but may prove inefficient for structurally complex applications such as CASE tools and VLSI design." It is important to note that RDBMS in this case is a system, which uses a database programming language that conforms to SQL:1992 or earlier versions of the SQL standard. We refer to these kinds of systems by using the abbreviation "$\text{RDBMS}_{\text{SQL}}$" from now on. The Third Generation Database System manifesto (Stonebraker et al., 1991) calls these systems second generation systems.

However, there are researchers who do not agree with the view that the relational model is not suitable in certain cases. Barghouti et al. (1996) present requirements to the Process-Centered Software Engineering Environments. They evaluate suitability of $\text{RDBMS}_{\text{SQL}}$ products to implement this kind of environment and conclude: "The other requirements may not be satisfied completely by commercially-available RDBMSs, due in many respects to the limitations of the current SQL standard, SQL-89." Barghouti et al. (1996) also add: "However, there is nothing intrinsic in the relational model that prohibits the extension of RDBMSs to satisfy these requirements." They note that RDBMSs as well as the SQL standard have evolved over the course of time.

Examples of the deficiencies of the underlying data model of $\text{RDBMS}_{\text{SQL}}$s:

1. Impossible to declare new data types.
2. Too big distinction between base- and derived tables.
3. Limited means for presenting missing information.
4. Complex language structure that allows us to solve some problems in many different ways but at the same time does not help to solve some other problems at all. For example, options for making queries based on the hierarchic or networked data are limited.

Are we trying to show that Mr. Halpin and other respectable researchers have reached to the wrong conclusions? On the contrary, important question is:

"What is relational data model and what are its components?" Some authors (Pascal, 2000), (Date and Darwen, 2000), (Date and Darwen, 2006), (Date et al., 2003), (Date, 2003) have concluded that SQL is an incomplete, inefficient and imprecise implementation of the relational data model. Therefore, systems that take advantage of SQL are not as powerful and flexible as they could be.

Many authors have proposed to use in the software engineering systems DBMSs that are built up based on some other data model than relational model (see Chapter 2). For example, Atkinson et al. (1989) present The Object-Oriented Database Systems Manifesto that attempts to define object-oriented database systems and their underlying data model.

In this dissertation, we are interested in the so-called Object-Relational DBMSs (ORDBMSs) and their underlying data models. There are many proposals about what should be the exact nature of these systems (Stonebraker et al., 1991; Seshadri, 1998; Date and Darwen, 2000). In general, they should combine features of the relational model (as interpreted by SQL:1992 or earlier standards) and object-oriented programming languages. In this dissertation, we deal with the two object-relational data models – OR_{SQL} and OR_{TTM}.

The SQL:1999 and SQL:2003 standards try to resolve some of the problems of an early SQL by providing additional features (by *extending* it). For example, they permit creation of new data types. It is said that these standards support object-relational paradigm (Calero et al., 2006). We refer to the systems that follow SQL:2003 (or its predecessor SQL:1999) standard by using the abbreviation "$ORDBMS_{SQL}$" from now on. We call the data model that is used by the $ORDBMS_{SQL}$s as "the OR_{SQL} data model" or "OR_{SQL}".

"The Third Manifesto is a detailed, formal and rigorous proposal for the future directions of data and database management systems (DBMSs for short)." (Date and Darwen, 2000, p. 3) It advocates the *relational data model as basis for future systems*. According to this approach, all the good features that are expected from object-relational data model can actually be implemented within the framework of the relational model. In particular, the support to complex data types is already present in the relational model in the form of domains (Date, 2003). Current standards and systems do not take all the principles of the relational model into account and it causes calls to *extend* the model (with possibly unnecessary features) or even abandon the relational model. The Third Manifesto can be seen as a compilation of principles of ORDBMS that is free from the burdens of SQL. ***"Accordingly, we also believe that a true object/relational system would be nothing more nor less than a true relational system – which is to say, a system that supports the relational model, with all that such support entails." (Date, 2003, p. 861)*** The authors of the manifest claim that they are not extending or replacing the relational model. "Thus, we regard our Manifesto as being very much in spirit of Codd's original work and continuing along the path he originally laid down." (Date and Darwen, 2000, p. xiv) We refer to the data model that is advocated by The Third Manifesto as "the OR_{TTM} data model" or "OR_{TTM}". We refer to The Third Manifesto compliant DBMSs by using the abbreviation "$ORDBMS_{TTM}$" from now on.

Next, we present the objectives of this dissertation.

- ***Objective 1**: To present metamodel-based comparison of the OR_{SQL} and OR_{TTM} data models.*

Data model is an abstract language and it is possible to create its metamodel. Metamodel is "a model of a model" that provides "the rules/grammar for the modelling language (ML) itself." (Henderson-Sellers, 2003)

Software engineering system is an example of a system that manages data that has complex structure. Many software engineering systems are file-based systems that do not use the services of a DBMS. Maybe it is consistent with the results of existing research and usage of DBMSs in this kind of systems is not advantageous?

- ***Objective 2**: To find out what are the problems of using RDBMSs or ORDBMSs in the software engineering systems according to the existing research literature.*

This investigation also helps to achieve the following objective:

- ***Objective 3**: To describe the design alternatives of databases of software engineering systems that will be implemented by using an ORDBMS.*

We present a sample software engineering system that uses an $ORDBMS_{SQL}$ in order to manage software engineering artifacts. This system uses some of the design ideas that are explained in this dissertation.

- ***Objective 4**: To demonstrate that the OR_{SQL} data model has shortcomings that cause difficulties in using the standard-compliant DBMSs in the software engineering systems.*

In addition, current DBMSs do not implement SQL in the full extent that often makes implementation of the software engineering system even more difficult. For example, possibilities to declare constraints are limited and updateable views have additional restrictions.

- ***Objective 5**: To demonstrate that the gap between the principles of OR_{SQL} (theory) and the actual implementation (practice) in current $ORDBMS_{SQL}$s causes additional problems to the designers of software engineering systems.*
- ***Objective 6**: To demonstrate that the data model that is specified in The Third Manifesto (OR_{TTM} data model) is a suitable basis for a DBMS so that this DBMS can be used in a software engineering system.*

Emmerich (1995) and Barhouti et al. (1996) present somewhat similar research. They investigate possibilities of using $RDBMS_{SQL}$s or Object-Oriented DBMSs in Process-Centered Engineering Environments. One difference with our work is that they do not consider ORDBMSs. Secondly, they perform their evaluation based on the commercial DBMS products. This research, on the other hand, investigates suitability of different object-relational *data models* for the management of software engineering artifacts. *In addition*, it referes to the problems of existing DBMSs.

Metamodels of languages, based on which software engineering systems database structure is created, often contain whole-part and generalization relationships. A DBMS is able to understand and enforce structural and operational properties of those relationships and objects that participate in these relationships (Zhang et al., 2001). The underlying data model of a DBMS determines the extent of these abilities. Therefore, properties of the data model determine how well a DBMS can capture knowledge about the real world entities and their behaviour. There exist proposals about how to preserve the semantics of generalization relationships in an ORDBMS$_{TTM}$ database but we are not aware of such work about the whole-part relationships.

- ***Objective 7****: To propose a set of designs for preserving the semantics of whole-part relationships in a database that is created by an ORDBMS$_{TTM}$ and guidelines explaining when to use these designs.*

In this dissertation we use the concept **"complex data type"** (or "complex type") in order to refer to: (a) relation types, (b) tuple types, (c) scalar types where the possible representation has more than one component, (d) scalar types where the possible representation has one component but this component has one of the types (a)-(c). We intend to show that the use of these kinds of types in the *real* relvars (tables) makes database design actually more complex. One reflection of that is the necessity to extend the *Orthogonal Database Design Principle.* This principle helps to prevent data redundancy across different relvars in a database.

- ***Objective 8****: To extend the Principle of Orthogonal Database Design (Date and McGoveran, 1994) so that it would take into account the use of real relvars that have attributes with complex types.*

We use the concept "relvar attribute" in order to refer to an attribute that is specified in the heading of relation type of a relvar.

Limitations

In this dissertation, we are not considering all the data models and their associated DBMSs, but only two object-relational data models and corresponding DBMSs. Examples of data models that are not under the evaluation in this dissertation are: hierarchical, network, TransRelational data model or object-oriented data models. There are many proposals about object-oriented data models in literature according to Atkinson et al. (1989).

One limiting factor of the research is that we do not have final versions of SQL:1999 and SQL:2003 standard documents at our disposal. We use information from the manual of Gulutzan and Pelzer (1999) in order to gain information about SQL:1999 and pre-publication version of SQL:2003 (Melton, 2003), (Melton, 2003b) (Melton, 2003c).

Information captured in the artifact that is recorded in a repository can be presented to the user in more than one way using graphical and textual notations. This dissertation concentrates on the management of the informational content of the artifacts and does not address the issues of the visualization of the artifacts.

One limitation of this study is that the system that is introduced in Chapter 4 is implemented partially and only by using an $\text{ORDBM}_{\text{SQL}}$. This dissertation is not accompanied with an implementation of a software engineering system that uses an $\text{ORDBMS}_{\text{TTM}}$.

Outline of the Dissertation

Figure 1 presents an overview of the structure of this dissertation.

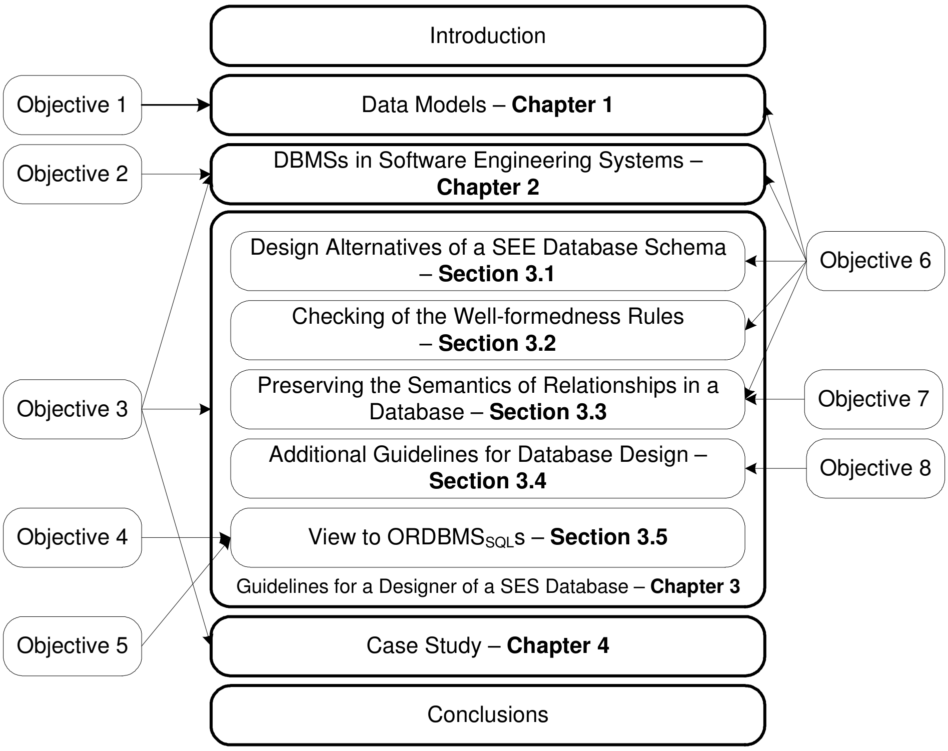

Introduction
Objective 1
Data Models – Chapter 1
Objective 2
DBMSs in Software Engineering Systems – Chapter 2
Design Alternatives of a SEE Database Schema – Section 3.1
Objective 6
Checking of the Well-formedness Rules – Section 3.2
Objective 3
Preserving the Semantics of Relationships in a Database – Section 3.3
Objective 7
Additional Guidelines for Database Design – Section 3.4
Objective 8
Objective 4
View to ORDBMS$_{\text{SQL}}$s – Section 3.5
Guidelines for a Designer of a SES Database – Chapter 3
Objective 5
Case Study – Chapter 4
Conclusions

Figure 1 Overview of the dissertation structure that relates the objectives of the dissertation with the chapters, which contribute to their accomplishment

In **Chapter 1**, we describe "data models" in general (see **section 1.1**) and in particular, two data models that have their roots in the relational model introduced by E. F. Codd. These data models are underlying data model of SQL:2003 (the OR_{SQL} data model) and relational data model as defined by The Third Manifesto (the OR_{TTM} data model). We describe these two models in the form of metamodel-based comparison (see **section 1.3**).

One purpose of this chapter is to give sufficient basis for further discussions by presenting and comparing the data models. This chapter extends the work of Calero et al. (2006) who propose an ontology of SQL:2003 Object-Relational features. In addition, metamodel of the OR_{TTM} data model is a novel result of this work. Other novel results are: proposal of the metamodel-based comparison method of data models, actual comparison of the two data models (OR_{SQL} and OR_{TTM}), metrics values that are calculated based on the metamodels, and findings of violations of the orthogonality principle in OR_{SQL}. We also refer to the shortcomings of the work of Calero et al. (2006) (see **section 1.4**).

Software engineering artifacts are created using a wide range of languages and tools. "Software Engineering System" (SES) is a class of complex systems which members assist their users during development of a software or information system. A SES could be a stand-alone tool or an *environment,* which is a collection of integrated tools (Harrison et al., 2000). A SES is usable in one or more development phases and helps to manage one or more types of software engineering artifacts. For example, it could be a CASE or a Meta-CASE tool, a pattern-based code generator, a web-based collaborative modeling environment or a reuse repository of software engineering artifacts. These systems have to record data (including artifacts) somewhere, just like business applications. Their developers can choose between different implementation strategies or combinations of them: (1) to use an existing DBMS; (2) to use an existing repository system; (3) to build a data management component from scratch. The results of **sections 1.1**, **1.2** and **1.3.4** have been accepted to be published in (Eessaar, 2006h). The results of **sections 1.3.2**, **1.3.7** and **1.3.8** have been accepted to be published in (Eessaar, 2007).

A SES is a good example of a complex system that demands a lot from a DBMS. In **Chapter 2**, we present a literature-based overview of SESs that use a DBMS (see **section 2.2**). We are most interested in systems that use a RDBMS or an ORDBMS because we think that existing *overview papers* about SESs do not pay enough attention to them. Papers about SESs often refer to the problems of $RDBMS_{SQL}$s and their underlying data model. This means that we need a better data model and DBMSs that use this model. Different papers refer to different problems. Our research of SESs helps to compile the thorough list of problems (see **section 2.3**). This part of the work is based on the first part of paper of Eessaar (2006c). We also present requirements to repository systems (see **section 2.1**) because a DBMS that is used in order to implement a SES must provide the technical means that help to fulfil at least some of these requirements.

Quite a lot of researchers and developers have proposed to use $ORDBMS_{SQL}$s in the SESs. Unfortunately they pay little attention to the possible problems of this approach. Therefore, **Chapter 2** presents actually the context of **Chapter 3**. **Chapter 3** contains guidelines for the design of a SES database. Firstly, we investigate how it is possible to implement these guidelines in an $ORDBMS_{TTM}$ database. In particular, this chapter describes approaches for recording artifacts in a database (see **section 3.1**), checking the well-formedness of artifacts (see **section 3.2**) and versioning of artifacts (see **section 3.2.4.1**). These sections (except discussion of "universal design") are based on the papers of Eessaar (2005a, 2006c).

Repository structure is worked out based on the metamodels of the languages (UML, Pattern and Component Markup Language etc.) that are used in order to create artifacts. These metamodels contain many generalization and whole-part relationships. **Section 3.3** investigates how it is possible to preserve semantics of this kind of relationships in an ORDBMS database. A lot of work about preserving relationship semantics in an $\text{ORDBMS}_{\text{SQL}}$ database has been done by different researchers. Therefore, we refer to the existing research and concentrate to the investigation, how to preserve relationship semantics in an $\text{ORDBMS}_{\text{TTM}}$ database. There are works that describe how to handle generalization relationships in an $\text{ORDBMS}_{\text{TTM}}$ database (see **section 3.3.1**). Therefore, we give a short overview about that work and concentrate to the whole-part relationships (see **section 3.3.2**). **Section 3.3.2** is mostly based on the paper of Eessaar (2006g) and **sections 3.3.3-3.3.4** are based on the papers of Eessaar (2006d, 2006e).

Application of the principle of *Orthogonal Database Design* (Date and McGoveran, 1994) helps to achieve a better repository database structure by avoiding data redundancy across the values of different relvars (tables). **Section 3.4** presents the *extended* Principle of Orthogonal Database Design, which takes into account the use of complex data types in a database. In addition, it presents two additional heuristic rules about avoiding redundancy within the value of one relvar (table). This part of the work is based on the papers of Eessaar (2006a, 2006b). We have improved the wording of the principle and rules compared to the work of Eessaar (2006a, 2006b).

Finally, in **section 3.5** we investigate the problems that come up if we try to implement in an $\text{ORDBMS}_{\text{SQL}}$ database the designs and guidelines that are presented in the previous sections of this chapter. As a result, we can refer to the problems of the OR_{SQL} data model and $\text{ORDBMS}_{\text{SQL}}$s that make their use in the software engineering systems (and other systems as well) more difficult. This part of the work is based on the sections of the papers of Eessaar (2005a, 2006c, 2006d, 2006e, 2006g). Findings of **sections 3.3 – 3.5** are applicable in the design of any database, including a repository database.

Chapter 4 contains a description of a web-based system analysis environment that provides queries in order to find violations of well-formedness rules. This part of the work is based on the paper of Eessaar (2006f).

Conclusions of the dissertation are given after **Chapter 4**. We give summary of the work that has been done and describe directions of future work. Conclusions in English language are followed by the conclusions in Estonian language.

Acknowledgements

Above all, I thank my supervisor, Prof. Rein Kuusik for the assistance in the preparation of this dissertation.

I am thankful for Doctoral School in Information and Communication Technology (IKTDK) and Estonian Information Technology Foundation

(EITSA) for supporting participation in the different international conferences. Feedback from these conferences allowed me to improve and extend this work. I express my thanks to Reet Elling who helped to organize trips to the conferences. I am grateful to Tarmo Veskioja and Mare-Anne Laane who have improved the grammar of some parts of this work. I also thank all my colleagues in Department of Informatics in general because they have created atmosphere that encourages learning and researching. I thank my student Erko Aaberg, who helped to build a prototype of the system that is presented in Chapter 4.

Finally, I thank my mother Eda for encouraging me during all these years. Without her support and love, I would not be the person who I am now.

LIST OF ABBREVIATIONS

BNF	- Backus Naur Form
BLOB	- Binary Large Object
CASE	- Computer Aided Software Engineering
CC check	- Consistency and completeness check
CIM	- Common Information Model
CLOB	- Character Large Object
CWM	- Common Warehouse Metamodel
DBMS	- Database Management System
DDL	- Data Definition Language
DML	- Data Manipulation Language
EAV/CR	- The entity-attribute-value representation with classes and relationships
EDBMS	- Engineering Database Management System
EF	- Enterprise Factory
OCL	- Object Constraint Language
ORDBMS	- Object Relational Database Management System
$\text{ORDBMS}_{\text{SQL}}$	- Object Relational Database Management System that conforms to SQL:1999 or later versions of the SQL standard.
OR_{SQL}	- Underlying data model of SQL:1999 or later versions of the SQL standard
$\text{ORDBMS}_{\text{TTM}}$	- Object Relational Database Management System that conforms to the prescriptions, proscriptions and suggestions of The Third Manifesto
OR_{TTM}	- Relational data model as defined by The Third Manifesto
PSEE	- Process-Centered Software Engineering Environment
POOD	- Principle of Orthogonal Design
QIP	- Quality Improvement Paradigm
$\text{RDBMS}_{\text{SQL}}$	- Relational Database Management System that conforms to SQL:1992 or earlier versions of the SQL standard
SES	- Software Engineering System
SQL	- Structured Query Language
SQL:1999	- (International Organization for Standardization) standard ISO/IEC 9075:1999 Database Language SQL
SQL:2003	- (International Organization for Standardization) standard ISO/IEC 9075-2003 Database Language SQL
UDF	- User-Defined Function
UDT	- User-Defined Type
UDST	- User-Defined Structured Type
UDR	- User-Defined Routine
UML	- Unified Modeling Language
XMI	- XML Metadata Interchange

1 DATA MODELS

This chapter contains an overview of data models in general and object-relational data models in particular.

1.1 Important Concepts of Data Models

CIM (Common Information Model) Database Model (DMTF CIM, 2006) is a conceptual model that describes common database management concepts. However, it models the concept "data model" only as an experimental property *DataModelType* of class *CommonDatabase*. We think that it is necessary to model this concept more precisely and present the domain model (see Figure 2). The classes with grey background are already present in CIM Database Model. The new classes are with white background.

A programming language is a formal language designed specifically for machine processing (Greenfield et al., 2004, p. 279). A data model is a kind of an abstract programming language (Date, 2003, p. 16) (see Figure 2) that specifies the data structures and operators, which are its structural and behavioural components, respectively. In addition, a data model specifies "a collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both" (Codd, 1981).

A Database Management System (Database System) (DBMS) is a software system used for managing databases. A user can interact with it by using a database programming language (DPL) that is designed according to some data model. A specification of a formal language, like modeling or programming language, must contain specifications of abstract syntax, semantics and concrete- and serialization syntaxes (Greenfield et al., 2004). The data model is the basis for the abstract syntax of a DPL. A database programming language has two sublanguages – a Data Definition Language (DDL) and a Data Manipulation Language (DML). Statements of a DDL are used in order to create data structures, operators and integrity rules that are prescribed by its underlying data model. Statements of a DML are used in order to perform operations with data.

A database can be divided into conceptual, external and internal levels according to ANSI/SPARC architecture (Date, 2003, p. 34). *Ideally*, a data model specifies structures, operators and constraints that belong to the logical levels - conceptual and external level (and not elements at the internal level). In addition, a DBMS should provide a *storage structure definition language* (SCDL) for managing storage structures at the internal level (Date and Darwen, 2006). In practice, there is often no separate SCDL. Instead, it is possible to specify elements of the internal level (indexes, tablespaces, clusters, segments etc.) and other properties of data storage by using DDL statements.

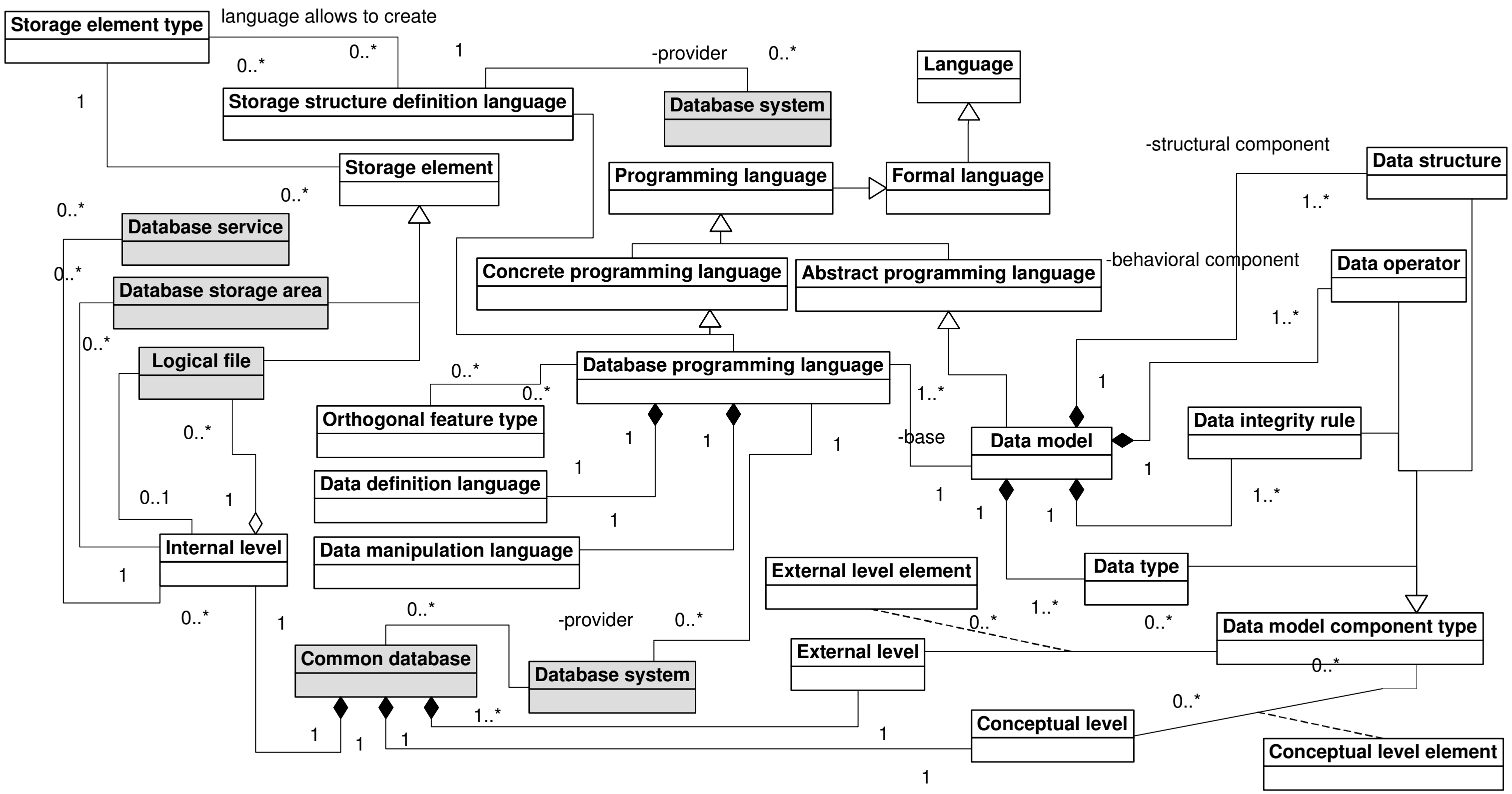


Figure 2 Domain model of data models (desired state of affairs)

Database programming languages provide features that are independent of a data model. The existence of these *orthogonal* features does not depend on the underlying data model of a database language and they could be present in many languages that have different underlying models. Examples of these orthogonal features are the support to the nested transactions (Date and Darwen, 2000, p. 195) or security mechanisms (for example, a possibility to specify roles, users and their privileges in a database). A data model can have more than one corresponding database programming languages. Different languages could provide support to different orthogonal features. For example, The Third Manifesto that is a proposal for future database systems uses the language name "D" in order to refer to any language that follows its principles. The manifest book also presents *Tutorial D* language that is: "a computationally complete programming language with fully integrated database functionality" (Date and Darwen, 2000). Nevertheless, the authors acknowledge that their proposed language is a "toy" language that must help learning. Industrial-strength languages would need additional features.

If we want to compare data models and reason about them, then we must have their specifications at our disposal. The relational model is an example of the data model that was formally specified before the appearance of systems that implemented it (Codd, 1981). Sometimes a data model is formally specified only after its implementations (DBMSs) have been created. This is, for example, true in case of hierarchic and network data models (Codd, 1981).

Nowadays object-relational data models are of major interest. The SQL:1999 and SQL:2003 standards specify the object-relational database programming *language*. We think that these specifications do not contain a clear and compact description of an object-relational data model. Melton (2003b) writes: "The structure of the Definition Schema is a representation of the data model of SQL." However, the specification of *Definition Schema* consists of DDL statements and short textual descriptions of the columns of tables in this schema (310 pages long) (Melton 2003c). Explanations that are more thorough are in the framework part (88 pages long) (Melton, 2003b) and foundation part (1332 pages long) (Melton, 2003).

The Third Manifesto, on the other hand, specifies the OR_{TTM} data model *and* the database programming language (Tutorial D) that is created based on this model. It presents a compact form of 58 prescriptions, proscriptions and suggestions with 11 pages (Date and Darwen, 2000). It distinguishes the issues that are associated with the relational model and the issues that are orthogonal to it.

An abstract syntax of a language describes its elements and rules about their interconnections (Greenfield et al., 2004). It is possible to use context-free grammars or metamodels in order to describe the abstract syntax (Greenfield et al., 2004). For example, context-free grammars are used in order to present the syntax of SQL and Tutorial D. The syntax is expressed by using a form of Backus-Naur Form (BNF) notation. Chaudhuri and Weikum (2000) write that "Understanding semantics of SQL (not even of SQL-92), covering all combinations of nested (and correlated) subqueries, null values, triggers, ADT functions, etc. is a nightmare." We need better ways how to present the data model to the interested readers.

People can benefit from a visual presentation of a concrete syntax of a programming language (Braz, 1990). Is it possible to specify an abstract syntax of a language by using visual means? Specification of the Unified Modeling Language (UML) (OMG formal/03-03-01) is an example of using a metamodeling approach in order to define an abstract syntax of a language. Metamodel "makes statements about what can be expressed in the valid models of a certain modeling language." (Seidewitz, 2003) If we use UML in order to create a metamodel, then the following is true: "A metamodel characterizes language elements as classes, and relationships between them using attributes and associations." (Greenfield et al., 2004, p. 289) It is possible to create metamodels by using other languages as well.

Other examples of using metamodeling approach are the metamodel-based comparison of workflow management systems (Mühlen, 1999) and ontologies (Davies et al., 2003) and description of Object Constraint Language (OCL) by Richters and Gogolla (1999). Habela (2002) presents the metamodel of the object-oriented database management systems. Calero et al. (2006) present the ontology of SQL:2003 Object-Relational Features by using UML class diagrams and well-formedness rules written in OCL.

1.2 Comparison Methods of Data Models

Applications that use databases become increasingly complex and they demand more and more from the DBMSs. A very important selection criterion of a DBMS is its underlying data model. How should we compare data models? The work of Codd and Date (1975) is an example of a thorough and methodical comparison of two data models. They present similarities and differences of *relational* and *network* data model in the form of discussion and examples. They even had to work out definitions of concepts of the network data model based on CODASYL DBTG language proposals in order to do it properly. Date and Codd (1975) compare the use of relational and network databases by the applications. Additional examples of comparisons are the comparison of the prescriptions, proscriptions and suggestions of The Third Manifesto with SQL (Date and Darwen, 2000, Appendix H), and with ODMG proposal of object model and associated database language (Date and Darwen, 2000, Appendix I).

Lack of clear and compact specifications of data models means that often they are compared, evaluated or judged based on DBMSs (see section 2.3). Inadequacies and shortcomings of the DBMSs can cause unfair criticism of a data model. A more precise method for evaluating data models is needed.

Siau and Rossi (1998) introduce and classify the methods for evaluating existing *information modeling methods* (we could also use the concept "data modeling"). The ideas behind the comparison methods, described by Siau and Rossi (1998), can be used in order to compare different *data models*. A comparison method is either empirical or non-empirical. Examples of empirical methods are surveys, laboratory and field experiments, case studies and action research. Next, we describe possible non-empirical methods.

Feature comparison. Data models can be compared with each other based on the features that they provide to the database designers. For example, Date and Darwen (2000, Appendix H) compare the relational model as presented in The Third Manifesto with the underlying model of SQL. The Third Manifesto is well-structured for making such comparisons because it consists of the sets of prescriptions, proscriptions and suggestions, each of which can be seen as a feature or a set of features. One could also create a checklist of the desired features and compare data models with this list. Codd (1981) names components of a data model and notes that comparisons of data models often ignore operators and integrity rules and therefore "run the risk of being meaningless".

The following methods require metamodels of data models.

Comparison based on metamodels. Instead of comparing "features" that are extracted from probably long and vague specifications based on subjective decisions, data models can be compared by finding common metamodel elements as well as elements that have no counterpart in another metamodel or that have more than one counterpart.

Comparisons based on the metrics values that are calculated based on the metamodels. For example, Rossi and Brinkkemper (1996) propose the set of metrics for comparing systems development methods and techniques. Therefore, if the metamodels of data models are available, then these metrics can be used in order to compare the data models.

Ontological evaluation. Chandrasekaran et al. (1999) write: "First of all, ontology is a representation vocabulary, often specialized to some domain or subject matter. More precisely, it is not the vocabulary as such that qualifies as an ontology, but the conceptualizations that the terms in the vocabulary are intended to capture." Ontological evaluation of a language is a comparison of the concrete metaclasses of a language metamodel (language constructs) with the concepts of an ontology in order to find ontological discrepancies: construct overload, construct redundancy, construct excess and construct deficit (Opdahl and Henderson-Sellers, 2002). For example, Opdahl and Henderson-Sellers (2002) have performed an evaluation of UML by comparing it with Bunge–Wand–Weber (BWW) model of information systems.

Are there any ontologies about databases? CIM (Common Information Model) Database Model (DMTF CIM, 2006) is a conceptual model that describes common database management concepts. These concepts correspond mainly to the internal (storage) level of ANSI/X3/Sparc DBMS Framework. Examples of the classes in this model are *LogicalFile*, *SystemResource*, *DatabaseServiceStatistics*. A data model specifies constructs that are used in order to build up a conceptual and external level of DBMS. In addition, the CIM Database model specifies components of SQL Schema. It is part of the specification of one data model but not databases in general.

Date and Darwen (2000, 2006) and Date (2003) use a set of core concepts ("type", "value", "variable", "operator") as a basis of the description of the OR_{TTM} data model. They do not present their research result as ontology but we believe that these core concepts should be part of an ontology that describes the most basic concepts of conceptual and external level of a database, independent of any specific data model.

The following brief overview (see also Figure 3) is based on the work of Date (2003). "A *value* is an individual constant which has no location in time or space." (Date, 2003) A value has at least one *appearance* (representation) that uses some encoding and therefore appearances do have locations in time and space. "A *variable* is a holder of the appearance of a value." (Date, 2003, p. 113) A variable has one value at a time, but its value can be replaced with another value – in other words, variables can be *updated*. Values and variables have *types*. An *operator* returns a value or updates a variable. A variable can only have a value that has the same type as the variable. Each data value has a type. Each parameter of an operator has a type. The result of a read-only operation has a type.

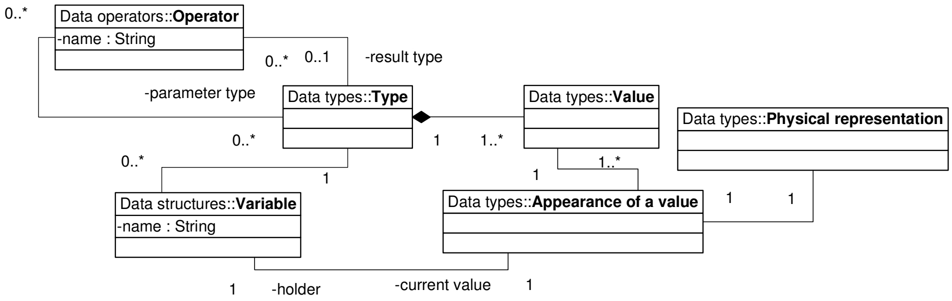


Figure 3 Some basic underlying concepts of OR_{TTM}

1.3 Comparison of the Data Models

Bećarević and Roantree (2004) write: "Object-relational databases do not have a standardised metamodel". This section contains metamodels of two object-relational data models as well as their *metamodel-based comparison*:

1. The data model that is described in The Third Manifesto (OR_{TTM}) (Date and Darwen, 2006).
2. The underlying data model of SQL:2003 (OR_{SQL}) (Melton, 2003).

The presented metamodels do not cover completely The Third Manifesto and SQL:2003 but should be thorough enough in order to compare the OR_{TTM} and OR_{SQL} data models. These metamodels should be seen as the first step towards creating complete metamodels. We think that the most appropriate creators of a metamodel are the designers of a data model.

The advantages of metamodels of data models:

1. Creation of a metamodel may cause actual specification of a data model. For example, there is no clear and compact specification of the *OR_{SQL} data model*. Instead, there is a large textual specification of the SQL database language. A foundation part of SQL:2003 (Melton, 2003) is 1332 pages long.
2. A metamodel visualizes underlying concepts of a data model. It is possible to get an overview about a data model with the help of much more compact document compared to purely textual specification.
3. If we create a metamodel by using some visual language (like UML) that is well known to the software engineering community, then it facilitates understanding of the data models among many professionals. Maybe it also helps to improve understanding of data models by the DBMS vendors and improve current DBMSs (see examples of problems in section 3.5.2).
4. A metamodel can be used for the teaching purposes. "A concept map is a graphical node-arc representation of the relationships among a collection of concepts." (Turns et al., 2000) Ferguson (2003) demonstrates that "UML class diagrams can be used as a concept-mapping tool". A metamodel of a data model can be used as a concept map in order to give visual overview of the data model constructs and their relationships. There is already research how to use concept maps in order to communicate information, create instructional materials and assess the students (Turns et al., 2000). The metamodel elements can be a basis for creating a dictionary that describes important concepts of this data model. The work of Date (2006) about OR_{TTM} is an example of such dictionaries.
5. A metamodel is a basis for creating a database catalog (see section 1.4.5) and metadata management systems that manage metadata about the various data sources.
6. It is possible to compare data models:
   - by finding mapping and discrepancies between their metamodel elements (see sections 1.3.2-1.3.5).
   - by calculating metrics values based on their metamodels (see section 1.3.7) and comparing these values. It is possible to use existing special tools like UML Model Measurement Tool (Lavazza and Agostini, 2005) in order to calculate metrics values.
7. Metamodels could help to improve a data model and its specification:
   - Inspection of visual structures in a metamodel helps to find violations of the orthogonality principle by the language that is specified by using this metamodel (see section 1.3.8).
   - Creation of a metamodel requires thorough study of existing specifications and therefore can help to find incompletenesses, inconsistencies and other mistakes in them (see section 1.4.4).
8. A metamodel helps to work out a profile in UML (OMG formal/05-07-04). For example, a metaclass can have a corresponding stereotype in a profile. The profile mechanism allows us to extend UML in order to use it for

different purposes. A profile can be used in order to create a (design-level) logical data model (meaning 2).

9. A metamodel of a data model is a basis for creating the metamodel of a specific database programming language.
10. The metamodels help to build up a federated DBMS. "A federated database system (FDBS) is a collection of cooperating but autonomous component databases systems. /..../ The software that provides controlled and coordinated manipulation of the component DBMs is called a federated database management system (FDBMS)." (Sheth and Larson, 1990) DBMSs that participate in a federation could have different underlying data models. The mapping of the metamodel elements helps to perform schema translation and schema integration tasks. An example of a FDBMS is ORDBMS$_{TTM}$ *Alphora Dataphor*. Another example is the federated multimedia database system EGVT (Bećarević and Roantree, 2004), the data of which is recorded in object-oriented and object-relational databases. For example, Bećarević and Roantree (2004) present and use mapping between the EGVT metamodel and the Oracle9i metamodel metaclasses.
11. The metamodels help to work out the language for interchanging the management information between management systems and applications. CIM (Common Information Model) is a step towards this direction. CIM v. 2.13 specifies some SQL Schema elements, but this specification is not complete (see section 1.4.1). The metamodels of different data models are a potentially important sources that help to extend CIM.
12. Model comparison (Kolovos et al., 2006) and model transformation (Kalnins et al., 2005), (Pedro et al., 2006) are operations that require existence of the metamodels of the models (that we want to compare or transform). In both cases, a system has to know the mapping between the elements of different metamodels. Based on this mapping it is possible to create comparison and transformation rules. An example of model comparison in case of data models is a comparison of an ORDBMS$_{SQL}$ and an ORDBMS$_{TTM}$ database. An example of model transformation is generation of a logical data model (meaning 2) based on an ORDBMS$_{TTM}$ database. Kalnins et al. (2005), Kolovos et al. (2006) and Pedro et al. (2006) present a very simplified SQL metamodel as part of their examples.

The Third Manifesto is structured as a set of prescriptions, proscriptions and suggestions. The manifest clearly distinguishes which of them are about the data model and which are orthogonal to it. The OR$_{TTM}$ metamodel is created based on The Third Manifesto relational model (RM) prescriptions, RM proscriptions (except 17 - transactions), RM very strong suggestions 1 (system keys), 2 (foreign key), 4 (transition constraints), 5 (quota queries), 6 (generalized transitive closure operator), 7 (generic operators). We also take into account the RM very strong suggestion 8 (special values) (Date and Darwen, 2000) that is removed from the third version of the manifest (Date and Darwen, 2006).

SQL:2003 is the official version of the SQL standard at the time of writing this dissertation. It contains a description of a database programming language, including its concrete syntax. This language has an underlying data model, which is not explicitly specified. SQL:2003 is a big international standard. We create the $\mathrm{OR}_{\mathrm{SQL}}$ metamodel based on the sections of the following parts of it: *Part 2: SQL/Foundation* (Melton, 2003) and *Part 11: SQL/Schemata* (Melton, 2003c). From *SQL/Foundation* we use the sections "Concepts" and "Schema definition and manipulation". From *SQL/Schemata* we use the section "Definition Schema". In the $\mathrm{OR}_{\mathrm{SQL}}$ metamodel, we present the metaclasses

- that have a counterpart in The Third Manifesto,
- that do not have a counterpart in The Third Manifesto. However, Date and Darwen (2000) or Date (2003) have discussed them and have reached the conclusion that for some reason they are unnecessary in $\mathrm{OR}_{\mathrm{TTM}}$.

The works (Date and Darwen, 2000), (Date, 2003), (Melton, 2003), (Date and Darwen, 2006) are the main sources of information for the comparison and discussion that are presented in sections 1.3.2-1.3.5. The discussion parts of the comparison should give a general overview of the main differences of the data models. Interested reader could find more thorough discussion from (Date and Darwen, 2000) and (Date, 2003).

1.3.1 Proposed Method for Comparing Data Models

We do the comparison in terms of the components of data models – data structures, integrity rules and data operators.

There are different viewpoints whether the specification of *data types* is a component of a data model. According to one school of thought, one of the main differences between the relational model and the object-relational model is that the former supports only simple predefined data types (INTEGER, CHAR, DATE, etc.) but the latter also supports complex types and allows users to create new types. On the other hand, Date and Darwen (2000, p. 21) write: "The question as to what data types are supported is orthogonal to the question of support for the relational model." Even Codd (1970) acknowledges the possibility of the non-simple domains (types), the permitted values of which are relations. One reason why he argues for eliminating non-simple domains is that they require more complicated data structures at the storage level than simple domains. Nevertheless, we decided to include data types to the comparison because The Third Manifesto and SQL:2003 have considerable differences in their support to the data types as well as using data types in order to build up a database.

This dissertation presents comparison of the $\mathrm{OR}_{\mathrm{TTM}}$ and $\mathrm{OR}_{\mathrm{SQL}}$ data models. However, the method can be used in order to compare other data models as well. The comparison consists of the following parts:

1. *Metamodels of the data models in the form of UML class diagrams*

We have to investigate whether there already exist metamodels of the data models (see section 1.4). If there is no metamodel or it is not precise enough, then we have to create a metamodel or improve the existing one.

Melton (2003b) writes: "The structure of the Definition Schema is a representation of the data model of SQL." Why cannot we just draw the OR_{SQL} metamodel *directly* based on the Definition Schema specification? In this case, we will not see important relationships. For example, assertions, table constraints and unique constraints are all constraints with some common and some different properties. The specification of the Definition Schema in SQL (Melton, 2003c) describes tables ASSERTIONS, TABLE_CONSTRAINTS and DOMAIN_CONSTRAINTS but does not describe table CONSTRAINTS. However, it describes table CHECK_CONSTRAINTS. Not all the constraints are check constraints. However, according to the CHECK constraint that is associated with table CHECK_CONSTRAINTS, it can actually contain data (names) of all the constraints. On the other hand, this general table does not contain information that must be present in case of all the constraints - whether a constraint is deferrable and whether it is initially deferred. Definition Schema in SQL presents *logical design* data model (meaning 2). On the other hand, the metamodel should present the *conceptual* model with all the important relationships (including generalization and whole-part) in order to help to understand the meaning of constructs in the data model and interconnections of these constructs.

If we create a metamodel of a data model based on the database language description, then we first have to decide which parts of the language are relevant in terms of data model and which are orthogonal to it (and therefore have no corresponding constructs in the metamodel of the data model).

We propose to use packages in order to control complexity and create groupings of logically interrelated classes. These packages are – *data types*, *data structures*, *data integrity* and *data operators*.

In some cases, it is necessary to add the stereotype <<singleton>> to a class as Ricters and Gogolla (1999) do. This stereotype indicates that there is exactly one instance of this class. An example is metaclass *Boolean* that belongs to the package *Data types*. Some attributes of the classes could have type *Enum*, which means that its possible value represents one of an enumerated set of values. In case of the OR_{SQL} metamodel, the attributes of metaclasses that have type Boolean can have the values *true* or *false,* but not *unknown*.

2. *Mapping between the metaclasses of the metamodels of the data models*

For each metaclass in one metamodel, we have to try to find one or more corresponding metaclasses from another metamodel. We have a pair of metaclasses in the mapping if the constructs behind these metaclasses have exactly the same semantics or they are semantically quite similar. Whether or not the constructs are semantically so similar that the mapping can be created depends on the opinions of the persons who perform the comparison. This comparison is a kind of framework that allows us to reason about semantic similarity of different constructs.

3. Discrepancies between the data models

We consider the constructs that are represented as metaclasses in the metamodels.

Let us assume that we compare two data models A and B. If we decide that data model A has much clearer and much more precise specification than the other data model B, then we can think about A as a kind of ontology. Then we can perform an ontological evaluation of data model B in order to find its construct redundancy, construct overload, construct excess and construct deficit problems.

Generally, we do not prefer one data model and want to compare them without prejudice. Based on the mapping between metaclasses of two data models A and B we can find:

a) Cases when a metaclass of A/B has more than one corresponding metaclass of B/A.

b) Cases when a metaclass of A/B does not correspond to any metaclass of B/A.

We could use the same names as Opdahl and Henderson-Sellers (2002) in order to refer to different cases of discrepancies. If a metaclass of the metamodel of A has more than one corresponding metaclass of the metamodel of B, then its reason could be:

- Data model B (and therefore its metamodel as well) is too complex. Data model A pays more attention to the orthogonality principle of language design. One requirement of this principle is that a language should provide a comparatively small set of primitive constructs (Date and Darwen, 2000). In this case the metaclasses of B that correspond to the metaclass of A have a common supertype or it is at least possible to create that supertype. We say that there is a *construct redundancy* in B.
- The construct of A is the counterpart of two or more constructs of B, the semantic of which is very different (in the metamodel of B their corresponding metaclasses do not have a common superclass and it is not possible to create that). We say that there is a *construct overload* in A.

If a metaclass of the metamodel of A has no corresponding metaclass of the metamodel of B, then B has *construct deficit* and data model A has *construct excess*. Its reasons could be:

- Data model B is less powerful than data model A because it does not provide an important construct that should be present in a well-designed data model.
- Metamodel of data model B does not have a clearly corresponding metaclass, but it could be created in the metamodel without violating principles of the data model (for example, by creating a common superclass of some existing metaclasses).
- Creators of data model B think that a construct is orthogonal to the data model and therefore it is missing from the specification of B.
- The construct is not in B because creators of B think that a similar effect can be achieved by using other constructs that are already present in B.

In the latter two cases, the authors of B might explicitly argue against a construct.

4. Mapping between the metaclasses does not mean that the constructs behind them have exactly the same semantics. Therefore, we need an additional section that contains the textual description of the differences.
5. *Metrics values.*

For each data model, we propose to calculate *at least* the amount of metaclasses and the amount of their attributes. It is sometimes difficult to decide whether to model something by using a class or using an attribute in a UML class diagram. "If in doubt, define something as a separate conceptual class rather than as an attribute." (Larman, 2002, p. 170) Therefore, we also present the sums of these two values. The resulting values characterize the relative complexity of the data models. We propose to calculate these values in case of each package of a metamodel - data structures, data integrity, data operators and data types as well as in general for the entire data model.

Rossi and Brinkkemper (1996) also propose to count relationship types. It is difficult to calculate this metrics value for each package because many relationship types cross boundaries of packages and connect metaclasses that are part of different packages.

If we know the amount of metalasses, attributes and relationship types, then it is also possible to calculate values of aggregate metrics that are proposed by Rossi and Brinkkemper (1996).

1.3.2 Comparison of Data Types

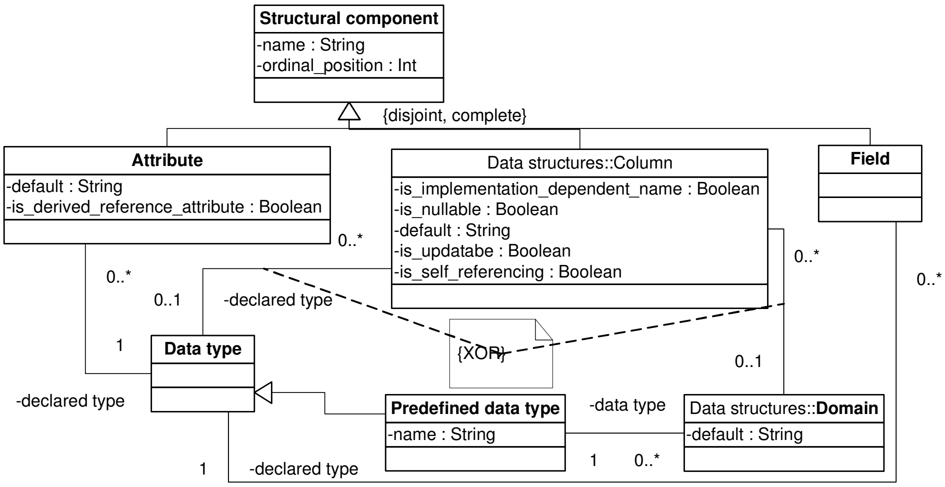


Figure 4 Structural components in OR$_{SQL}$

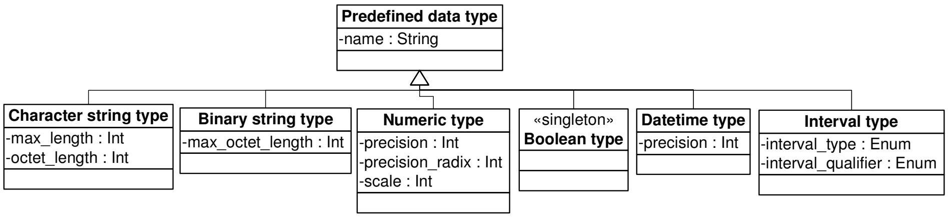


Figure 5 Predefined data types in OR$_{SQL}$

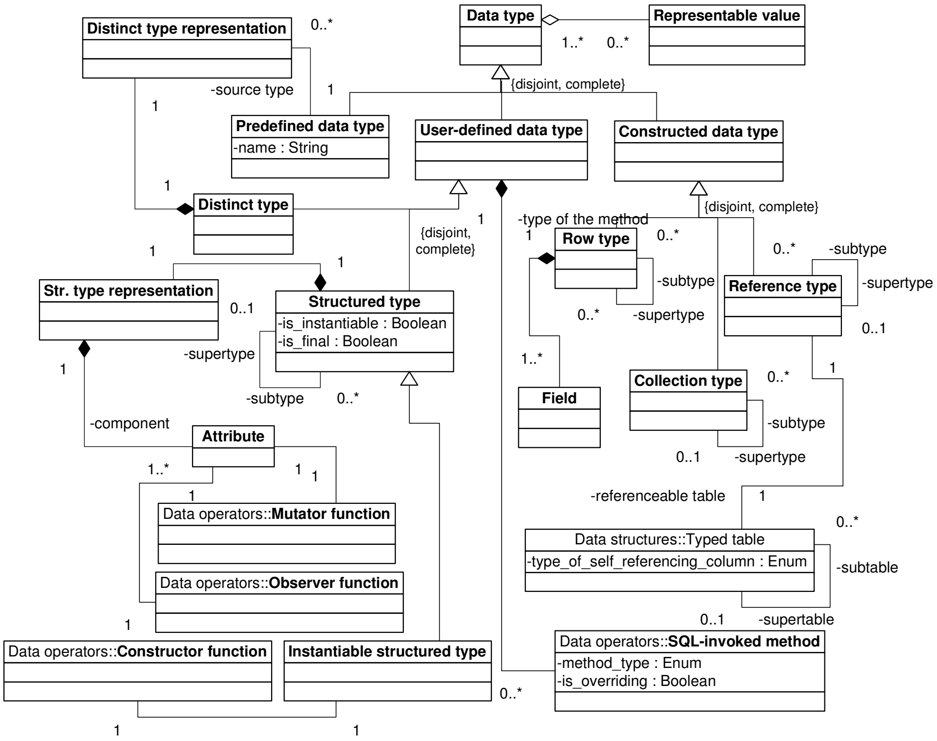


Figure 6 Data types in OR$_{SQL}$

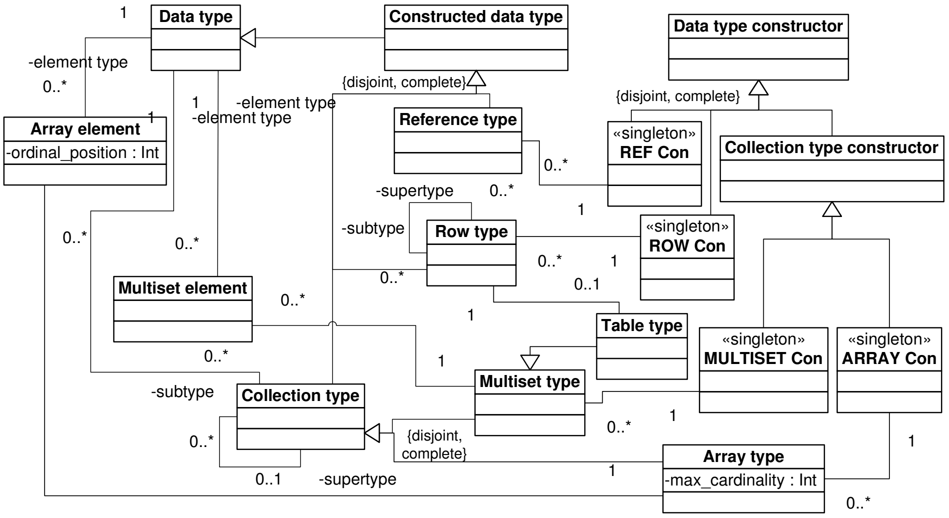


Figure 7 Data type constructors in OR$_{SQL}$

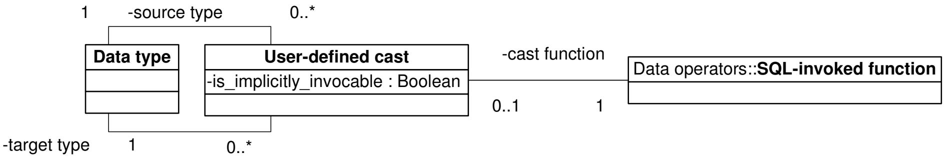
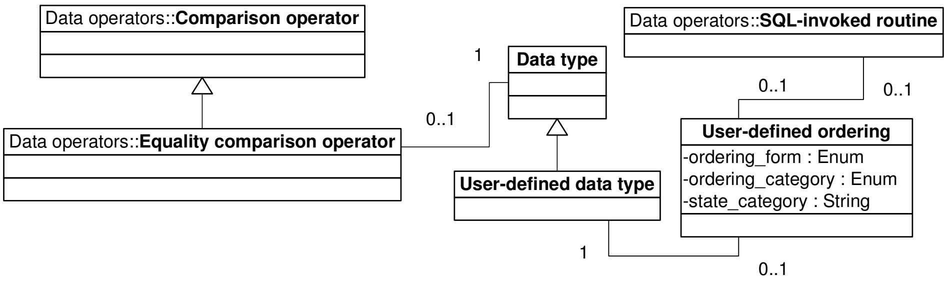
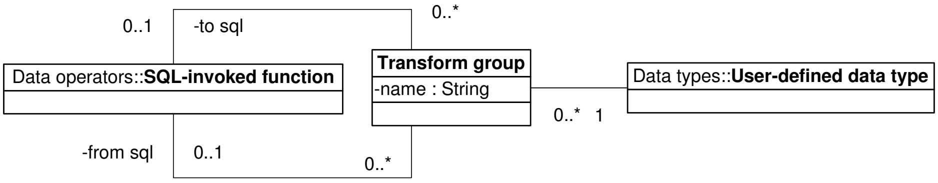


Figure 8 Casts in OR$_{SQL}$



Data operators::Comparison operator
Data operators::Equality comparison operator
1
0..1
Data type
User-defined data type
Data operators::SQL-invoked routine
0..1
0..1
User-defined ordering
-ordering_form : Enum
-ordering_category : Enum
-state_category : String
1
0..1

Figure 9 Equality comparison operators in OR$_{SQL}$



0..1
-to sql
0..*
Data operators::SQL-invoked function
Transform group
-name : String
Data types::User-defined data type
0..*
1
-from sql
0..1
0..*

Figure 10 Transforms in OR$_{SQL}$

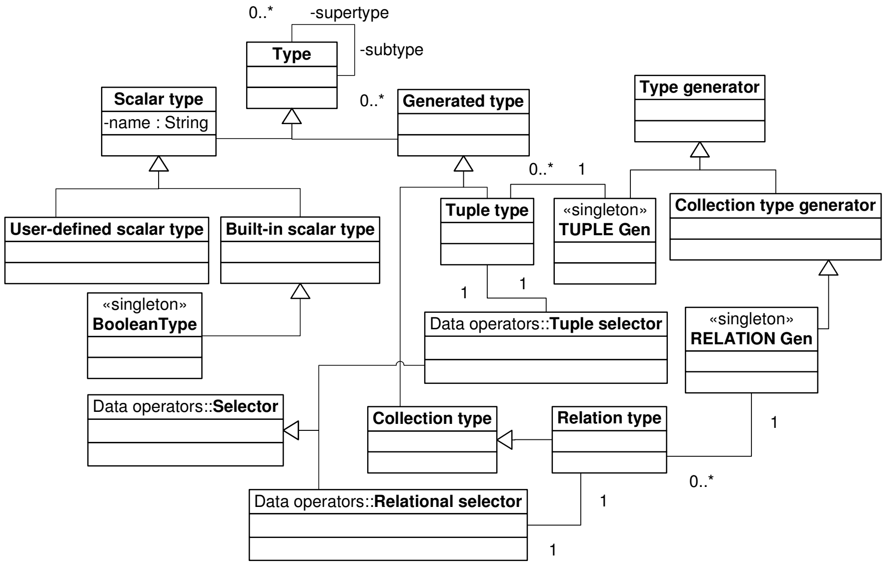


Figure 11 Data types in OR$_{TTM}$

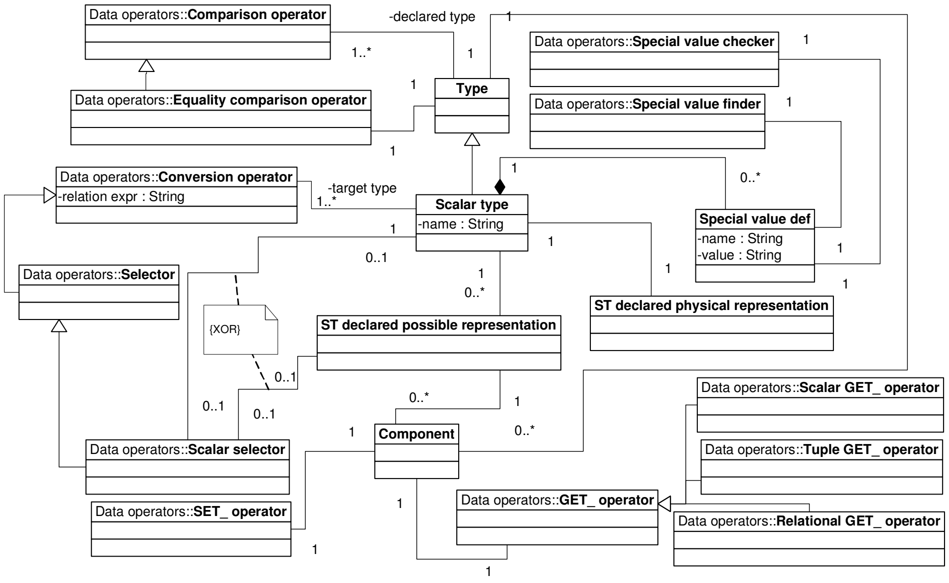

Data operators::Comparison operator
-declared type
1
1..*
1
Data operators::Special value checker
1
Data operators::Equality comparison operator
1
Type
Data operators::Special value finder
1
1
1
0..*
Data operators::Conversion operator
-relation expr : String
-target type
1..*
Scalar type
-name : String
1
Special value def
-name : String
-value : String
1
0..1
1
1
Data operators::Selector
1
0..*
1
ST declared physical representation
{XOR}
ST declared possible representation
1
0..1
0..1
0..1
0..*
1
Data operators::Scalar GET_ operator
1
Component
0..*
Data operators::Tuple GET_ operator
Data operators::Scalar selector
Data operators::GET_ operator
Data operators::SET_ operator
1
Data operators::Relational GET_ operator
1
1

Figure 12 Scalar types in OR$_{TTM}$

Table 1 Mapping of OR$_{SQL}$ and OR$_{TTM}$ metaclasses that belong to the packages "Data types"

ID	*OR$_{SQL}$ metaclass*	*OR$_{TTM}$ metaclass that represents the most similar concept*
1	Attribute	Component
2	Boolean type	BooleanType (truth-value type)

ID	*OR_{SQL} metaclass*	*OR_{TTM} metaclass that represents the most similar concept*
3	Collection type	Collection type
4	Collection type constructor	Collection type generator
5	Constructed data type	Generated type
6	Data type (Data type descriptor)	Type (data type, domain)
7	Data type constructor	Type generator
8	Distinct type	User-defined scalar type
9	Distinct type representation	ST (Scalar type) declared possible representation
10	Field	Attribute
11	Instantiable structured type	User-defined scalar type
12	Predefined data type (built-in data type)	Built-in scalar type (system-defined type)
13	Representable value	Value
14	ROW Con	TUPLE Gen
15	Row type	Tuple type
16	Row	Tuple (tuple value)
17	Row component	Tuple component
18	Str. type representation	ST (Scalar type) declared possible representation
19	Structured type	User-defined scalar type
20	Table type	Relation type
21	User-defined cast	Conversion operator, Selector, Scalar selector
22	User-defined data type (UDT, abstract data type, ADT)	User-defined scalar type

Construct redundancy in OR_{SQL}:

- *Instantiable structured type, User-defined data type*, *Distinct type* and *Structured type* in OR_{SQL} vs. *User-defined scalar type* in OR_{TTM};
- *Distinct type representation* and *Str. type representation* in OR_{SQL} vs. *ST declared possible representation* in OR_{TTM}.

Construct redundancy in OR_{TTM}:

- *Conversion operator*, *Selector*, *Scalar selector* in OR_{TTM} vs. *User-defined cast* in OR_{SQL}.

Construct deficit in OR_{TTM}: *ARRAY Con*$^{(1)}$, *Array element*$^{(1)}$, *Array type*$^{(1)}$, *Character string type*$^{(2)}$, *Binary string type*$^{(2)}$, *Datetime type*$^{(2)}$, *Interval type*$^{(2)}$, *MULTISET Con*$^{(1)}$, *Multiset element*$^{(1)}$, *Multiset type*$^{(1)}$, *Numeric type*$^{(2)}$, *User-defined ordering*, *REF Con*$^{(3)}$, *Reference type*$^{(3)}$, *Transform group*$^{(4)}$, *Typed table*$^{(3)}$.

$^{(1)}$ – Date and Darwen (2000) suggested ARRAY and SET type generators as *orthogonal* features, but more lately they have come to a conclusion that these types of generators and the corresponding types are unnecessary (Date and Darwen, 2006). Please also note that a set cannot contain repeating elements but a multiset can.

[(2)] – OR_{TTM} does not prohibit built-in types, but lets vendors of DBMSs to decide which predefined types to implement.

[(3)] – The Third Manifesto argues explicitly against pointers and typed tables in the section "OO Prescriptions" (Date and Darwen, 2006).

[(4)] – Authors of OR_{TTM} think that a call level interface is orthogonal to a data model.

Construct deficit in OR_{SQL}: *Appearance of a value*, *ST declared* p*hysical (actual) representation*, *RELATION Gen*, *Special value def.*

In addition, we note that the following OR_{TTM} metaclasses do not have one clearly corresponding metaclass in the current OR_{SQL} metamodel: *Body*, *Relation heading*, *Tuple heading* (see Figure 21)*, Scalar type*. However, it is possible to create these metaclasses as abstractions without violating the principles of OR_{SQL}.

1.3.2.1 Discussion

Among other things, a type is a finite set of values that the computer system is able to represent (Date, 2003). Distinct types have no values in common according to The Third Manifesto. Data types in OR_{TTM} have to be distinct. Data types in OR_{SQL} do not have to be distinct – a representable value can belong to more than one data type (Melton, 2003, p.11).

OR_{TTM} prescribes only one built-in scalar type - Boolean. $ORDBMS_{TTM}$ vendors and database programming language designers have the freedom to provide additional built-in types. For example, Tutorial D language specifies additional built-in types: INTEGER, RATIONAL and CHAR (Date and Darwen, 2006). Date et al. (2003) describe the use of timestamp and interval types (and corresponding operators) in order to build up a database that contains time-related data. If we use OR_{TTM} as a basis of an Engineering DBMS, then this system could, for example, provide built-in types (and operators) that correspond to certain artifact types (see section 3.1.1).

OR_{SQL} specifies many predefined data types (see Figure 5). It is interesting to note that Boolean type was firstly specified not in the first edition of SQL standard (SQL-86), but in a major revision SQL:1999 (Gulutzan and Pelzer, 1999). OR_{TTM} advocates two-valued logic and therefore type Boolean includes only values *TRUE* and *FALSE.* OR_{SQL}, on the other hand, uses three-value logic. In this case type Boolean should include values *TRUE*, *FALSE* and *UNKNOWN.* OR_{SQL} uses "NULL value" in order to represent the value *UNKNOWN.* NULL behaves sometimes differently than other values. The well-known example is that the result of the comparison of two NULL's (NULL=NULL) is not *TRUE.* This and other examples are basis on the viewpoint that "NULL is not a value in SQL because it does not have all the properties of the values" (Date and Darwen, 2000, p. 426). OR_{TTM} rejects the use of NULLs and stresses that all the attributes in the relations must always have a value (see also the discussion about special values at the end of this section).

Both $\mathrm{OR_{TTM}}$ and $\mathrm{OR_{SQL}}$ permit database users to create data types and operators ($\mathrm{OR_{TTM}}$) or routines ($\mathrm{OR_{SQL}}$) that make operations with the values with these types. In case of user-defined (scalar) types, $\mathrm{OR_{SQL}}$ distinguishes distinct types and structured types (see Figure 6). There is no such distinction in $\mathrm{OR_{TTM}}$ (see Figure 11).

A possible representation of a type specifies how the users see it. A physical representation of a type specifies how the values are recorded at the storage level. SQL:2003 states: "The definition of a user defined type specifies a representation for values of that type. /.../ physical representations of user-defined type values are implementation-dependent." (Melton, 2003, p. 37) Based on these citations we conclude that the creator of a user-defined type specifies a possible representation of this type in $\mathrm{OR_{SQL}}$.

Each type in $\mathrm{OR_{TTM}}$ must have associated selector operator that is a kind of conversion operator. All of its arguments are literals and its successful invocation returns value with this type. Its invocation must always be explicit and the only way to destroy this operator is to destroy the type.

One difference between $\mathrm{OR_{SQL}}$ and $\mathrm{OR_{TTM}}$ is that $\mathrm{OR_{SQL}}$ does not allow us to declare more than one possible representation in a user-defined type declaration, but $\mathrm{OR_{TTM}}$ allows that. Each declared possible representation PR of a scalar type T in $\mathrm{OR_{TTM}}$ must have associated automatically created selector operator (see Figure 12). Its invocation returns value of type T with the possible representation PR. In addition, $\mathrm{OR_{TTM}}$ permits the creation of additional conversion operators. $\mathrm{OR_{SQL}}$ allows the creation of user-defined casts for type conversion (see Figure 8). It is possible to specify a cast so that its invocation would be implicit. It is possible to destroy a cast without destroying a type.

In $\mathrm{OR_{TTM}}$, each type *must* have an associated equality comparison operator for comparing equality of values with this type. In $\mathrm{OR_{SQL}}$, a user-defined type *can* have an associated equality comparison operator. This operator can be created only after a user-defined ordering has been created (see Figure 9). A DBMS has to create ordering automatically in case of creation of a distinct type. It has to be created explicitly after the creation of a structured type.

In $\mathrm{OR_{SQL}}$, a structured type can have associated transform groups, which group the functions that help exchange structured type values with host language programs and with external routines (see Figure 10). Other *Orthogonal* Prescription 3 of The Third Manifesto states that a database programming language may support, but does not require (a) invocations from so-called "host programs" written in other languages and (b) the use of other languages for implementation of user-defined operators. Therefore, issues that are addressed by transform groups in $\mathrm{OR_{SQL}}$ are orthogonal to $\mathrm{OR_{TTM}}$.

$\mathrm{OR_{TTM}}$ requires that an $\mathrm{ORDBMS_{TTM}}$ must support two type generators that allow creation of non-scalar types – TUPLE and RELATION (see Figure 11). $\mathrm{OR_{SQL}}$ specifies four type constructors – REF, ROW, ARRAY and MULTISET (see Figure 7).

Fields of a constructed row type are left-to-right ordered in $\mathrm{OR_{SQL}}$ (see attribute *ordinal_position* in the metaclass *Structural component* in Figure 4). If we change the order of fields in the declaration of a row type, then this declaration specifies a new type. On the other hand, attributes of a tuple type are not left-to-right ordered in $\mathrm{OR}_{\mathrm{TTM}}$.

The value of a constructed multiset type in $\mathrm{OR}_{\mathrm{SQL}}$ is an unordered collection of elements. All these elements must have the same type (see Figure 7). It can be any type that is supported by a particular $\mathrm{ORDBMS}_{\mathrm{SQL}}$ (except the multiset type itself). This collection can contain repeating elements. The value of a relation type in $\mathrm{OR}_{\mathrm{TTM}}$ is an unordered set of tuples all of which have the same tuple type. The set cannot contain repeating elements (tuples). $\mathrm{OR}_{\mathrm{SQL}}$ uses the concept "table type" in the context of table functions. The wording "<returns type>::= <returns data type> [<result cast>] | <returns table type>" (Melton, 2003, p. 676) gives an impression that a table type is not a data type. Actually the result of an invocation of a table function has a composite type MULTISET(ROW(...)) that is created by using two different type constructors.

Reference type, a kind of constructed data type, is used together with the typed tables in $\mathrm{OR}_{\mathrm{SQL}}$ (see Figure 6). In contrast to $\mathrm{OR}_{\mathrm{TTM}}$, $\mathrm{OR}_{\mathrm{SQL}}$ allows us to create typed tables based on the user-defined structured types. The row type of a typed table is derived from a structured type. A typed table is a referencable table. "A REF value is a value that references a row in a referenceable table." (Melton, 2003, p. 43). A reference type is a set of REF values that reference rows in a typed table that is defined based on a structured type. These values are like Object ID-s in object systems that "are addresses – at least conceptually – and are hidden from the user" (Date, 2003, p. 826). Such state of the affairs is caused by the view of SQL creators that the object-oriented concepts "class" (or type) and "instance" are the counterparts of the database concepts "table" and "row", respectively. $\mathrm{OR}_{\mathrm{TTM}}$, on the other hand, advocates that the counterpart of the concept "class" is the concept "data type".

$\mathrm{OR}_{\mathrm{TTM}}$ treats the concepts "domain" and "data type" as synonyms. $\mathrm{OR}_{\mathrm{SQL}}$, on the other hand, distinguishes these concepts. "A data type is a set of representable values." (Melton, 2003, p. 11) "A domain is a set of permissible values." (Melton, 2003, p. 49) A domain in $\mathrm{OR}_{\mathrm{SQL}}$ is a reusable specification of the properties of the base table columns (see Figure 14). A domain must be associated with a *predefined* data type. A domain may have one or more associated domain constraints and it may specify a default value.

A value can be missing from a database for different reasons. Note that the fact that some information is missing for some reason is also information that should be recorded in a database. $\mathrm{OR}_{\mathrm{TTM}}$ and $\mathrm{OR}_{\mathrm{SQL}}$ deal differently with the missing values in a database. $\mathrm{OR}_{\mathrm{SQL}}$ uses NULL's in order to represent the missing information. On the other hand, The Third Manifesto strongly suggests (Date and Darwen, 2000, p. 218) that each scalar type can have associated special values (see Figure 12). A declaration of these special values is part of the declaration of the type. Each special value represents some reason why the information is missing. The special values belong to the set of permitted values of a type. Each special value must have two associated scalar operators. One of the operators (we call it *Special value finder*) is used in order to return the special value. Another operator (we call it *Special value checker*), the declared type of which is Boolean checks whether the result of the evaluation of a scalar expression is the special value or not.

1.3.3 Comparison of Data Structures

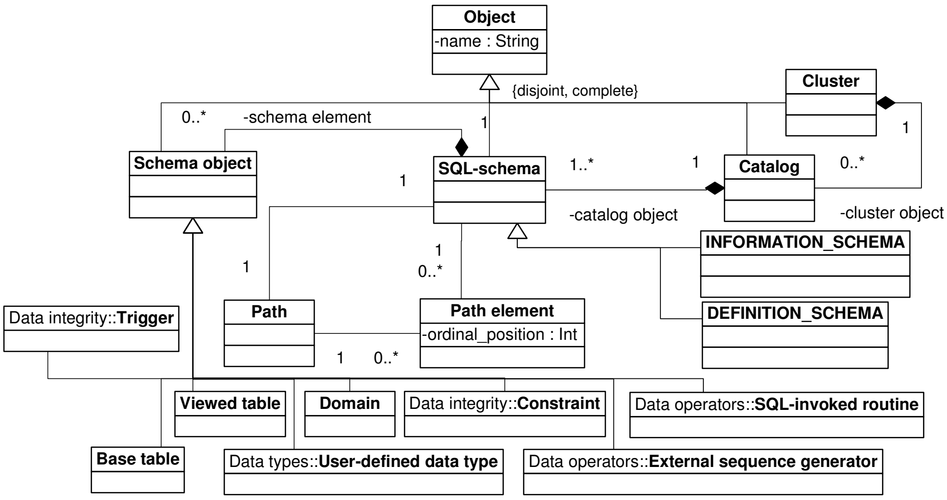
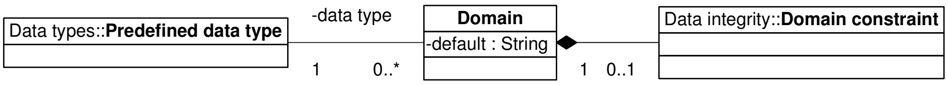


Figure 13 Objects in OR$_{SQL}$



Data types::Predefined data type
-data type
1
0..*
Domain
-default : String
1
0..1
Data integrity::Domain constraint

Figure 14 Domains in OR$_{SQL}$

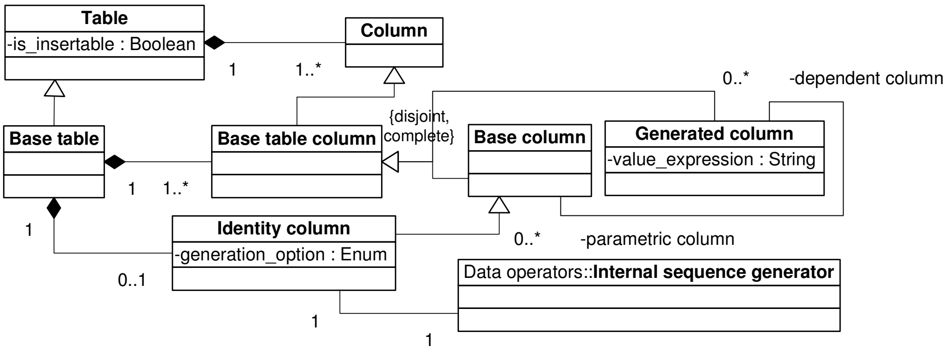


Figure 15 Columns in OR$_{SQL}$

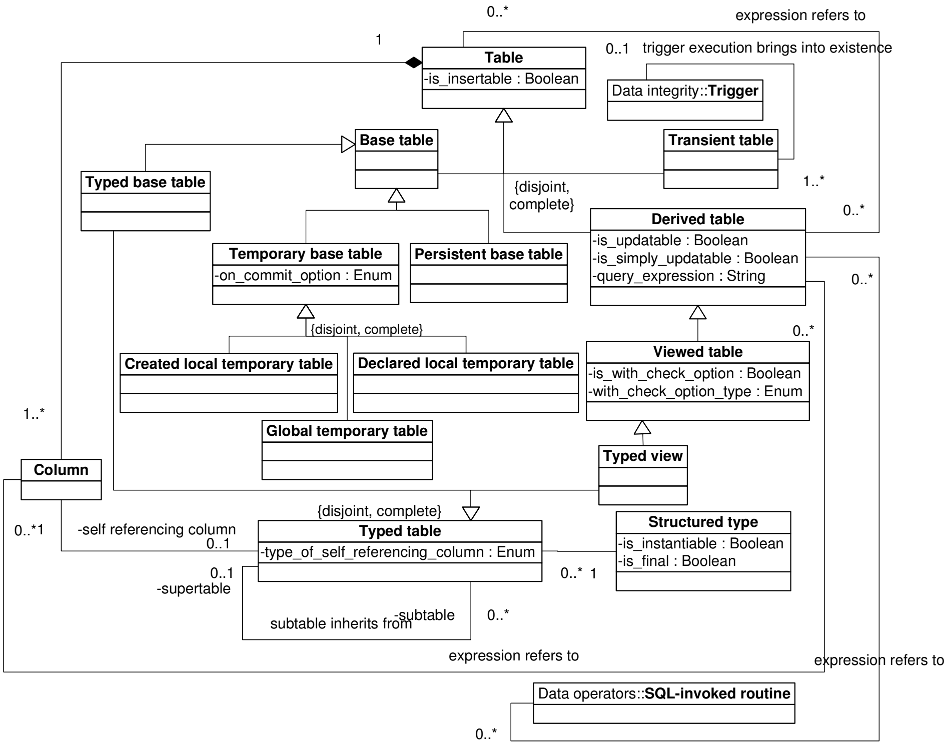


Figure 16 Tables in OR$_{SQL}$

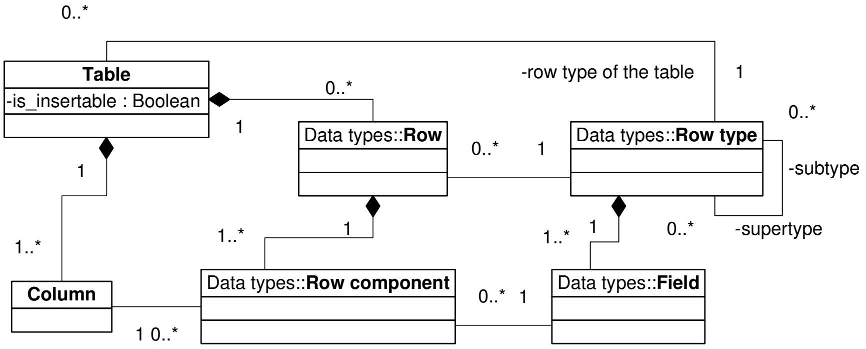


Figure 17 Rows in OR$_{SQL}$

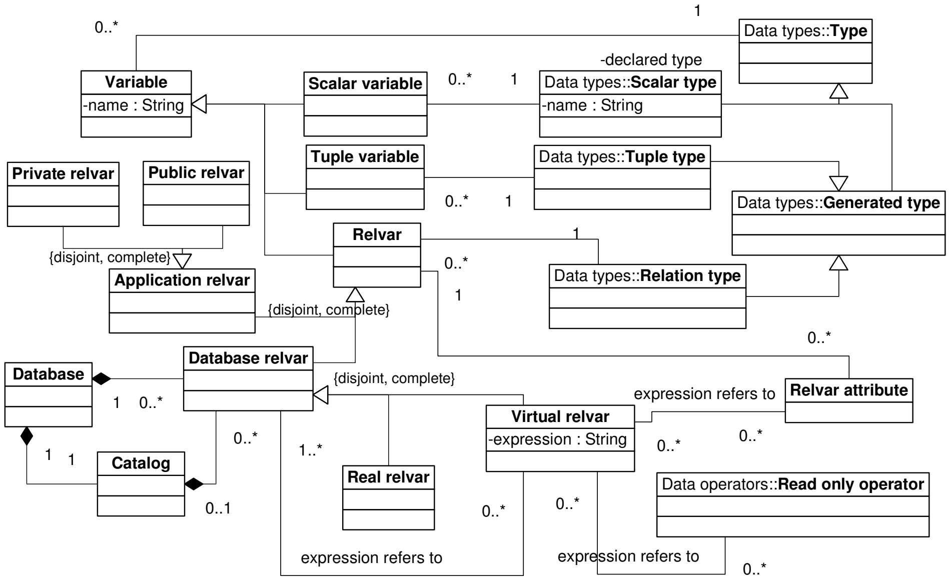


Figure 18 Variables in OR$_{\text{TTM}}$

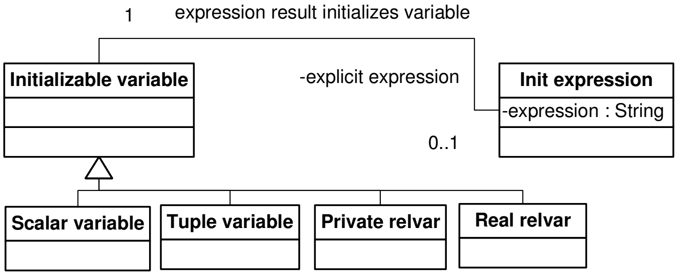


Figure 19 Initializable variable in OR$_{\text{TTM}}$

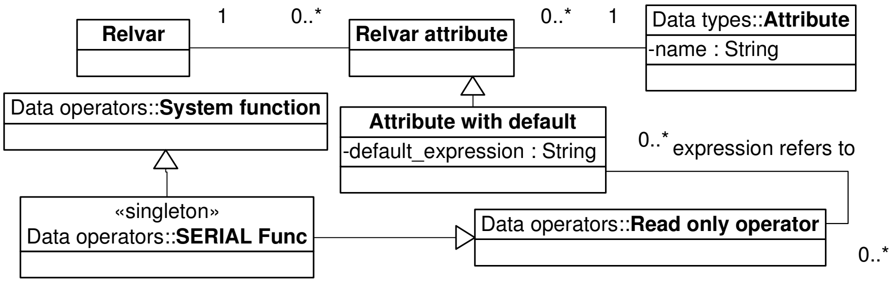

Relvar
1
0..*
Relvar attribute
0..*
1
Data types::Attribute
-name : String
Data operators::System function
Attribute with default
-default_expression : String
0..* expression refers to
«singleton»
Data operators::SERIAL Func
Data operators::Read only operator
0..*

Figure 20 Relvar attributes in OR$_{\text{TTM}}$

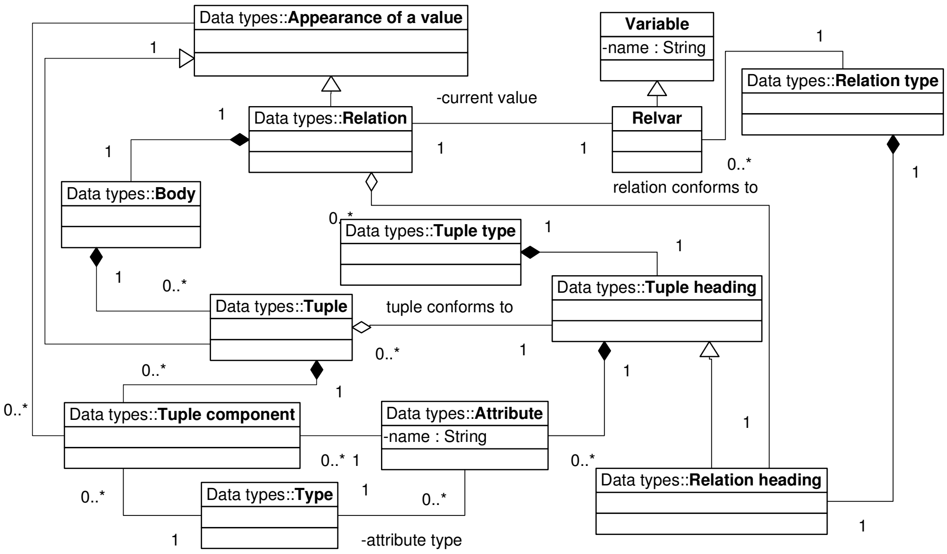


Figure 21 Relations in OR_{TTM}

Table 2 Mapping of OR_{SQL} and OR_{TTM} metaclasses that belong to the packages "Data structures"

ID	OR_{SQL} metaclass	OR_{TTM} metaclass that represents the most similar concept
1	Base column	Relvar attribute
2	Base table	Real relvar (Base relvar), Relation
3	Base table column	Relvar attribute
4	Catalog	Database
5	Cluster (mentioned as SQL Object by Gulutzan and Pelzer (1999, p. 27))	Database
6	Column	Relvar attribute (Attribute of relvar)
7	Created local temporary table	Private relvar, Relation
8	Declared local temporary table	Private relvar, Relation
9	DEFINITION_SCHEMA	Catalog
10	Derived table	Relation
11	Generated column	Attribute with default
12	Global temporary table	Private relvar, Relation
13	Identity column	Attribute with default
14	INFORMATION_SCHEMA	Catalog
15	Persistent base table	Real relvar, Relation
16	Table	Relvar (Relation variable) Relation (Value of relation variable)
17	Temporary base table	Private relvar, Relation
18	Viewed table (View)	Virtual relvar, Relation

Construct redundancy in OR_{SQL}:

- *Catalog* and *Cluster* in OR_{SQL} vs. *Database* in OR_{TTM};

- *DEFINITION_SCHEMA* and *INFORMATION_SCHEMA* in $\mathrm{OR}_{\mathrm{SQL}}$ vs. *Catalog* in $\mathrm{OR}_{\mathrm{TTM}}$;
- *Generated column* and *Identity column* in $\mathrm{OR}_{\mathrm{SQL}}$ vs. *Attribute with default* in $\mathrm{OR}_{\mathrm{TTM}}$;
- *Created local temporary table*, *Declared local temporary table*, *Global temporary table*, *Temporary base table* in $\mathrm{OR}_{\mathrm{SQL}}$ vs. *Private relvar* in $\mathrm{OR}_{\mathrm{TTM}}$.

Construct overload in $\mathrm{OR}_{\mathrm{SQL}}$: *Created local temporary table*, *Declared local temporary table*, *Global temporary table*, *Temporary base table, Base table*, *Persistent base table*, *Table*, *Viewed table* in $\mathrm{OR}_{\mathrm{SQL}}$ correspond to *Relvar* (or its subtypes) and *Relation* in $\mathrm{OR}_{\mathrm{TTM}}$. Relvar is a variable and relation is its value.

Construct deficit in $\mathrm{OR}_{\mathrm{TTM}}$: *Domain*, *Path element*, *Path*, *SQL-schema*, *Transient table*, *Typed table*$^{(1)}$, *Typed base table*$^{(1)}$, *Typed view*$^{(1)}$.

$^{(1)}$ - The Third Manifesto argues explicitly against pointers and typed tables in the section "OO Prescriptions" (Date and Darwen, 2006).

Construct deficit in $\mathrm{OR}_{\mathrm{SQL}}$: *Application relvar*, *Initializable variable* $^{(1)}$, *Init expression*, *Public relvar*, *Scalar variable*$^{(1)}$, *Tuple variable*$^{(1)}$, *Variable*$^{(1)}$.

$^{(1)}$ – It is possible to use variables in SQL-invoked routines.

1.3.3.1 Discussion

$\mathrm{OR}_{\mathrm{TTM}}$ is built up based on a set of core concepts: "value", "variable", "type", "operator" (see Figure 3) that makes the specification easier to understand as compared to SQL:2003. Gulutzan and Pelzer (1999, p. 255) write: "The SQL Standard describes the concepts on which SQL is based in terms of objects, such as Tables" (see Figure 13). In this context, the concept "object" is not a counterpart to the $\mathrm{OR}_{\mathrm{TTM}}$ concept "value", but rather to the concept "variable". An object is a cluster, a catalog, a SQL-schema or a schema object.

The basic data structure in $\mathrm{OR}_{\mathrm{TTM}}$ is a database relation variable (see Figure 18). Although $\mathrm{OR}_{\mathrm{TTM}}$ also specifies application relvars, we use the concept *relvar* from now on in order to refer to a database relation variable, if not stated otherwise. The basic data structure in SQL is a table (see Figure 16). A database in $\mathrm{OR}_{\mathrm{TTM}}$ is a named set of database relvars. A database itself can be seen as a variable (Dbvar) (Date and Darwen, 2000, p. 155). Scalar and tuple variables are not permitted in the databases. $\mathrm{OR}_{\mathrm{SQL}}$ does not define the concept "database" and specifies instead the composite objects "cluster", "catalogue", and "SQL-schema" (see Figure 13).

$\mathrm{OR}_{\mathrm{SQL}}$ permits creation of temporary base tables in order to allow users (applications) to record the results of their data operations temporarily to a database so that other sessions cannot access this data (see Figure 16). $\mathrm{OR}_{\mathrm{TTM}}$ allows us to use private application relvars for this purpose (see Figure 18). "The definition of a global temporary table or a created local temporary table appears in a schema." (Melton, 2003, p. 52) The content of this kind of a table is temporary, but the table as structural element persists after the end of a session. A declared local temporary table is materialized in some schema if it is for the first time referenced in a session, and it is dropped at the end of this session. In OR_{TTM}, private application relvars are not part of a database.

The concept "catalog" has different meanings in OR_{TTM} and OR_{SQL}. In OR_{TTM}, it means a data dictionary that contains some database relvars. Its purpose is to present the description of the data that is recorded in that database. In OR_{SQL}, a catalog is a named group of schemas, which are in turn named groups of schema objects. Each catalog in OR_{SQL} must contain a schema with the name "INFORMATION_SCHEMA" (see Figure 13). This schema consists of views and domains, the purpose of which is to describe the SQL-data, which belongs to that catalog. The SQL standard specifies these views. "The INFORMATION_SCHEMA Views are based on the Tables of an Ur-Schema called DEFINITION_SCHEMA, but the Standard does not require it to actually exist – its purpose is merely to provide a data model to support INFORMATION_SCHEMA." (Gulutzan and Pelzer, 1999, p. 287) Therefore, OR_{SQL} metaclasses *INFORMATION_SCHEMA* and *DEFINITION_ SCHEMA* are counterparts of OR_{TTM} metaclass *catalog*. Views in OR_{SQL} INFORMATION_SCHEMA are not updateable – it is not possible to insert/update/delete data in underlying base tables through these views. OR_{TTM}, on the other hand, prescribes that the authorized user must have a possibility to assign new values to the relvars that are part of a catalog. OR_{TTM} does not specify the exact structure of a catalog as OR_{SQL} does.

OR_{TTM} clearly distinguishes the concepts "value" and "variable". A variable has at any moment one value, but it is possible to change this value. Defining a scalar variable, a tuple variable, a real relvar or a private relvar has the effect of initializing this variable to some value. This value can be specified explicitly by specifying an *init expression* that returns a value that has the same type as the variable (see Figure 19). A value of a relvar is called a *relation* in OR_{TTM} (see Figure 21). In OR_{SQL}, the concept "table" means "table value" as well as "table variable". The next definition is a good example of that: "A table is a collection of rows having one or more columns." (Melton, 2003, p. 51) All the rows in a table are values that have the same row type (see Figure 17). All the values of the *n-th* field in a row are the values of the *n-th* column in the table. Differences of OR_{TTM} relation and OR_{SQL} table (value) according to Date and Darwen (2000, p. 430) and Date (2003, p. 151):

- Left-to-right order of columns has significance in OR_{SQL}. For example, writer of SQL INSERT statements has to know the order of columns. The order of attributes is not important in OR_{TTM} relations and all the attributes are identified by their names and not by their ordinal position.
- A base table or a viewed table cannot contain two or more columns with the same name in OR_{SQL}. However, a derived table that is derived directly or indirectly from one or more other tables by the evaluation of a query expression can contain more than one column with the same name. In OR_{TTM}, a heading of a relation cannot contain more than one attribute with the same name.

- An OR_{SQL} table is a collection (of rows) that can contain duplicates. Database users can prevent them, but are not obliged to do so. Declaring a primary key or a unique constraint prevents duplicate rows in a base table. Special syntax in a query expression prevents duplicated rows in a derived table. OR_{TTM} explicitly prohibits duplicate tuples in the relations. Each relvar *must* have at least one candidate key. In addition, an $ORDBMS_{TTM}$ has to automatically remove duplicates from the result of the evaluation of a query expression.
- A table in OR_{SQL} must have at least one column. OR_{TTM}, on the other hand, specifies two special relations – *TABLE_DEE* and *TABLE_DUM* where the set of attributes is empty – in other words, these relations have no attributes (Date, 2003, p. 154).

One difference of OR_{SQL} tables and OR_{TTM} relvars is that columns in a table are ordered (see attribute *ordinal_position* in OR_{SQL} metaclass *Structural component* in Figure 4), but attributes in a relvar are not. In addition, OR_{SQL} permits creation of a base table based on a user-defined structured type (see the metaclasses *Typed base table* and *Typed view* in Figure 16). It is not possible to create a relvar based on a scalar type in OR_{TTM}. OR_{SQL} allows inheritance between the typed tables. OR_{TTM}, on the other hand, does not consider inheritance between relvars as a good idea and allows only inheritance between the types (as part of very strong orthogonal (to data model) suggestions). One difference of OR_{SQL} viewed tables and OR_{TTM} virtual relvars is that not all logically updateable views are actually updateable in OR_{SQL}.

It is possible to declare a default value to a relvar attribute in OR_{TTM} as well as to a base table column in OR_{SQL}. OR_{TTM} strongly suggests to use the system function SERIAL in order to automatically generate unique values to some attribute of a relvar (see Figure 20). This value can be obtained via the default mechanism. A system function may require access to certain "environmental variables" and involves certain "hidden arguments" (Date and Darwen, 2000, p. 204). The similar effect can be achieved in OR_{SQL} by using the identity column of a table (see Figure 15) that has an associated internal sequence generator (see Figure 30). "A sequence generator is a mechanism for generating successive exact numeric values, one at a time." (Melton, 2003, p. 77) If a row that is inserted to a table does not contain a column corresponding to an identity column, then the value for this column is generated by an internal sequence generator (Melton, 2003, p. 57).

A value of a generated column in OR_{SQL} base table is the result of evaluation of a generation expression, the declared type of which is "by implication that of the column" (Melton, 2003, p. 57) (see Figure 15). A limitation is that "A generation expression can reference base columns of the base table to which it belongs but cannot otherwise access SQLdata." (Melton, 2003, p. 57) A similar effect can be achieved in OR_{TTM} by specifying that the default value of an attribute is the result of evaluating some expression. This expression can refer to one or more read-only operators (see Figure 20).

1.3.4 Comparison of Data Integrity Rules

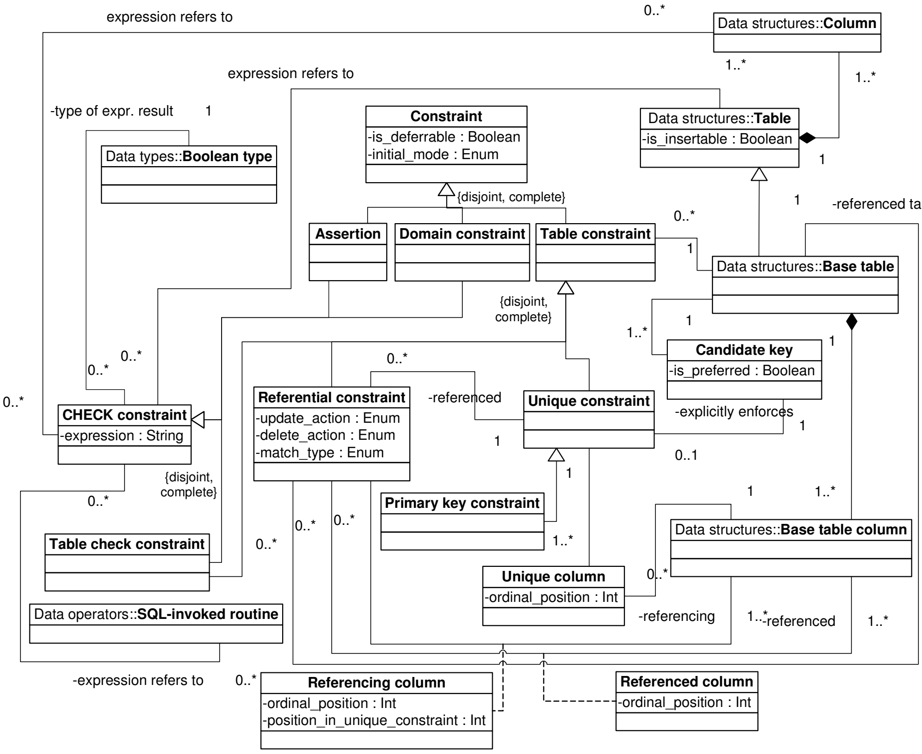


Figure 22 Constraints in OR_{SQL}

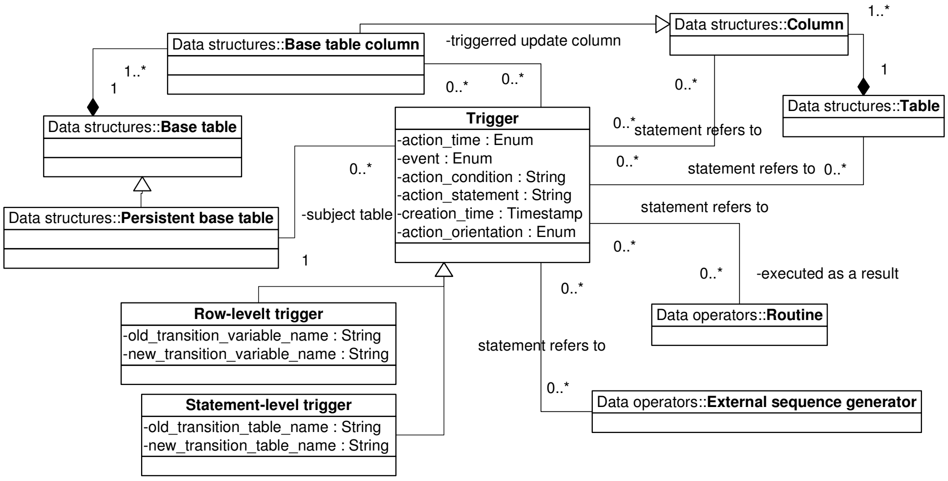

1..*
Data structures::Column
Data structures::Base table column
-triggerred update column
1..*
1
1
0..*
0..*
0..*
Data structures::Table
Data structures::Base table
Trigger
-action_time : Enum
-event : Enum
-action_condition : String
-action_statement : String
-creation_time : Timestamp
-action_orientation : Enum
0..*
statement refers to
0..*
0..*
statement refers to
0..*
Data structures::Persistent base table
-subject table
1
statement refers to
0..*
0..*
-executed as a result
0..*
Data operators::Routine
Row-levelt trigger
-old_transition_variable_name : String
-new_transition_variable_name : String
statement refers to
0..*
Data operators::External sequence generator
Statement-level trigger
-old_transition_table_name : String
-new_transition_table_name : String

Figure 23 Triggers in OR_{SQL}

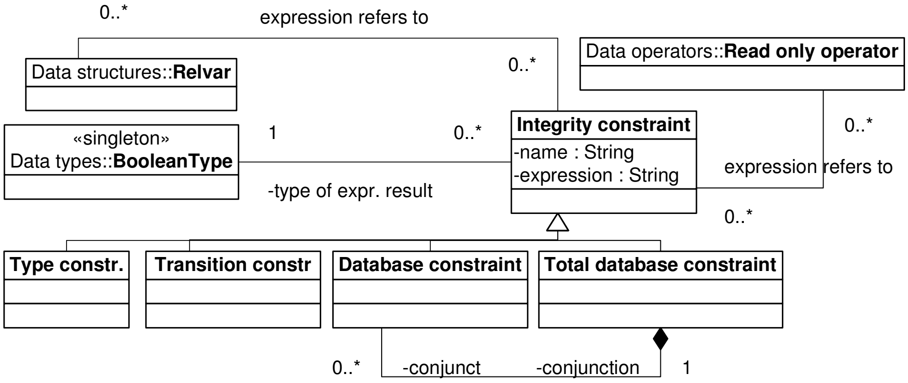


Figure 24 Constraints in OR_{TTM}

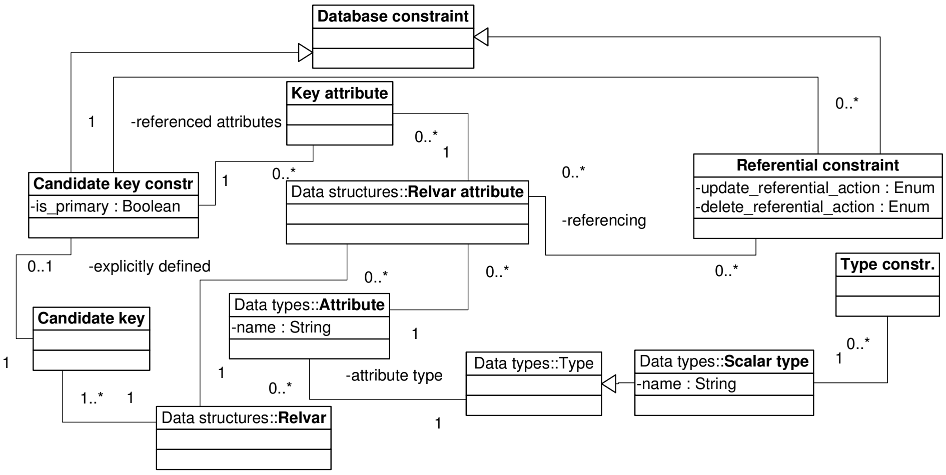


Figure 25 Candidate key and referential constraints in OR_{TTM}

Table 3 Mapping of OR_{SQL} and OR_{TTM} metaclasses that belong to the packages "Data integrity"

ID	OR_{SQL} metaclass	OR_{TTM} metaclass that represents the most similar concept
1	Assertion	Database constraint
2	Candidate key	Candidate key
3	CHECK constraint	Database constraint
4	Constraint	Integrity constraint
5	Primary key constraint	Candidate key constraint
6	Referential constraint (Foreign key constraint)	Referential constraint (Foreign key constraint)
7	Table check constraint	Database constraint
8	Table constraint	Database constraint
9	Unique column	Key attribute
10	Unique constraint	Candidate key constraint

Construct redundancy in OR_{SQL}:

- *Assertion*, *CHECK constraint*, *Table check constraint* and *Table constraint* in OR_{SQL} vs. *Database constraint* in OR_{TTM}.
- *Primary key constraint* and *Unique constraint* in OR_{SQL} vs. *Candidate key* constraint in OR_{TTM}.

Construct deficit in OR_{TTM}: *Domain constraint*, *Row-level trigger*[(1)], *Statement-level trigger*[(1)], *Trigger*[(1)], *Referenced column*[(2)], *Referencing column*[(2)].

[(1)] - Authors of OR_{TTM} are not strictly against triggered procedures, but find that this feature is not foundational part of data models. It is possible to use triggered procedures in case of different data models and therefore we can say that this feature is orthogonal to a data model.

[(2)] - These two metaclasses were created in the OR_{SQL} metamodel in order to show that columns have ordinal positions in a referential constraint. OR_{TTM}, on the other hand, tries not to complicate the language and prescribes that we have to refer to an attribute only by using its name and not by its ordinal position.

Construct deficit in OR_{SQL}: *Transition constraint*, *Type constraint*.

In addition, we note that the OR_{TTM} metaclass *Total database constraint* does not have one clearly corresponding metaclass in the current OR_{SQL} metamodel. However, it is possible to create this metaclass as an abstraction without violating the principles of OR_{SQL}.

1.3.4.1 Discussion

Both OR_{TTM} and OR_{SQL} allow us to create *declarative constraints* by using nonprocedural database programming language. Such a constraint specifies a rule, but does not prescribe a DBMS how to check it. In OR_{SQL}, it is also possible to implement constraints by creating trigger procedures that can contain statements of imperative language (SQL procedural extensions or other) (see Figure 23). Greenfield et al. (2004, p. 227) propose the definition: "An imperative specification describes instructions to be executed without describing the desired results of execution." "An SQL Trigger is a named chain reaction that you set of with an SQL-data change statement." (Gulutzan and Pelzer, 1999, p. 463) Authors of OR_{TTM} do not prohibit triggers (they use the concept "triggered procedure"). However, they take the position that the use of triggered procedures is in many cases unnecessary if declarative constraints are fully supported by a DBMS.Type constraint in OR_{TTM} limits the values that belong to this type (see Figure 24). In addition, OR_{TTM} permits creation of a type as a subtype of another type by using a *specialization by constraint*. It is not possible to create *declarative* type constraints in OR_{SQL} or to create types by using the specialization by constraints. According to Mattos and DeMichiel (1994) specialization by constraints should be prohibited because it requires, for example, overloading of operators. Date and Darwen (2000, Appendix G) discuss negative implications of the approach that prohibits specialization by constraint.

In OR_{SQL}, the correctness of a value that belongs to a user-defined type can be checked by the SQL-invoked methods of this type. Methods can be implemented by using some *imperative language* (SQL procedural extensions or other). Note that the OR_{SQL} domain constraint (see Figure 22) is not a counterpart to the OR_{TTM} type constraint because a domain in OR_{SQL} is not a data type but a reusable specification of column properties.

A database constraint determines the legal values that a set of database relation variables can have (see Figure 24). Earlier versions of The Third Manifesto (for example, Date and Darwen (2000)) distinguished a relvar- and database constraints. A relvar constraint constrains the values of exactly one relvar and a database constraint constrains the values of more than one relvars. In addition, earlier versions of The Third Manifesto used the concept "attribute constraint". An attribute constraint is enforced by the declaration that an attribute has a type.

The values of the relvars that are referenced by the Boolean expression of database constraints must be such that the result of the evaluation of this expression is *TRUE*. In OR_{SQL}, a table check constraint (see Figure 22) constrains values in a base table with which it is associated. Its expression can contain a subquery that can refer to other tables as well. An expression that is associated with an assertion in OR_{SQL} refers to one or more tables. Constraints in OR_{SQL} are satisfied if the result of the evaluation of their expression is not *false* (that means it is either *true* or *unknown*).

If a variable VA that has a value V1 obtains a new value V2, then the transition constraint (Date 2003, p. 268) (see Figure 24) checks whether such transition of values was legal. It is not possible to create *declarative* transition constraints in OR_{SQL}, but it is possible to implement them by using triggers.

Differences of candidate keys in OR_{SQL} (see Figure 22) and OR_{TTM} (see Figure 25) (Date and Darwen, 2000, p. 431):

- OR_{SQL} states that the set of candidate keys in a table is not empty. However, the declaration of a unique constraint in a base table is not mandatory and it is not possible to explicitly declare the keys in the viewed tables. OR_{TTM} requires that each relvar should have at least one candidate key. An explicit specification of at least one candidate key is mandatory in case of the relvars that are not virtual relvars. It is optional in case of a virtual relvar because an $ORDBMS_{TTM}$ should be able to determine the candidate keys of a virtual relvar based on the candidate keys of its underlying real relvars.
- OR_{SQL} advocates the selection of one candidate key as a primary key. In OR_{TTM}, the selection of the first among the equals is not necessary.
- In OR_{SQL}, a primary key or a uniqueness constraint must be declared over *one or more* columns. OR_{TTM} specifies two relvars (*TABLE_DEE* and *TABLE_DUM*) that have no attributes. These relvars have exactly one candidate key that has no components (Date, 195, p.129).
- OR_{SQL} permits one key to be a proper subset of another that is violation of the *irreducibility* property of a candidate key.

- OR_{SQL} specifies that the unique constraint descriptor must include the ordinal position of a column within a constraint (Melton, 2003, p. 65). In OR_{TTM}, this ordinal position is not important.

In OR_{TTM}, constraints are checked right after the end of the statement that might cause their violation. OR_{TTM} permits a multiple assignment statement that assigns a new value to more than one relvar. In that case a constraint is checked after more than one relvar obtains a new value. In OR_{SQL} it is possible to defer the checking of the constraints to the end of transaction (see attribute *is_deferrable* of metaclass *Constraint* in Figure 22) and determine that the isolation level of transactions is as high as possible - SERIALIZABLE.

1.3.5 Comparison of Data Operators

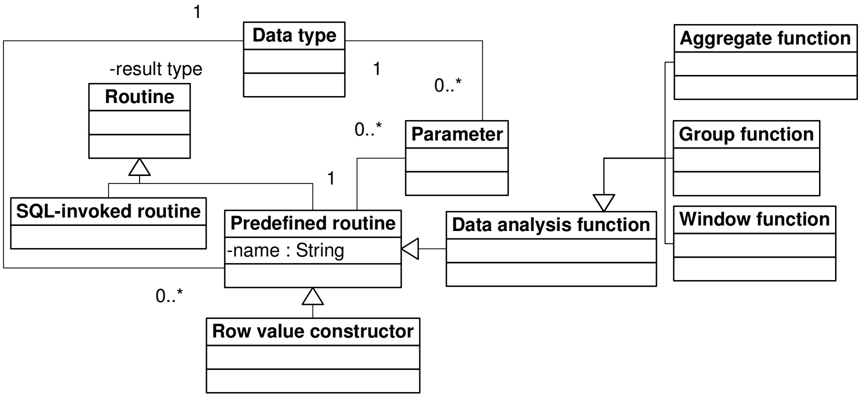


Figure 26 Predefined routines in OR_{SQL}

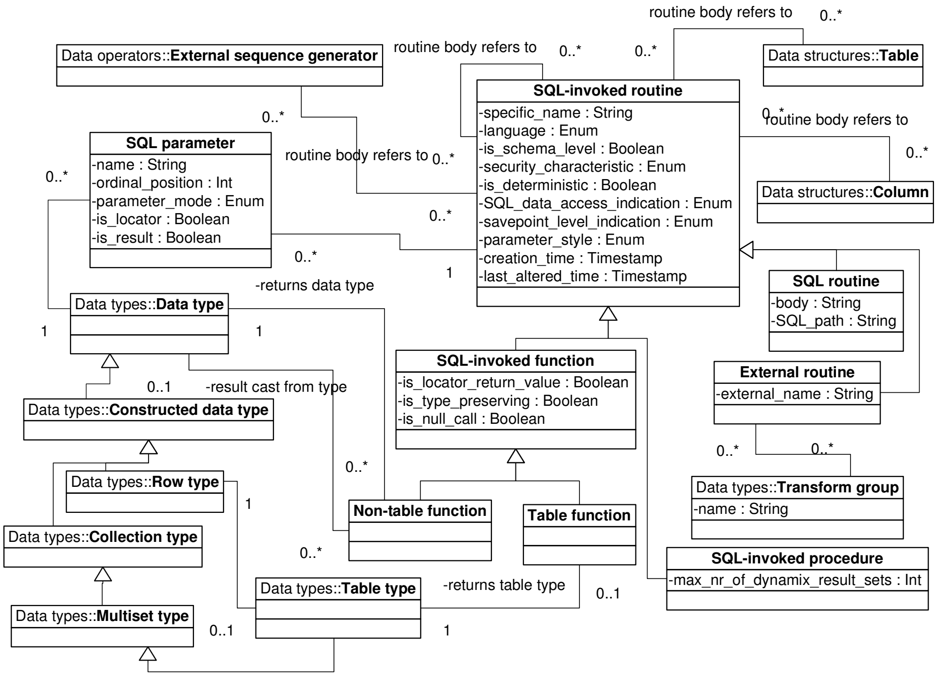


Figure 27 SQL-invoked routines in OR$_{SQL}$

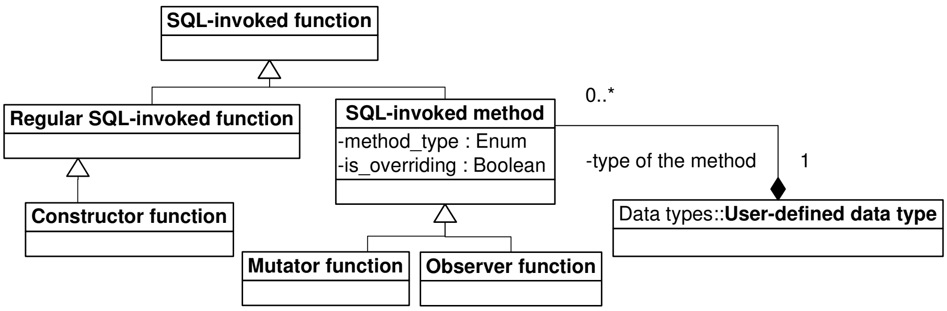


Figure 28 SQL-invoked functions in OR$_{SQL}$

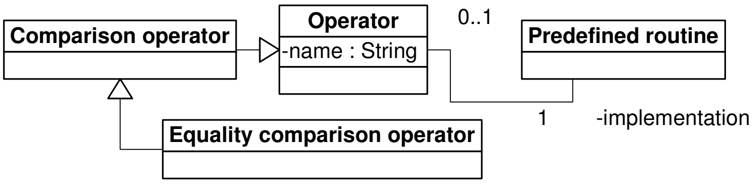


Figure 29 Operators in OR$_{SQL}$

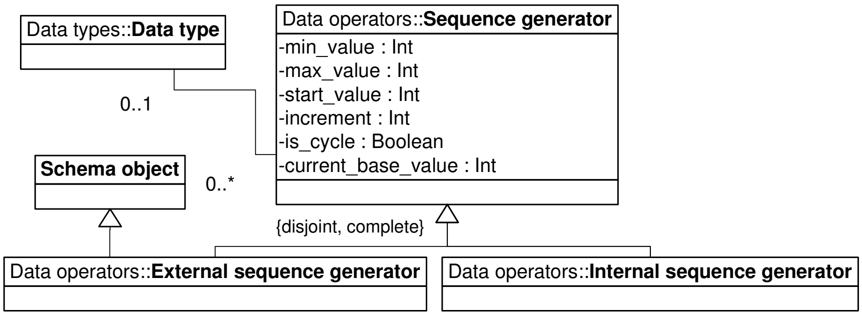


Figure 30 Sequence generators in OR$_{\text{SQL}}$

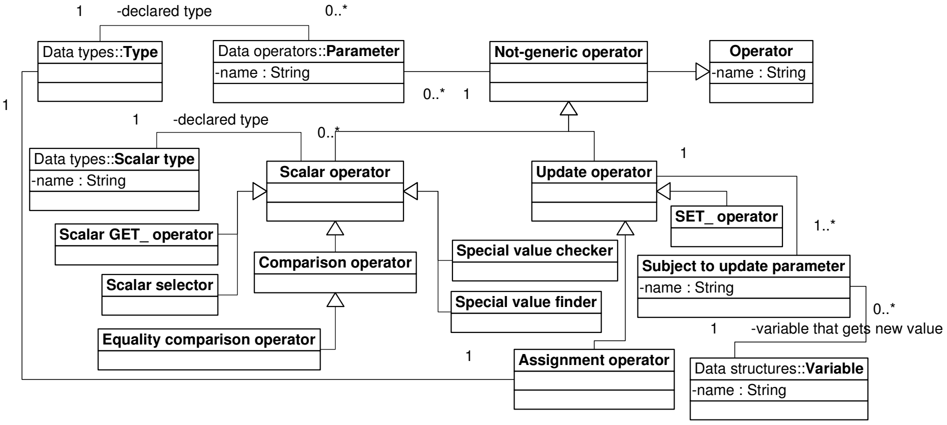


Figure 31 Scalar and update operators in OR$_{\text{TTM}}$

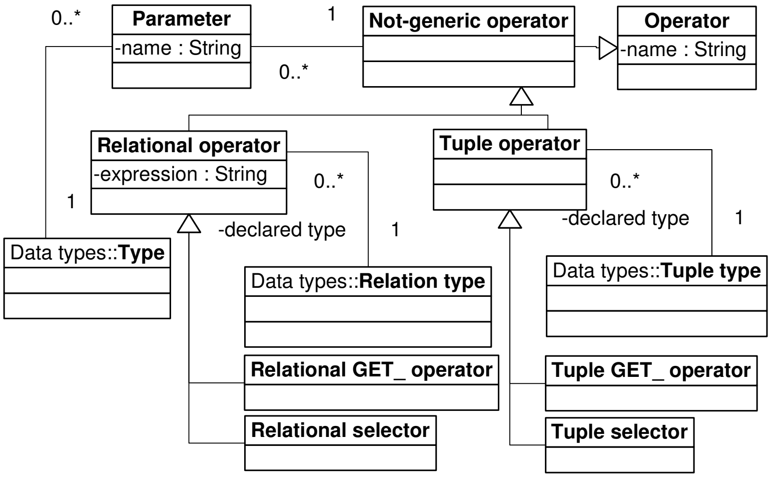


Figure 32 Relational and tuple operators in OR$_{\text{TTM}}$

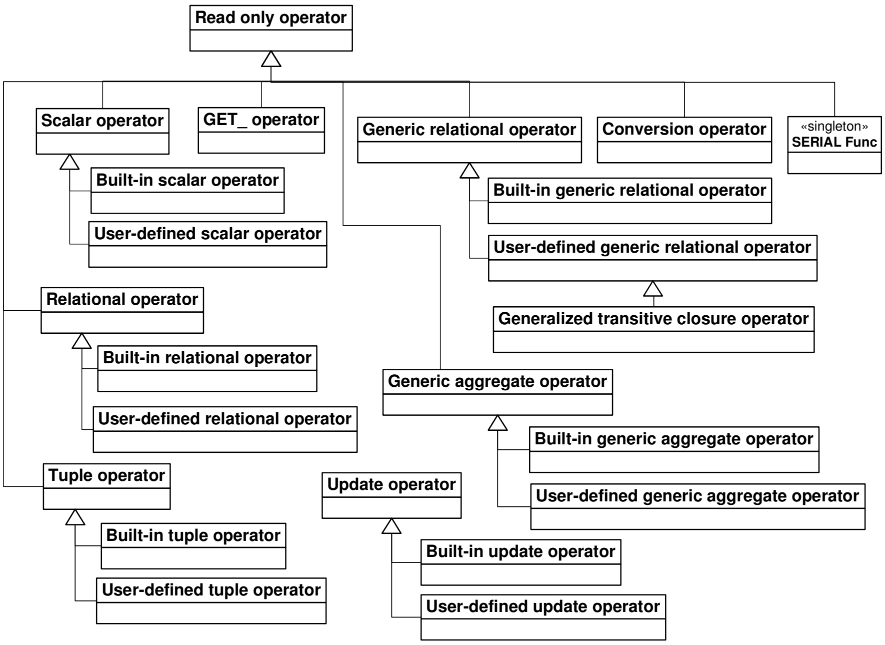

Read only operator
Scalar operator
GET_ operator
Generic relational operator
Conversion operator
«singleton»
SERIAL Func
Built-in scalar operator
User-defined scalar operator
Built-in generic relational operator
User-defined generic relational operator
Relational operator
Generalized transitive closure operator
Built-in relational operator
Generic aggregate operator
User-defined relational operator
Built-in generic aggregate operator
Tuple operator
Update operator
User-defined generic aggregate operator
Built-in tuple operator
Built-in update operator
User-defined tuple operator
User-defined update operator

Figure 33 Built-in and user-defined operators in OR$_{TTM}$

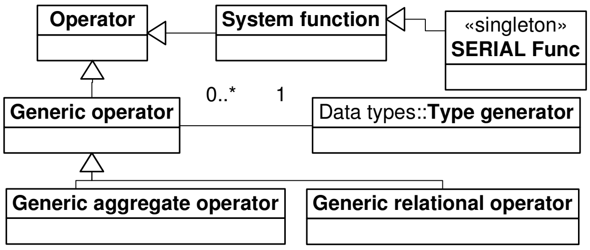

Operator
System function
«singleton»
SERIAL Func
0..*
1
Generic operator
Data types::Type generator
Generic aggregate operator
Generic relational operator

Figure 34 Generic operators and system functions in OR$_{TTM}$

Table 4 Mapping of OR$_{SQL}$ and OR$_{TTM}$ metaclasses that belong to the packages "Data operators"

ID	OR$_{SQL}$ metaclass	OR$_{TTM}$ metaclass that represents the most similar concept
1	Aggregate function	Built-in generic aggregate operator
2	Comparison operator	Comparison operator
3	Data analysis function	Read-only operator
4	Equality comparison operator	Equality comparison operator
5	External routine	User-defined scalar operator User-defined tuple operator User-defined relational operator User-defined update operator
6	External sequence generator	SERIAL func
7	Internal sequence generator	SERIAL func

ID	OR_{SQL} metaclass	OR_{TTM} metaclass that represents the most similar concept
8	Mutator function	SET_ operator
9	Non-table function	User-defined scalar operator
10	Observer function	GET_ operator Scalar GET_ operator Tuple GET_ operator Relation GET_ operator
11	Operator	Built-in scalar operator
12	Parameter	Parameter
13	Parameter (where parameter mode is OUT)	Subject to update parameter
14	Predefined routine	Built-in scalar operator
15	Regular SQL-invoked function	User-defined scalar operator User-defined tuple operator User-defined relational operator
16	Routine	Operator
17	Row value constructor	Tuple selector
18	Sequence generator	SERIAL func
19	SQL-invoked function	User-defined scalar operator User-defined tuple operator User-defined relational operator
20	SQL-invoked method	Read-only operator Update operator
21	SQL-invoked procedure	User-defined update operator
22	SQL-invoked routine (Schema-level routine)	User-defined scalar operator User-defined tuple operator User-defined relational operator User-defined update operator
23	SQL parameter	Parameter
24	SQL routine	User-defined scalar operator User-defined tuple operator User-defined relational operator User-defined update operator
25	Table function	User-defined relational operator
26	Window function	Relational operator

Construct redundancy in OR_{SQL}:

- *Operator*, *Predefined routine* in OR_{SQL} vs. *Built-in scalar operator* in OR_{TTM}.
- *External sequence generator*, *Internal sequence generator*, *Sequence generator* in OR_{SQL} vs. *SERIAL func* in OR_{TTM}.
- *External routine*, *Regular SQL-invoked function*, *SQL-invoked function*, *SQL-invoked routine*, *SQL-routine*, *Table function* in OR_{SQL} vs. *User-defined relational operator* in OR_{TTM}.
- *External routine*, *Non-table function*, *Regular SQL-invoked function*, *SQL-invoked function*, *SQL-invoked routine*, *SQL-routine* in OR_{SQL} vs. *User-defined scalar operator* in OR_{TTM}.

- *External routine*, *Regular SQL-invoked function*, *SQL-invoked function*, *SQL-invoked routine*, *SQL-routine* in OR_{SQL} vs. *User-defined tuple operator* in OR_{TTM}.
- *External routine*, *SQL-invoked procedure*, *SQL-invoked routine*, *SQL-routine* in OR_{SQL} vs. *User-defined update operator* in OR_{TTM}.

Construct deficit in OR_{TTM}: *Constructor function*, *Group function*.

Construct deficit in OR_{SQL}: *Assignment operator*, *Built-in relational operator*[(1)], *Built-in tuple operator*, *Built-in update operator*, *Generalized transitive closure operator*, *Relational selector*, *Special value finder*, *Special value checker*, *System function*, *Tuple operator*, *User-defined generic aggregate operator*, *User-defined generic relational operator*.

[(1)] - OR_{SQL} specifies built-in operators that operate on multisets and return multisets.

In addition, we note that the following OR_{TTM} metaclasses do not have one clearly corresponding metaclass in the current OR_{SQL} metamodel: *Built-in generic relational operator*, *Generic aggregate operator*, *Generic operator*, *Not-generic operator*, *Generic relational operator*, *Relational operator*, *Scalar operator*. However, we think that it is possible to create these metaclasses as abstractions without violating the principles of OR_{SQL}.

1.3.5.1 Discussion

Both OR_{TTM} and OR_{SQL} provide the means for performing operations in a database. Firstly, integrity constraints and triggers that are associated with the relvars/tables can be used in order to prohibit recording of incorrect data in a database (see section 1.3.4). Checking of these constraints is an operation. Triggers can also be used in order to trigger execution of some routine in a database.

In addition, OR_{TTM} allows us to use operators and OR_{SQL} allows the use of operators and routines for making operations in a database. OR_{TTM} uses the general concept "operator". OR_{SQL}, on the other hand, uses the concepts "operator", "function", "procedure", and "method" for the same subject matter and therefore actually makes the specification much more complicated. OR_{SQL} uses the concept "operator" in order to describe the infix, prefix or postfix notation for calling a function (see Figure 29). OR_{SQL} does not permit users to create new "operators".

"Functions and procedures correspond very roughly to our read only and update operators, respectively; methods behave like functions, but are invoked using a different syntactic side." (Date, 2003, p. 132) Invocations of the OR_{TTM} scalar (see Figure 31), relational and tuple operators (see Figure 32) produce scalar values, relations and tuples, respectively. These operators are read-only and cannot change the values of relvars in a database or modify the structure of a relvars. Routine body of OR_{SQL} SQL-invoked function can contain SQL executable statements like INSERT/UPDATE/DELETE statements (Melton, 2003, p. 676).

An update operator in OR_{TTM} (see Figure 31) is not typed at all because its invocation does not produce a value as a result. On the other hand, SQL-invoked procedure can have output parameters. SET_ operator that is a kind of update operator is used in OR_{TTM} in order to update scalar variable so that if its value before update is v and after update is v', then the possible representations of these values differ by only one component. OR_{SQL} specifies mutator functions for the same purposes. The invocation of a mutator function returns a value (Melton, 2003, p. 8).

OR_{TTM} specifies assignment operators that allow us to assign new values to variables. We decided to model assignment operator as the subclass of *update operator* (see Figure 31). "An update operator is an operator that, when invoked, is allowed to update at least one variable that is not purely local to that operator." (Date and Darwen, 2006) An assignment operator is an update operator that has two parameters and assigns a new value to exactly one variable. One of the parameters allows us to access a variable and is therefore called "subject to update parameter". Another parameter allows us to pass the new value of the variable to the operator. OR_{SQL} does not use the concepts "variable" and "assignment operator", but it specifies statements for modifying data in a table.

A method is associated with a user-defined type in OR_{SQL} (see Figure 28). OR_{TTM} on the contrary advocates that types and operators should be unbundled and that security mechanism helps to give to an operator access to the internals of instances of any number of types.

OR_{TTM} permits creation of relational or tuple operators, the result of which has a specific relation type or tuple type, respectively (see Figure 32). If this operator has no parameters, then it is an analogy to a virtual relvar (view). OR_{SQL} permits creation of views. In addition, OR_{SQL} permits creation of table functions. A table function can have input parameters and its result has a table type (see Figure 27). Actually it is a composite type MULTISET(ROW(...)) that is created by using two different type constructors. One difference between the relational operators in OR_{TTM} and the table functions in OR_{SQL} is that the former eliminates automatically repeating tuples from the result, but the latter needs explicit specification in order to remove repeating rows from the result.

OR_{SQL} specifies different window functions. One of them is a rank function. Date and Darwen (2000, p. 210) demonstrate that the result that is expected from a rank function can be achieved in terms of usual relational algebra operators. In addition, they propose a relational operator RANK, which is a kind of shorthand.

Relational algebra operators UNION, INTERSECT and JOIN are the examples of generic relational operators. OR_{SQL} allows us to perform relational algebra operations, but does not use the concept "operator" in that context. OR_{SQL} does not permit creation of user-defined generic operators as OR_{TTM} does (see Figure 34). If a generic operator GO is associated with a type generator TG then it is possible to apply GO to any value that has a type which is generated by TG. For example, generic relational operator works for all relations and allows us to derive relations from other relations.

1.3.6 Observations

This section contains some observations that we made during the creation of the metamodels and comparison of them.

We created the mapping between the metaclasses (see Table 1 - Table 4). In many cases, it was possible to find pairs of metaclasses, which present constructs that have a *relatively similar* meaning. However, we think that none of these pairs has semantic equivalence.

It was sometimes difficult to determine a package for a particular metaclass in case of $\mathrm{OR}_{\mathrm{SQL}}$. We found three metaclasses in $\mathrm{OR}_{\mathrm{SQL}}$ that we were unable to classify – *Object*, *Schema object* and *Structural component*. These metaclasses "represent an abstraction of properties that are common to multiple disjoint types." (Guizzardi, 2005, p. 112) In addition, these disjoint types (metaclasses) belong to different packages. For example, user-defined data types, base tables and SQL-invoked routines are schema objects and therefore *Objects* as well. However, the metaclass *User-defined data type* belongs to the package *Data types*, the metaclass *Base table* belongs to the package *Data structures* and the metaclass *SQL-invoked routine* belongs to the package *Data operators*. *Structural component* represents a generalization of metaclasses *Field*, *Attribute* and *Column*. The first two belong to the package *Data types* and the latter belongs to the package *Data structures*.

Table 5 shows some metaclasses that belong to a package, but we found it difficult to determine the most appropriate package for them. We do not claim that this classification is definitive, but rather it reflects our current understanding of the subject matter. Mark "*" in column *Possible packages* identifies a package that we finally chose for this metaclass.

Table 5 Metaclasses of the $\mathrm{OR}_{\mathrm{SQL}}$ metamodel that we found difficult to classify

Metaclass	Possible packages	Comment
Domain	Data structures * Data types	"A domain definition specifies a data type." (Melton, 2003, p., 49) A domain is not a data type (data types), but rather a reusable specification of column properties (data structures).
Sequence generator (and its subclasses)	Data operators* Data structures Data integrity	"A sequence generator is a mechanism for generating successive exact numeric values, one at a time." (Melton, 2003, p. 77) It can be used for generating key (data integrity) values that are recorded in some table (data structures). We choose to think about it as a special routine (data operators) that is similar to $\mathrm{OR}_{\mathrm{TTM}}$ system function "SERIAL".

Metaclass	Possible packages	Comment
Path and Path element	Data operators Data structures*	A path determines the search order for user-defined routines (data operators) (Melton, 2003, p. 473). However, its specification is part of schema definition (data structures).
Table type	Data operators Data types*	The SQL standard uses the concept "table type" in connection with table functions (data operators). The wording " <returns type> ::=<returns data type> [<result cast>]| <returns table type>" (Melton, 2003, p. 676) gives an impression that table type is not a data type. If <returns type> RST specifies TABLE and TCL is the <table function column list>, then "RST is equivalent to the <returns type> ROW TCL MULTISET." (Melton, 2003, p. 678) Therefore, we decided to model table type as a data type that is created by using two type constructors – MULTISET and ROW.
Transform group	Data operators Data types*	A transform is an object that associates a user-defined type (Data types) with two SQL-invoked functions (Data operators) "that are automatically invoked when values of user-defined types are transferred from SQL-environment to host languages or vice-versa." (Melton, 2003, p. 42)
User-defined cast	Data operators Data types *	A user-defined cast is an object that associates a source data type, a target data type (Data types) and a routine (Data operators) that is invoked if casting of a value with source data type is needed.
User-defined ordering	Data operators Data types*	A user-defined ordering is an object that can accompany a user-defined data type (Data types). It specifies the method how to compare two UDT values and optionally determines a routine that performs comparison (Data operators).
Trigger (and its subclasses)	Data integrity* Data operators Data structures	A trigger is a specification for a given action (Data operators) to take place every time a given operation takes place on a given object (Data structures). We decided to place these classes to the package "Data integrity" because triggers can be used in order to implement integrity constraint. We have to note that it is possible to use triggers for other purposes as well.

We placed $\mathrm{OR}_{\mathrm{TTM}}$ metaclasses *Database* and *Catalog* and $\mathrm{OR}_{\mathrm{SQL}}$ metaclasses *Cluster*, *Catalog*, *Schema*, *DEFINITION_SCHEMA* and *INFORMATION_SCHEMA* to the package *Data structures*. Another option is to create separate package *Container* for these metaclasses because they represent actually composites of objects that make up a database.

The previously described problems are one example that SQL:2003 and therefore OR_{SQL} as well are too complex. SQL:2003 specification is much longer and uses more concepts (see section 1.3.7) than The Third Manifesto.

Descriptions of data types, data structures, data integrity and data operators depend on each other. It is not possible to understand concepts from one package without studying concepts in the other packages. For example, the metamodels contain metaclasses that model the values. Among other things, a type is a set of values that the computer system is able to represent (Date, 2003). Therefore, we decided to place the metaclasses about the values to the package *Data types* (see section 1.3.2) and not to create separate package *Data values*. However, the description of the *values* of relation variables is placed to section *Data structures* (see section 1.3.3) because it is no possible to explain this topic without explaining the concept *relation variable*.

Different levels of abstraction in the metamodels make it difficult to compare them. For example, OR_{TTM} uses the concepts *Relation heading* and *Body*. The corresponding concepts in OR_{SQL} would be *Table heading* and *Table body*, but OR_{SQL} does not use them. However, at the higher level of abstraction, it is possible to imagine that a table has a heading and a body.

If two data models use the same concept, then it does not mean that the constructs behind this concept are semantically equivalent or similar. For example, both OR_{TTM} and OR_{SQL} use the concept *domain*. However, in OR_{TTM} it means a data type and in OR_{SQL} a reusable specification of column properties.

1.3.7 Metrics Values

Rossi and Brinkkemper (1996) propose a set of metrics, the values of which can be calculated based on the metamodels and that help to compare the complexity of system development methods and techniques. It is also possible to use these metrics in case of the data models if their metamodels are available.

This section contains the values of three types of metrics – the number of metaclasses, the number of attributes of metaclasses and the sum of these values (see Table 6). The metrics values are calculated for OR_{SQL} and OR_{TTM}.

For the comparison purposes we also present the metrics values for the underlying data model of SQL:1992. We do not present the metamodel of the underlying data model of SQL:1992 in this dissertation. We calculate these values based on the OR_{SQL} metamodel by taking into account the new features that were added to SQL:1999 and SQL:2003. We use appendix D (Incompatibilities with SQL-92) of the book about SQL:1999 (Gulutzan and Pelzer, 1999) and the annex E (Incompatibilities with ISO/IEC 9075:1999) of SQL:2003's Part 2: SQL/Foundation (Melton, 2003) in order to collect data for this purpose.

In case of these metrics, bigger values mean bigger complexity. However, Rossi and Brinkemper (1996) write: "the metrics by themselves cannot be used to judge the “goodness” or the appropriateness for the task of the method" and should be used together with other comparison methods. We have followed this advice (see section 1.3.1).

The underlying data model of SQL:1992 has smaller metrics values as compared to OR_{SQL} and OR_{TTM}. In this case, smaller metrics values (and complexity) are caused by the lack of many important features (for example, the lack of possibilities to declare new types and operators). This actually makes creation of applications that use a database more difficult because more work has to be done by an application. Section 2.3 and the work of Eessaar (2006c) contain a literature-based overview of problems of the underlying data model of SQL:1992 or earlier versions of the SQL standard. The OR_{SQL} and OR_{TTM} data models try to solve many of these problems.

Table 6 Metrics values that are calculated based on the metamodels of data models

	SQL:1992	OR_{SQL}	OR_{TTM}
The number of metaclasses that deal with the *data types* + the number of their attributes	10+10=20	37+21=58	27+4=31
The number of metaclasses that deal with the *data structures* + the number of their attributes	18+12=30	26+17=43	16+4=20
The number of metaclasses that deal with the *data integrity* + the number of their attributes	13+11=24	16+21=37	9+5=14
The number of metaclasses that deal with the *data operators* + the number of their attributes	8+2=10	27+32=59	42+5=47
The number of metaclasses that we cannot classify + the number of their attributes	3+3=6	3+3=6	-
∑	**52+38=90**	**109+94=203**	**94+18=112**

All three metrics values of OR_{SQL} are bigger than the corresponding metrics values of OR_{TTM}. The amount of metaclasses in the OR_{SQL} and OR_{TTM} metamodels is quite similar. The metaclasses of the OR_{SQL} metamodel have significantly more attributes as compared to the metaclasses of the OR_{TTM} metamodel. It points to the bigger complexity of OR_{SQL} compared to OR_{TTM} and is caused by the following reasons:

- A database designer who designs a database based on OR_{SQL} has more opportunities to "tune" the schema objects.
- OR_{SQL} makes use of such properties that according to the creators of OR_{TTM} complicate the data model without providing an advantage. For example, OR_{SQL} and SQL database language attaches significance to the ordinal positions of structural components and columns that participate in a primary key, uniqueness or referential constraint.

In this case, smaller metrics values of OR_{TTM} compared to OR_{SQL} do not mean that OR_{SQL} is "better". Firstly, analysis of similarities and discrepancies of OR_{SQL} and OR_{TTM} shows that despite the lack of some constructs in OR_{TTM} it is still possible to use an $ORDBMS_{TTM}$ in the cases that require the use of these constructs in an ORDBMS$_{SQL}$. We just have to use some construct (that is actually present in OR$_{SQL}$) in a way that is not possible in OR$_{SQL}$ due to its limitations. In addition, OR$_{SQL}$ violates orthogonality principle (see section 1.3.8). Therefore, we have to agree with Rossi and Brinkemper (1996) that it is not possible to determine "goodness" of a method or data model by using only the metrics values (see Table 6).

1.3.8 Orthogonality Principle in Language Design

Date and Darwen (2000, p. 505) explain that a programming language that displays orthogonality provides "(a) a comparatively *small* set of primitive constructs together with (b) consistent rules for putting those constructs together, and (c) every possible combination of those constructs is both legal and meaningful (*in other words, a deliberate attempt has been made to avoid arbitrary restrictions*)." (Date and Darwen, 2000, p. 505) [*Italics* added by the author] It is also true in case of abstract programming languages like data models.

An advantage of OR$_{TTM}$ compared to OR$_{SQL}$ is that OR$_{TTM}$ is based on the small set of core concepts that makes the model much easier to understand (see requirement (a) of orthogonality)(see also Figure 3). Unlike OR$_{SQL}$, OR$_{TTM}$ uses the concepts "variable" and "operator" as a basis of specification of its data structures and data operators, respectively. Some of the concepts are metaphors that help to make a data model easier to understand to people with a programming background. Examples of such concepts are "variable" and "assignment operator". Rittgen (2006) recommends to use metaphors in the software engineering in order to make a particular topic more understandable because "they resort to knowledge that is rooted in common sense and therefore shared by everybody." (Rittgen, 2006, p. 434)

Date and Darwen (2000, p. 435-436) illustrate SQL violations of the orthogonality principle with a non-exhaustive list of examples. These examples are about the requirement (c) of orthogonality. We found additional examples. We present the problem in OR$_{SQL}$ as well as the comment about the state of affairs in OR$_{TTM}$.

1. Attributes, fields and columns are structural components. A column can be associated with a domain, but an attribute or a field cannot be associated with a domain.

   Note: OR$_{SQL}$ should allow us to associate a domain with any structural component in order to better follow the orthogonality principle.

   OR$_{TTM}$: OR$_{TTM}$ does not use the constructs *field*, *column* and *domain*. An attribute can have a type that is either a built-in or a user-defined scalar type or a non-scalar type (tulpe- or relation type).
2. It is not possible to declare a default value to a field, but it is possible in case of attributes and columns.

   OR$_{TTM}$: A relvar attribute (virtual relvars are not excluded) can have a default value (Date and Darwen, 2000, p. 202). We cannot declare a default value to an attribute that belongs to the heading of a relation type.

3. It is possible to use generated columns but not generated attributes or fields.
   OR_{TTM}: A default value of a relvar attribute can be found by using some expression.
4. A nullability characteristic is part of the column descriptor but not part of an attribute or field descriptor (Melton, 2003, p. 49).
   OR_{TTM}: OR_{TTM} rejects the use of NULL's in order to present missing information.
5. A domain can be associated only with a predefined data type but not with a user-defined type or a constructed type.
   OR_{TTM}: OR_{TTM} uses the concepts *domain* and *type* as synonyms. Attributes that are in the heading of a relation- or tuple type or components of a possible representation of a scalar type have a type.
6. Both base tables and viewed tables have columns. However, it is not possible to declare a default value to the column of a viewed table. Together with the updateable views, it could allow us to record different default values in a column of a base table in the different situations.
   OR_{TTM}: A relvar attribute can have a default value (Date and Darwen, 2000, p. 202). The authors do not distinguish between a real- and a virtual database relvars in this case.
7. A base table or a viewed table cannot contain two or more columns with the same name in OR_{SQL}. However, a derived table that is derived from one or more other tables by the evaluation of a query expression can contain more than one column with the same name.
   OR_{TTM}: OR_{TTM} does not allow two attributes with the same name in a relvar, in a relation or in the heading of relation- or tuple type.
8. Table constraints can only be explicitly associated with the base tables but not, for example, with the viewed tables.
   OR_{TTM}: In OR_{TTM}, an expression of a database constraint contains names of (it refers to) one or more relvars.
9. It is possible to create temporary base tables but not temporary viewed tables.
   OR_{TTM}: OR_{TTM} specifies private and public application relvars. It does not allow us to create *temporary* real- or virtual database relvars.
10. It is possible to create a typed table based on a user-defined structured type but not based on a distinct type.
    OR_{TTM}: OR_{TTM} does not support typed tables.
11. Each typed table must have exactly one self-referencing column. If this typed table is a typed base table, then this column has an implicit uniqueness constraint. On the other hand, SQL permits not-typed base tables, which do not have any associated (explicitly or implicitly defined) uniqueness constraint.
    OR_{TTM}: OR_{TTM} does not support typed tables. However, each real relvar must have at least one explicitly defined candidate key.
12. A subject table of a trigger can only be a persistent base table. It cannot be a viewed table.

OR$_{TTM}$: OR$_{TTM}$ does not specify triggered procedures.

The data model evolves over time, some orthogonality violations disappear but others come into existence. For example, Date and Darwen (2000, p. 436) note based on SQL:1999 that only the surrogate key column of a typed base table can use "VALUES ARE SYSTEM GENERATED" option. However, SQL:2003 allows us to use identity columns in the not-typed base tables.

If a metamodel contains a generalization relationship between metaclasses (see Figure 35) so that some attributes and/or relationships are at the superclass level and some are at the subclass level, then it could be a sign of a *possible* violation of requirement (c) of the orthogonality principle. For example, in case of problem (10) we could replace the letters in the figure in the following way: A – User-defined type, B – Structured type, C – Distinct type, D – Typed table.

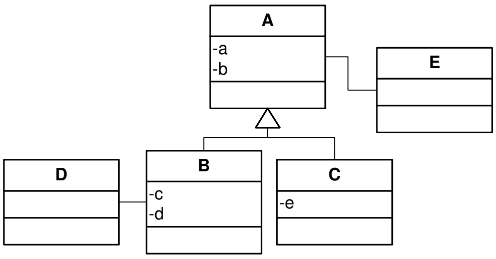


Figure 35 Constructs in a metamodel that identify possible violation of the orthogonality principle

The result of our research supports the opinion of Rossi and Brinkkemper (1996) that it is not possible to determine "goodness" of the model based only on metrics values. Metrics values show that OR$_{SQL}$ is more complex than OR$_{TTM}$. However, the designers of OR$_{TTM}$ have paid more attention to the principle of *orthogonality* as compared to the designers of OR$_{SQL}$.

1.4 Comparison with the Existing Sate of the Art

1.4.1 Object-Oriented Database Metamodel

Habela (2002) presents a metamodel of *Object-Oriented* DBMSs. On the other hand, we deal with the *object-relational* data models and present the metamodel-based comparison of two data models. Habela (2000) describes the roles of a metamodel. Firstly, he states that the metamodel must "support the understanding of the introduced data model by all parties" and therefore he presents a conceptual view of the discussed metamodel constructs as a UML class diagram (Habela, 2000, p. 66). This metamodel is a simplified and improved version of the ODMG (Object Data Management Group) metamodel. Large portion of the work of Habela (2000) is dedicated to the problem how to *implement* a metadata repository based on the metamodel. He claims that the presented metamodel is too complex and proposes the flattened metamodel structure that contains classes *MetaObject*, *MetaAttribute*, *MetaValue*, *MetaRelationship*. In general, he proposes to use the "Universal Data Model" approach. We explain the problems of this approach in section 3.1.4.

1.4.2 CIM Database Model

CIM (Common Information Model) is a conceptual information model that specifies different areas of information technology management. Part of CIM Database Model (DMTF CIM, 2006) is model of SQL Schema. It presents only some of the most basic concepts (*Object*, *CharacterSet*, *Schema*, *Trigger*, *Table*, *Domain*, *Constraint* and *UserDefinedType*) and mostly generalization relationships between them. We think that the generalization relationship between experimental classes *SqlDomain* and *SqlDomainConstraint* does not model correctly the semantics of the relationship between these two constructs (see Figure 14 in order to see how we modelled it).

The $\mathrm{OR}_{\mathrm{SQL}}$ metamodel that is presented in this dissertation is much more extensive. In addition, part of CIM Database model, which specifies SQL Schema contains concepts (for example, *ManagedElement*, *LogicalElement*) that are not used in the SQL standard.

1.4.3 Common Warehouse Metamodel

The relational package of the Common Warehouse Metamodel (CMW) Specification (OMG formal/03-03-02) contains a metamodel of a relational database. Calero et al. (2006) present a list of problems of this metamodel. We divide these problems into the following categories:

- The metamodel uses the concepts that the standard does not use. Examples of such concepts: *structural feature*, *named column set*, *SQL simple type*.
- The metamodel specifies a type of database objects (index) that the standard does not specify.
- The metamodel does not specify the types of database objects that are described in the standard. For example, the metamodel does not specify the sequence generators, which were incorporated first to SQL:2003. The metamodel also does not specify the domains, although they were present in SQL:1999. The metamodel uses the class *ChekConstraint* that can refer to zero or more tables, but it does not explicitly use the concept *Assertion*.
- The metamodel contains relationships that are not consistent with the standard. For example, a column is an attribute according to the model. However, the standard uses the concept *attribute* specifically in the context of structured types. It is true that columns of a typed table are created based on the attributes of a structured type, but not all the tables are typed tables.

In addition, we note that the metamodel does not always take into account the following guidelines for creating expressive and easily understandable domain models:

- "Relate conceptual classes with an association, not with an attribute." (Larman, 2002, p. 169) The metamodel contains foreign key attributes in conceptual classes.

- "If in doubt, define something as a separate conceptual classes rather as an attribute." (Larman, 2002, p. 170) For example, *character set* and *collation* are types of schema objects that are modeled as attributes.

1.4.4 SQL:2003 Ontology

Calero et al. (2006) present an ontology of the object-relational features of the SQL:2003 standard. We use the pre-publication version of their article. They do not use a special-purpose language for presenting the ontology (for example, OWL). Instead, they present UML class diagrams and well-formedness rules written in OCL. In comparison, semi-formal specification of the abstract syntax of a language contains metamodel and well-formedness rules (OMG formal/03-03-01, p. xxxiv). A metamodel can be presented by using UML class diagrams and well-formedness rules as prose and OCL expressions. Thus, what is the difference between UML class diagrams that are part of the SQL ontology and the metamodel that is presented in this dissertation? Gruber (1995) notes that one design criterion of ontologies requires that an ontology should require the *minimal ontological commitment*. Gruber (1995) comments that "since ontological commitment is based on consistent use of vocabulary" it can be minimized by "defining only those terms that are essential to the communication of knowledge consistent with that theory." Calero et al. (2006) follow this guideline and present only the most important concepts and their interconnections. On the other hand, a metamodel should describe all the language constructs without simplifications.

Calero et al. (2006) divide the ontology into two sub-ontologies – *DataTypes* and *SchemaObjects*. This classification is not precise enough. Firstly, a user-defined data type is also a schema object. In addition, some schema objects are data structures (for example, base table), some schema objects help to perform operations with data (for example, SQL-invoked routine) and some schema objects help to constrain data in a database (for example, constraint). Therefore, we think that it is more reasonable to use a classification that takes into account components of data models – *Data structures*, *Data operators*, *Data integrity* and in addition *Data types*.

We compared UML class diagrams that are part of the ontology and the $\mathrm{OR}_{\mathrm{SQL}}$ metamodel that is presented in this dissertation. We discovered that the ontology is sometimes not precise enough. Next, we present the problems together with the figures and comments. Each figure has two parts. Part (a) of a figure is a fragment of the ontology that is presented by Calero et al. (2006). Part (b) of a figure is a fragment of the metamodel that is presented in this dissertation (see section 1.3) and which in our view better reflects the SQL:2003 standard. The problems with the ontology (Calero et al., 2006) can be divided into two categories.

1. The fragments of ontology are too general and require additional well-formedness rules in order to reflect the standard correctly. In addition, rules in OCL are more difficult to understand than visual diagrams. The work of

Calero et al. (2006) actually does not contain these rules. Problems 1, 2, 5, 6 belong to this category.

2. The ontology does not reflect the standard correctly. Problems 3, 4, 7, 8 and 9 belong to this category.

We do not consider the missing classes (except in case of problem 8), attributes or associations because they are probably missing due to the simplifications. For example, the ontology does not show that the identity column is associated with an internal sequence generator.

Problem 1: There is an association between *Domain* and *Data type* (see Figure 36).

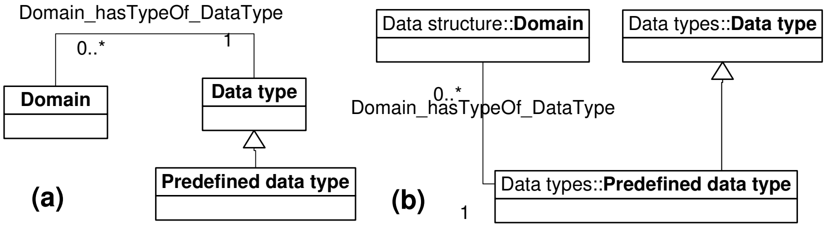


Figure 36 Possible ways to model Domains

Comment: A *Domain* must be associated with a *Predefined data type* (Melton, 2003, p. 603).

Problem 2: An element of a *Collection type* has an attribute *ordinal_position* (Figure 37).

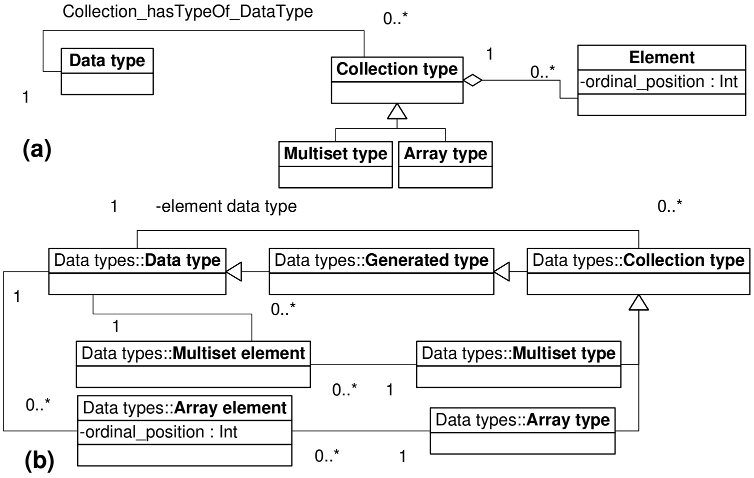


Figure 37 Possible ways to model collections

Comment: Collection type in SQL:2003 is either a multiset type or an array type. An array is an ordered collection, but a multiset is an unordered collection. "Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset." (Melton, 2003, p. 46)

Problem 3: There is no relationship to show inheritance between the collection types (see Figure 38).

Comment: Melton (2003) explains that a collection type CT2 can be a subtype of another collection type CT1 "if and only if CT1 is the same kind of collection as CT2 and the element type of CT2 is a subtype of the element type of CT1." (Melton, 2003, p. 45) Unfortunately, sections "<array value constructor>" (Melton, 2003, p. 285) and "<multiset value constructor>" (Melton, 2003, p. 291) do not contain any reference that it is possible to use subtypes in case of these type constructors. Therefore, it seems that the SQL standard is confusing and a possibility to use subtypes in this case is only theoretical. Calero et al. (2006) also point to this inconsistency in SQL:2003.

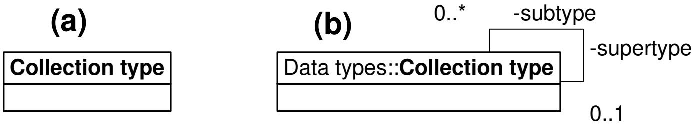


Figure 38 Possible ways to model collection types

Problem 4: Each *Method* must have exactly one associated *Data type* in case of relationship type "MethodResult_isCastedTo_DataType" (see Figure 39).

Comment: A method specification descriptor includes "The <result cast from type>, *if any*." (Melton, 2003, p. 39) The last part of this sentence indicates that some methods do not have an associated <result cast from type>.

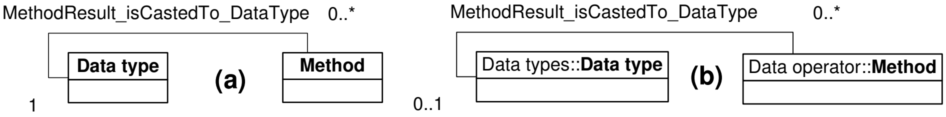


Figure 39 Possible ways to model methods

Problem 5: The model contains the relationship type according to which a *BaseTable* has zero or more subtables and zero or one supertable (see Figure 40).

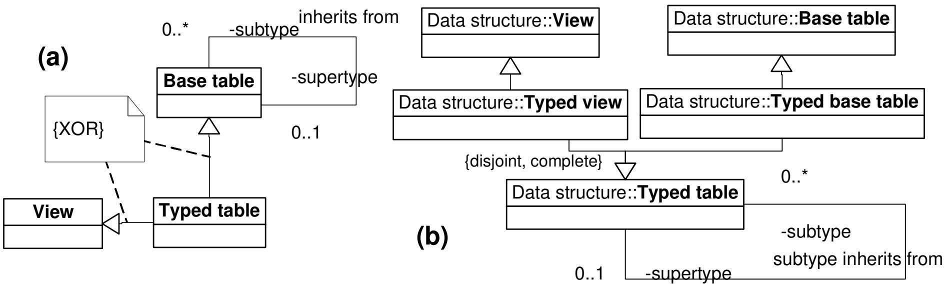


Figure 40 Possible ways to model typed tables

Comment: SQL supports the inheritance relationship only between typed tables (for example, Ta and Tb where Tb is supertable and Ta is subtable). A base table or a view can be a typed table (Melton, 2003, p. 54). "Both Ta and Tb shall be created on a structured type and the structured type of Ta shall be a direct subtype of the structured type of Tb." (Melton, 2003, p. 55)

Problem 6: The model contains the relationship type between *Generated column* and *Column* according to which each *Generated column* must be associated with one or more *Columns* (see Figure 41). These columns are referenced by the generation expression of a generated column. The ontology causes the following wrong impressions.

1. It is possible to specify generated column in a view definition the same way as in a base table definition.
2. Expression of a generated column *must* reference to at least one column.
3. Expression of a generated column can reference another generated column of same base table.

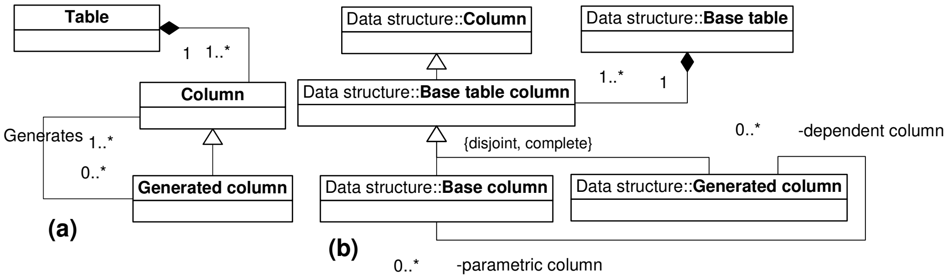


Figure 41 Possible ways to model generated columns

Comment: SQL:2003 states: "A column of a base table is either a base column or a generated column. /.../ A generation expression *can* reference *base columns* of the *base* table to which it belongs but cannot otherwise access SQL-data." (Melton, 2003, p. 57) In our view, the word "can" means that an expression refers to *zero* or more base columns. In addition, an expression cannot reference generated columns.

Problem 7: The ontology does not model correctly uniqueness constraint (see Figure 42). The ontology causes the following wrong impressions.

1. Each base table in SQL *must* have at least one unique constraint.
2. There is no reference whether subclasses of class Column are disjoint or not. It may give an impression that an identity column or a generated column cannot be unique column at the same time.
3. A column can have only one ordinal position within unique constraints.
4. It is possible to explicitly declare a unique constraint to a viewed table.

Comment: We think that the association between *Candidate key* and *Unique constraint* has wrong cardinality and participation constraints. SQL:2003 states that in each table "The set of candidate keys SCK is nonempty" (Melton, 2003, p. 75). However, SQL does not oblige database designers to declare uniqueness constraints to a base table. It is also comment to the impression 1. Therefore, each unique constraint is associated with exactly one candidate key and each candidate key is associated with zero or one unique constraint. Comment to the impression 3 is that a column can participate in more than one uniqueness constraint. Therefore, a column can have more than one different ordinal position within uniqueness constraints. Comment to the impression 4 is that only statements for creating or altering *base* tables permit creation of unique constraints.

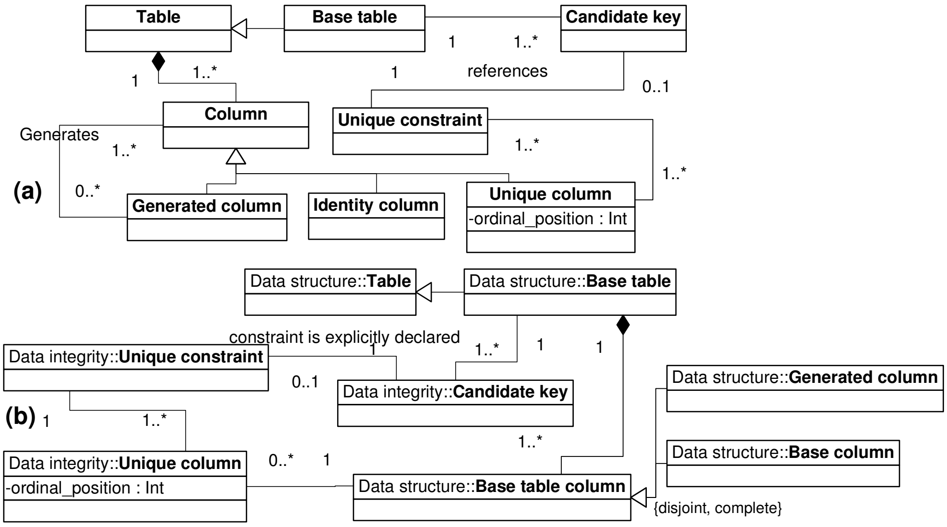


Figure 42 Possible ways to model uniqueness constraint

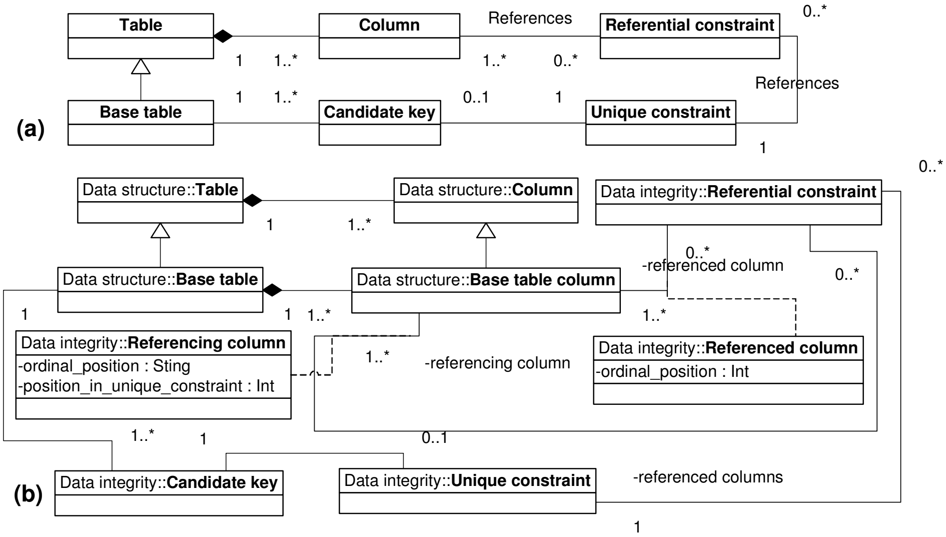


Figure 43 Possible ways to model referential constraint

Problem 8: The ontology does not model correctly referential constraints (see Figure 43).

1. It is not possible to understand whether the association between *Column* and *Referential constraint* models referencing columns or referenced columns.

2. The ontology causes the wrong impression that referenced columns or referencing columns can be any columns, including columns of derived tables and columns of transient tables.
3. Earlier we stated that we do not consider missing model elements. However, Calero et al. (2006) show that a unique column, a field and an attribute have an ordinal position. However, they do not show that OR_{SQL} pays attention to the ordinal positions of column names that are specified in the referential constraint.

Comments: SQL:2003 states "The referenced table shall be a base table." (Melton, 2003, p. 550) Therefore, referenced columns must be columns of a base table. A referencing table is identified by the containing <table definition> or <alter table statement> according to Melton (2003, p. 549). It is possible to create or alter only base tables by using such statements. Therefore, we conclude that referencing table is a base table and referencing columns must be columns of the base table. "The <referencing columns> shall contain the same number of <column name>s as the <referenced table and columns>. The i-th column identified in the <referencing columns> corresponds to the i-th column identified in the <referenced table and columns>. The declared type of each referencing column shall be comparable to the declared type of the corresponding referenced column." (Melton, 2003, p. 550) We conclude that a DBMS has to know the ordinal positions of column names in <referencing columns> and in <referenced table and columns> in order to enforce this rule.

Problem 9: The ontology states that each SQL-schema must contain at least one schema element (see Figure 44).

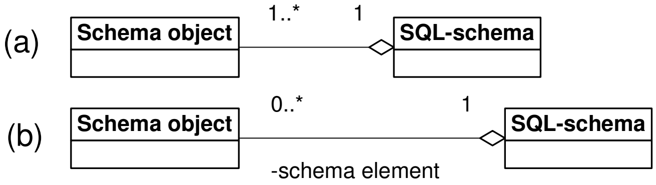


Figure 44 Possible ways to model schemas

Comment: Specification of schema elements is optional part of CREATE SCHEMA statement (Melton, 2003, p. 519).

1.4.5 SQL Definition Schema

A database must include a catalog (sometimes also called "system catalog" or "data dictionary"), which contains data that describes the data types, data structures, data operators and integrity rules that are used in this database. The structure of the catalog reflects the metamodel of underlying data model of a DBMS where this database is created.

Among other things, the SQL standard specifies *base tables* that must belong to DEFINITION_SCHEMA (Melton, 2003c) (see section 1.3.3). "The only purpose of the Definition Schema is to provide a data model to support the Information Schema and to assist understanding." (Melton, 2003c)

If we think about the OR_{SQL} metamodel as *data model (meaning 2),* then we could create a set of base tables based on that model and these tables could be part of a database catalog.

We evaluated the OR_{SQL} metamodel by checking, whether the base tables of the "Definition Schema" and their columns, which are specified by the SQL standard (Melton, 2003c) have corresponding base tables and columns in a hypothetical database that we could create based on the OR_{SQL} metamodel. Calero et al. (2006) used the same method in order to evaluate their SQL ontology.

We do not present the metamodel of the entire SQL language in this dissertation. Therefore, we found as we expected that some Definition Schema base tables have no counterpart in our metamodel. These tables are:

- The tables that contain information about authorizations: AUTHORIZATIONS, ROLE_AUTHORIZATION_DESCRIPTIONS
- The tables that contain information about privileges: COLUMN_PRIVILEGES, ROUTINE_PRIVILEGES, TABLE_METHOD_PRIVILEGES, TABLE_PRIVILEGES, USER_PRIVILEGES, USER_DEFINED_TYPE_PRIVILEGES.
- The tables that contain information about the SQL schema objects that we think are orthogonal to underlying data model (character sets, collations, translations): CHARACTER_ENCODING_FORMS, CHARACTER_REPERTOIRS, CHARACTER_SETS, COLLATIONS, COLLATION_CHARCTER_SET_APPLICABILITY, TRANSLATIONS.
- The tables that contain specific information about the SQL standard or a DBMS: SQL_IMPLEMENTATION_INFO, SQL_LANGUAGES, SQL_SIZING, SQL_SIZING_PROFILES.

In addition, the OR_{SQL} metamodel that is presented in this dissertation does not model the concept "module", which is specified in SQL:2003 Part 4 "ISO/IEC 9075-4, Persistent Stored Modules". We think that ability to group the routines is orthogonal to the data model.

If we want to create the metamodel of entire SQL, then we have to extend the OR_{SQL} metamodel in order to cover the previously mentioned constructs as well.

1.5 Summary

The main goal of this chapter is to give a general overview of the OR_{TTM} and OR_{SQL} data models and to compare them. It should make it easier to understand the following chapters.

This chapter contains the following novel results:

- Proposal of the comparison method of data models. One precondition of using this method is the existence of metamodels of the data models.
- Description of the OR_{TTM} and OR_{SQL} data models in the form of metamodels.

These metamodels are presented as UML class diagrams. Metamodel of a data model provides more compact and visual overview about the model components and their associations, compared to purely textual description. The existing literature does not provide clear and complete specification of the OR_{SQL} data model. Instead, there is a large textual specification of SQL database language. There are some other attempts to specify *parts* of the OR_{SQL} data model in the form of metamodel, but none of them is currently as extensive as this work.

- Metamodel-based comparison of the OR_{TTM} and OR_{SQL} data models.

We found mapping between the metaclasses and discrepancies of data models based on the metaclasses. Existing comparison of these data models (Date and Darwen, 2000, Appendix H) does not use this method. They present textual feature-based comparison of principles of The Third Manifesto and SQL. Metamodel-based comparison is similar to the feature-based comparison because it has subjective nature. We had to decide when to create mapping between the metamodel elements based on our understanding of the data models. In our view, the biggest challenge of the metamodel-based comparison method is the creation of the mapping of metamodel elements. One precondition of this work is the existence of clear definitions of the data model constructs. Therefore, the use of such comparison method could trigger the creation and improvement of the definitions.

The advantages of metamodel-based comparison are:

1. Metamodels that are created by using popular modeling language make the comparison more understandable to wider audience.
2. Part of SQL/Foundation document (Melton, 2003) is "SQL feature taxonomy". It could be used in the feature-based comparison. However, the additional advantage of the OR_{SQL} metamodel is that it illustrates interconnections of these "features".
3. In general, a feature could cover more than one data model concept. For example, many prescriptions, proscriptions and suggestions of The Third Manifesto have more than one corresponding metamodel element. Therefore, in case of a metamodel-based comparison, we perform the comparison between more fine-grained elements and the result can be more precise.
4. It is easier to calculate metrics values based on metamodels. These values help to see the relative complexity of the data models.

- Date and Darwen (2000) present examples about how OR_{SQL} violates the orthogonality principle. We found additional examples by studying the metamodel of OR_{SQL}.
- The description of the shortcomings of the ontology that is presented by Calero et al. (2006).

2 DATABASE MANAGEMENT SYSTEMS IN EXISTING SOFTWARE ENGINEERING SYSTEMS

Greenfield et al. (2004) define a formal model as an artifact that captures *metadata* in a form that can be interpreted by humans and processed by tools. Can we use *data* management systems for the artifact management?

This chapter gives an overview of the use of DBMSs in software engineering systems (SES). We want to investigate observations and opinions of researchers about using a RDBMS or an ORDBMS in order to build up a SES. We want to show that the use of these kinds of DBMSs in the engineering systems is already common practice, but there exists the need to improve these DBMSs. Harrison et al. (2000) present summary of history, present situation and future trends of SESs. Next, we list *examples* of SESs.

- CASE environment.
- Meta-CASE environment.
- System that helps to manage and provides access to *repository of reusable artifacts.*

All these systems need to record artifacts and/or data about them somewhere and can take advantage of database technology.

A *CASE environment* allows us to model a system by using a modeling language that is typically a general-purpose language (like UML). Among other things, this system could allow generating new models based on the existing ones, generating code and documentation based on the models and generating models based on the code.

A *Meta-CASE environment* (Zhang and Lyytinen, 2001) permits the specification of new domain-specific modeling languages that use domain specific vocabulary. We can use these languages in order to create artifacts. This kind of system also allows us to specify possible operations with the artifacts.

Many of these systems record their data *directly* in files (we call them *file-based systems*). Some historical reasons of such design decision could be:

- Limitations of DBMSs (see section 2.3). However, DBMSs have improved and evolved over the course of time.
- Expectation that file-based systems have better performance compared to DBMS-based systems.
- DBMS-based system requires the installation of additional software.

File-based systems have their own serious problems. Connolly and Begg (2002, p.12-14) list limitations of applications that access directly files in order to record and retrieve data:

1. Data that is scattered across different files is separated and isolated.

2. Lack of central database could cause the duplication of data (in this case artifacts) in the different computers.
3. Application code depends on the physical structure and storage of data files.
4. It is possible that an application cannot access the data in a file that is created by another application, due to incompatible file formats.
5. New queries based on the data have to be written by an application developer and therefore getting answer to unplanned queries takes quite a lot of time.

There are different views, whether a file-based system has better performance or not compared to a DBMS-based system. Gruhn and Schneider (1998) write: "If process models were stored in files, the access to this information would demand to open many files. This would not be fast enough when dealing with large numbers of related processes." The direct use of files buy a modeling tool also causes problems with the model partitioning and references between models (Greenfield et al., 2004).

If a system (CASE, Meta-CASE or other) uses help of a DBMS and a database that is in the local computer, then it helps to avoid problems 1, 3, 4, 5. Miguel et al. (1990) have come to the similar conclusion. They classify the architecture of CASE environments as tool centric or data/knowledge centric. This classification also applies to any other software engineering environment that is installed in the same computer through which it is used. Data is scattered across different files that are used by different sub-tools of a CASE environment in case of the tool centric architecture. Data that is used by different sub-tools is in one repository database (knowledge base) in case of the data/knowledge centric architecture. Miguel et al. (1990, p. 418) see many advantages of data/knowledge centric architecture, including:

a) "The data base becomes the medium of communication and coordination between tools." (Miguel et al. 1990, p. 418)
b) Each tool uses the set of views that present the required data in the required format.
c) System developer does not have to implement the features that are available in the database system.

An example of CASE system that records data in a database is PARallel Software Engineering CASE system (Gray, 1997). However, even the system that is not an environment, but a single tool, can take advantage of the database systems because of the advantage (c). In addition, in this case it is easier to integrate this tool with an existing system that also uses a database. An example of a tool that could use a database is *pattern-based code generator*. It could be a separate tool or part of a CASE or a Meta-CASE environment or part of a system that deals with the management of patterns. Brash and Stirna (1999) define pattern as "accumulated experience of various business practices that may be useful for tackling similar issues under similar circumstances." Each pattern has a name and emphasizes only one problem in a big problem-space. For example, Florijn et al. (1997) presents a tool that among other things is able to generate program elements based on a selected pattern. Part of the system is database of fragments. A fragment is a design element that has a type (class, method, pattern, association, etc.) and roles that can contain references to other fragments (Florijn et al., 1997).

We note that a CASE or a Meta-CASE environment could be a multi-user environment that records its data directly in a central database. It helps to avoid problems with the duplication of data. For example, web-based system EA WebModeler that records data in a central database provides form-based web interface for creating system specifications. In Chapter 4, we propose web-based modeling system, which records models in a central database.

Another example of the software engineering system is a system that helps to manage the shared database of *reusable* software engineering artifacts. We note that many authors use the concepts "element" or "component" instead of the concept "artifact". This kind of system "provides organization, storage, management, and access facilities for reusable software components." (Constantopoulos et al., 1995) These components could be created during any phase of software development life cycle. An example of this kind of system is Software Information Base (Constantopoulos et al., 1995).

It is possible to use different approaches in order to collect and publish these reusable components. For example, an enterprise that develops systems consists of the development organization and enterprise factory according to Experience Factory (EF) approach (Basili et al., 1994). These parts of an organization have distinct goals. The mission of the development organization is to develop and deliver systems. The mission of the EF is to learn from experience and improve software development practice. The Quality Improvement Paradigm (QIP) specifies six steps that are used in EF in order to plan and execute a project, analyse its results, and package the gained experience for later reuse. Only after that, the experience elements that are recorded in the experience base become available to the public. Q-Labs Experience Management System (Seaman et al., 1999) is an example of the system that implements EF approach (see section 2.2.4.1). It is possible to present experience elements in the form of patterns. Matjás (2006) presents an example of system that makes patterns electronically available (see section 2.2.1). This system deals with the object-oriented design patterns (Gamma et al., 1995). Many printed catalogues of different types of patterns have emerged: object-oriented design patterns (Rising, 2000), (Larman, 2002), data modeling patterns (Hay, 1996), (Silverston, 2001), analysis patterns (Fowler, 1997), project management patterns (Brown, 2000) and modeling guidelines patterns (Evitts, 2000) are some of the examples. These catalogues should be in electronic form together with a search engine, in order to be more useful.

There are also systems that manage data *about* the artifacts. Artifacts themselves are distributed between different computers. An example of such system is Guide to Available Mathematical Software (GAMS) (Boisvert, 1994) that contains data about the software modules. There is also system Agora (Seacord et al., 1998) that is search engine of software components. It searches components that are available in the Internet, collects data about them and compiles an index (see section 2.2.2).

All the previously mentioned systems must contain an information management component. This component must satisfy requirements to a repository system, at least in some extent. Therefore, in the next section (see section 2.1) we describe shortly requirements to the repository systems. A repository system consists of a repository manager (engine) and a repository (database). "A repository is a shared database of information about engineered artifacts produced or used by an enterprise." (Bernstein and Dayal, 1994) A repository manager provides services for modeling, retrieving, and managing objects in a repository and therefore must offer functions of a DBMS and additional functions (Bernstein and Dayal, 1994).

It is possible to build a software engineering system on top of a commercial repository system. Another possibility is to build it on top of a DBMS. In this case, we have to implement the necessary functionalities of a repository system that a DBMS does not automatically provide within the database or in the application code. The amount and simplicity of this additional work depends on the properties of underlying data model of a DBMS.

2.1 Requirements to the Repository Systems

Bernstein and Dayal (1994), Singh and Han (1996), Bernstein (1998) and Blaha et al. (1998) describe necessary functionalities of the repository systems. Emmerich (1995) and Barghouti et al. (1996) present requirements to the information management component of Process-Centered Software Engineering Environments. Tombros and Geppert (1995) present requirements to a DBMS that can be used in order to implement a Process-Centered Software Development Environment.

Some of these requirements are fulfilled by the DBMSs:

1. A DBMS is built up based on a data model that determines the data structures, operators and integrity checking mechanisms that are usable in a repository database.
2. Data Manipulation and Data Definition Languages that conform to this data model.
3. Access control.
4. Transactions.
5. Possibility to distribute data between different servers.
6. Possibility to replicate data.
7. Possibility to backup and restore data.
8. User interface to the database administrator.
9. Programming interface.

Functionalities (3) – (9) are orthogonal to the underlying data model of a DBMS.

The creation of a repository system or an information management component of SES from scratch means the reimplementation of the functionalities of DBMSs. Earlier researches have pointed to the problems of DBMSs that limit their use in the repository systems. For example, DBMSs do not provide many of the value-added services of the repository systems (Sidle, 1980) and do not meet the data storage needs of the repository systems (Miguel et al., 1990). The shortcomings of data models and DBMSs have caused a lot of criticism (see section 2.3).

The database technology has evolved and matured over the years. Section 2.2 refers to many systems that use a general-purpose DBMS in order to manage engineering artifacts and data about them. Dittrich et al. (2000) also note that the use of general-purpose DBMSs in the repository systems gains popularity.

Next, we list the functionalities that are specific to a repository system according to Bernstein and Dayal (1994) and Bernstein (1998):

1. Reuse repository should be adaptable to the needs of a specific organization and reuse project in order to support management of all required artifacts (Feldmann, 1999). Therefore, a designer should be able to *dynamically extend* the schema of the repository database. The extension activities include the definition of new types/operators and modification of the existing ones.
2. The system must be able to react to different events. The events could trigger actions. Examples of the events are that a deadline has passed or all the goals of a milestone have been achieved. The triggered actions could be the generation of code or document or *notification* of users. Jasper (1994) uses the concept "active repository" in order to refer to the repositories that offer this functionality.
3. The system should allow users to acquire and release exclusive or shared rights on artifacts by checking them in and out. Haskin and Lorie (1982) write that this functionality allows users to transport a complex object to their workspace in order to modify it. The system has to lock all parts of this object. The modification could last hours or days rather than minutes. The system should update the object after the end of modification, synchronize the concurrent updates, release the locks and hence make the object fully available to the users of the repository.
4. The system should allow us to manage semantically meaningful snapshots of selected artifacts. These snapshots are called *versions*. The system should be able to restore a particular artifact version in order to present it to the users, create a new version based on it or compare it with other artifacts.
5. The system should allow us to manage bindings (called *configurations*) between a version of a composite object that consists of other objects and a version of each of its (versioned) components.
6. The system should allow us to manage views to the objects in a repository. These views, which are called *contexts,* determine, for example, user preferences, and rules and constraints that are applicable to the objects in them.

7. The system should allow us to track the state of an object in the repository on the basis of a specified *workflow* control model.
8. The system must be able to manage relationships between the objects that are recorded in the repository.
9. The repository system product could provide built-in database schemas that are designed according to some information (data) models (meaning 2). These ready-made schemas should allow us to record different engineering artifacts.

Additional value-adding features that a repository system could provide:

1. The system should allow exporting and importing artifacts from various sources (Blaha et al., 1998).
2. The system should have reverse engineering and code generation capabilities (Blaha et al., 1998).
3. The system should allow us to specify the mapping between the elements of different kind of artifacts that are recorded in the repository (Blaha et al., 1998).
4. The system should allow *users* to specify forms and reports that are used in order to access data in the repository (Blaha et al., 1998). After changing the schema of the repository database, a user interface has to be modified as well. It must be possible to describe mapping between the elements of repository schema and elements of the user interface.
5. A user of a repository system must have possibility to adjust the user interface as well as mapping between models and physical schema (Blaha et al., 1998).
6. The system should allow us to enforce the existing conventions and to define new ones (Blaha et al., 1998). For example, a database designer who works with a physical database design model could enforce the convention for naming integrity constraints.
7. The system should use hypertext (Oinas-Kukkonen and Rossi, 1999). Artifacts could contain hyperlinks to other artifacts and other resources both within as well as outside the repository. These links do not depend on the relationships that are recorded in the repository.
8. Liu et al. (1996) require verification functions that checks correctness of an artifact against some criteria.
9. Liu et al. (1996) require script generation function, the purpose of which is to generate serialized versions of the recorded artifacts.

2.2 Existing Software Engineering Systems

It is possible to classify (software) engineering systems based on their use of DBMSs in order to build up an information management component:

1. Systems that do not use a DBMS at all (see section 2.2.1).
   - Systems that use custom-built information management components.
   - Systems that use a commercial repository system.

2. Heterogeneous systems that use the different means for the data management (see section 2.2.2).
   - Systems that use a DBMS for the data management as well as record data in the files that are not managed by a DBMS.
   - Systems that use more than one DBMS (possibly with the different underlying data models).
3. Systems that use a DBMS for the management of all the data. Barghouti et al. (1996) calls it "closed world view of existing data management systems". A DBMS could be either:
   - An engineering DBMS that is created specifically for the engineering applications. This kind of DBMS can use the same data model as some general-purpose DBMS or it can use specifically designed data model that supposedly makes management of engineering data more easier (see section 2.2.3).
   - A general-purpose DBMS (see section 2.2.4). Batory and Thomas (1997) write: "General-purpose DBMSs are heavyweight; they are feature-laden systems that are designed to support the data management needs of a broad class of applications."

Next, we list different possibilities for recording artifacts in case the system uses the help of a DBMS:

1. An artifact is in the files that are not managed by a DBMS. A database contains references to these files. Bernstein and Dayal (1994) refer to some problems of this approach - data changes in the files cannot be part of database transactions, it is difficult to ensure consistency of a database and the content of files and it is difficult to make queries based on artifacts.
2. An artifact is recorded in a database without decomposing it (see section 3.1.1). For example, an artifact can be recorded as a large object (LOB) in case of using ORDBMS$_{SQL}$ (see section 3.5.1)
3. An artifact is divided into components. These components are recorded in a database (see sections 3.1.2 and 3.1.3).

Next sections contain *examples* of the software engineering systems. We concentrate our attention to the systems that have been described in the scientific papers. We do not claim that these lists are complete. Some of the examples are about the systems that are not used in the software engineering field. It shows that we can use DBMSs in different kinds of engineering systems.

2.2.1 Systems that do not Use a DBMS

This section refers to the systems that do not use a DBMS at all (see Table 7).

Table 7 Examples of software engineering systems that do not use the help of a DBMS

ID) **Name:** Content. Comments.	Comments about data management	Reference
1) **Arcadia**: Software objects that are	Custom-built object	Taylor et

ID) **Name:** Content. Comments.	Comments about data management	Reference
either internal data structures or external products. Examples of internal data structures are parse trees, symbol tables, and abstract syntax graphs. Examples of external products are source code, executable modules, documentation and test plans.	management system.	al. (1988)
2) **Process WEAVER**: Fragments of process description.	Custom-built component that records data in UNIX-files.	Fernström (1993)
3) **Software Information Base (SIB)**: Data about the reusable software components that specify requirements, designs and implementations of software.	Custom-built object management system.	Constantopoulos et al. (1995)
4) Business rules	Commercial repository system Rochade.	Herbst (1996)
5) **Conceptual Schema Reuse (CSR) toolkit**: Reusable conceptual schema components. These components contain schema descriptions as well as semantic descriptors, certification data, reuse history and reuse guidelines. *Comment*: This toolkit provides reuse-oriented services to KHEOPS database design environment.	Custom-built repository system with the selection and insertion tools. Part of the system that deals with the repository management is implemented by using Eclipse Prolog.	Ruggia and Ambrosio (1997)
6) **GraMMi (graphical meta-data-driven modeling tool)**: Conceptual design models of data warehouses.	Commercial repository system Softlab Enabler.	Sapia et al. (2000)
7) Collection of analysis patterns that are usable in ArgoCASEGEO tool. Analysis pattern in this context is any part of a requirement analysis specification that could be used during the GIS application development.	The collection of patterns is recorded in a catalog that is structured through directory architecture. Each pattern is recorded in a separate directory. A directory contains at least a XMI file with a model and a XML file with the data about a pattern.	de Freitas Sodré et al. (2005)
8) Object-oriented software design patterns that were presented by Gamma et al. (1995). *Comment*: The system is a web-based catalogue.	Patterns are recorded as OWL ontology (in XML files).	Matjás (2006)
9) According to OMG Reusable Asset Specification (RAS), a reusable asset that describes a solution to a software development problem should be	The specification suggests that reusable assets should be collected to the central RAS repository. The	OMG ptc/04-06-06

ID) **Name:** Content. Comments.	Comments about data management	Reference
implemented as a package *file*.	specification does not explicitly prohibit or advise to use a DBMS in order to build up a repository. However, it seems that one way is to do it without using the help of a DBMS.	

2.2.2 Heterogeneous Systems

This section refers to the systems that do not record artifacts in a database that is created by using a DBMS. The databases contain only data *about* the artifacts and references to them (see Table 8). The artifacts themselves are in the files that are not managed by the DBMS.

Sign "-" in the columns that contain data about DBMSs (see Table 8-Table 14) means that we do not know the concrete DBMS product that was used in order to build up a particular system.

Table 8 Examples of software engineering systems that do not record all the data in a database

ID) **Name:** Content. Comments.	DBMS type: product	Reference
1) **Project Master Data Base** (also uses the concept "environment database"): Data that is gathered during the entire project lifecycle (persons, tools, products, milestones, requirements, software components, test cases etc.). *Comments*: Artifacts that contain large amount of textual data are recorded in the files that are not managed by a DBMS.	RDBMS: Ingres	Penedo (1987)
2) **The C Information Abstraction System (CIA)**: Source code files of C programs and data about the following global elements of C programs: files, macros, global variables, data types, and functions. *Comments*: The data that is extracted from the source code is recorded in a database. The code is in the files that are not managed by a DBMS.	Any RDBMS is suitable	Chen et al. (1990)
3) **XREF**: Data about programs, including data about files, functions and relationships. *Comments*: Each programming language that is supported by XREF must have corresponding program analyser. The data that is extracted from the source code is recorded in a database. The code is in the files that are not managed by a DBMS.	RDBMS: XREFDB	Lejter et al. (1992)

ID) **Name:** Content. Comments.	DBMS type: product	Reference
4) **Guide to Available Mathematical Software (GAMS) Repository**: Index of the mathematical or statistical software modules or packages that are physically in the different repositories (and computers). Index contains also references to abstracts, documentation, source code, examples and tests that are associated with the modules or packages.	RDBMS: RIM DBMS	Boisvert (1994)
5) **REGINA Software Library project**: Data *about* the software components, including their classification. These components can be at different levels of granularity (classes, class libraries, binary components, but also frameworks) and could be implemented using different programming languages.	RDBMS: Oracle	Behle (1998)
6) **Agora search engine**: Data *about* the software components that are available in the Internet. *Comments*: The system compiles the index automatically by going out over the Internet. Components themselves are in the different computers.	-	Seacord et al. (1998)
7) **Software Engineering Experience Environment (SEEE)**: Experience elements that are captured according to the Enterprise Factory approach (Basili et al., 1994). *Comments*: SEEE consists of an Experience Base (EB) specific part and an artifact specific part. The artifacts could be recorded in a database or in the files that are not managed by a DBMS. The EB specific part of the system is used in order to manage characterizations of these artifacts.	ORDBMS	Althoff et al. (1999)
8) **Online-repository for the Embedded Software (ORES)**: Software components. *Comments*: Code, tutorial, documentation are recorded in files that are not managed by a DBMS	ORDBMS: Oracle8i	Yen et al. (2001)
9) **OSCAR**: Active software artifacts and process data about them. Process data includes data about the actors who change the artifacts, the rationale associated with those changes and tools that were used in order to create artifacts. *Comments*: Process data is recorded in a database and artifacts are in the files that are not managed by a DBMS. Authors think that if they record artifacts outside the database, then it helps to improve performance of the system. However, they acknowledge the loss of the query and transaction services. OSCAR is an open-source system that is part of the distributed software development environment GENESIS.	-	Boldyreff et al. (2002)

ID) **Name:** Content. Comments.	DBMS type: product	Reference
10) **AGAP**: Software patterns and associated diagrams. *Comments*: Software patterns are recorded in a database and their associated diagrams are recorded in a shared directory as XMI files.	-	Conte et al. (2004)

The problem of work (Lejter et al., 1992) is that the authors use the name XREFDB in order to refer to a DBMS as well as to a database. XREFDB DBMS is part of FIELD programming environment and therefore it is fair to say that it is not a general-purpose DBMS, but rather a simple embedded DBMS that accompanies a complex software tool.

Mocko et al. (1994) presents an example of an engineering system (other than software engineering) that uses a RDBMS (MySQL) in order to record data about behavioural models. The behavioural model in this context is "a model that captures the mathematical description of the physical behaviour of a product" (Mocko et al., 2004). Data about the models contains references to the files that contain executable models.

2.2.3 Systems that Use Only an Engineering DBMS

One possibility to classify the research about the Engineering DBMSs is according to the results of the research.

1. A proposal of a data model that could be used by an Engineering DBMS. This data model is supposedly better suited for the management of engineering artifacts than the existing ones. Table 9 contains names and references of some of these models. These models typically use principles of object-orientation together with the ideas from the hierarchical or network data models.
2. A data model together with the implemented DBMS that uses this data model. Examples of engineering DBMSs are PRIMA (Prototype implementation of the MAD model) (Härder et al., 1987), ROSE (The Relational Object System for Engineering) (Hardwick and Spooner, 1989), GRAS (for GRAph Storage)(Kiesel et al., 1995) and Cons-Base (Savnik et al., 1993).
3. A system that uses some engineering DBMS in order to manage engineering artifacts. Table 10 refers to some of these systems.

Table 9 Examples of special data models for the engineering databases

Name	Reference
Hybrid relational and network data model	Haynie (1981)
Molecule-Atom Data Model	Härder et al. (1987)
ROSE (The Relational Object System for Engineering) Data Model	Hardwick and Spooner (1989)
Construction Database Model.	Savnik et al. (1993)
GRAS (Graph Storage) Data Model	Kiesel et al. (1995)
Contiguous Connection Model	Wurden (1997)

Table 10 Examples of software engineering systems that use an Engineering DBMS

ID) **Name:** Content. Comments.	DBMS product	Reference
1) **EPOS**: Process models and process artifacts. *Comment*: EPOS is Process-centered Software Engineering Environment. "EPOS-DB is a proprietary, client-server database to store process models in the context of versioned, nested, long and cooperating transactions. It has a structurally object-oriented data model and its own change-oriented version (COV) model" (Ambriola et al., 1997).	EPOS-DB	Ambriola et al. (1997)
2) **MultiText Analytical Repository System (MARS)**: Computer program source code, the analyses results of this code and supplementary data.	MultiTex	Cox et al. (1999)

There are more examples of engineering systems that use an Engineering DBMSs but these systems are not for the software engineering (see Table 11).

Table 11 Examples of other types of engineering systems that use an Engineering DBMS

ID) **Name:** Content. Comments.	DBMS product	Reference
1) **MARVEL:** Artifacts that are produced by the users of an engineering project. *Comment*: This system is similar to the Meta-CASE systems in the sense that it allows us to adapt the system to the needs of a particular project. "MARVEL is a knowledge-based engineering environment that can be instantiated with the artifacts and tools for a specific engineering project, together with rules regulating the (technical) conduct of the project." (Kaiser et al., 1988)	Custom built database-system	Kaiser et al. (1988)
2) Design objects. *Comment*: The system uses the hybrid approach according to which detailed data about the design objects is recorded in a ROSE database and index of this data is created in a relational database. Design objects can, for example, be design circuits. The index helps to get an overview of dependencies between the objects and the results of changing the objects.	ROSE	Hardwick and Samaras (1989)
3) **Cons-Cad**: CAD models. *Comment*: This system uses Construction Database Model.	Cons-Base	Savnik et al. (1993)
4) **M-Sync**: Mechanical engineering data. *Comment*: The system allows us to synchronize and distribute the data. AMOS II DBMS is a main-memory resident ORDBMS that has the peer-to-peer	AMOS II (Active Mediator Object	Ma et al. (2005)

ID) **Name:** Content. Comments.	DBMS product	Reference
communication capability.	System) DBMS	

2.2.4 Systems that Use Only a General-purpose DBMS

This section refers to the systems that use the help of a general-purpose DBMS and record data about the artifacts as well as the artifacts themselves in a database.

2.2.4.1 Systems that Use a RDBMS

This section refers to the systems that use RDBMSs that conform to SQL:1992 or earlier standards (RDBMS$_{SQL}$s). There are conflicting views whether the relational data model is suitable to use in the engineering applications. Hardwick (1984) concludes based on the literature study that the relational systems are more suited to design applications than the systems that use the network models because (relational) "algebra enables the relational system to calculate relationships dynamically, on demand" and user of relational system does not have to use explicitly pointers. On the other hand, Katz (1990) writes: "Other models, such as the relational model, require rather drastic extensions to form a suitable base for engineering design applications." Dittrich et al. (2000) is also not against the view that "the relational data model is generally not considered powerful enough for the modeling of software repositories". Bernstein et al. (2000) and Dittrich et al. (2000) note that object-oriented DBMSs (OODBMSs) and object-relational DBMSs (ORDBMSs) are suitable platforms on which to implement model management systems and software repositories, respectively. Probably they do not mention relational databases as suitable platform due to criticism towards RDBMS$_{SQL}$s (see section 2.3). Despite that, there are many examples of systems that use a RDBMS$_{SQL}$ (see Table 12).

Table 12 Examples of software engineering systems that use a RDBMS

ID) **Name:** Content. Comments.	RDBMS product	Reference
1) **OMEGA**: Fine-grained data about the procedures, statements, variables etc. that make up a program, which is created by using Pascal-like language called Model. System extracts data from the source code and records it in 58 relations.	INGRES	Linton (1984)
2) **Class library management system for object-oriented programming:** Data about the classes that belong to the class library.	INGRES	Ng et al. (1993)

ID) **Name:** Content. Comments.	RDBMS product	Reference
3) **Integrated toolset for program understanding:** Repository contains data that is extracted from the program code (data items, data types, procedure calls etc.). *Comment*: The repository is used by the tools Rigi and REFINE. Rigi has to discover abstractions from software representations and present them in a meaningful way to software engineers. REFINE helps to cluster program fragments.	SQL/DS	Mylopoulos et al. (1994)
4) **A CASE tool that fulfils the requirements of PARSE (PARallel Software Engineering) project**: Models (process graphs) that are created by using the CASE tool. "Process graphs promote a structured, top-down approach to parallel software development." (Gray, 1997, p. 238)	Oracle	Gray (1997)
5) **JB (Jade Bird) Component Library system – JBCL**: Software components.	Sybase	Keqin et al. (1997)
6) Conceptual and physical data models and mapping between their elements. For example, the mapping between attributes of entity types and columns of tables.	MS-Access	Blaha et al. (1998)
7) **APSARA - A Web-based Tool to Automate Pattern Retrieval and Synthesis**: Reusable patterns that help to automate the design of object-oriented systems. The database contains also class models that are associated with patterns. These models specify classes, their attributes, methods and relationships. *Comments*: The Apsara system creates object-oriented specification (class model) based on the description of requirements in natural language. It searches significant words from the text and then searches patterns based on these words. The system combines different patterns in order to create a final model.	MS-Access	Purao (1998)
8) Repository of **FUNSON net** approach: process models that could be used for the workflow management. Comment: "The FUNSOFT net approach has been implemented in a commercially available workflow management environment, called Leu." (Gruhn and Schneider, 1998)	Oracle	Gruhn and Schneider (1998)

ID) **Name:** Content. Comments.	RDBMS product	Reference
9) **Q-Labs Experience Management System (EMS)**: Software experiences that are captured according to the Enterprise Factory approach (Basili et al., 1994). A perspective is a set of experience packages. "A perspective is defined by three parts: a classification part, a relationship part, and a body part." (Seaman et al., 1999). The perspective body has associated files that are recorded in a database as large objects.	-	Seaman et al. (1999)
10) Models that specify requirements and object-oriented implementation of a large commercial application system - stock broker trading system GEOS. The system also records associations between the requirements and implementation elements. *Comments*: The system allows users to specify requirements and collects data about the implementation by reverse-engineering source code.	SQL-Access	Sneed and Dombovari (1999)
11) **BORE (Building an Organizational Repository of Experiences)**: Cases that describe project-specific solutions to problems that occur during the software development activities. The system also allows us to record associations between the cases, associations between activities, states of activities, options for the activities, questions and answers, references to the documents and domain rules.	-	Henninger (2001)
12) **R^2**: Specifications of requirements. *Comments*: The system allows us to specify requirements as diagrams.	Oracle	Lopez et al. (2002)
13) Aspects that are used in AspectJ program. Aspects are constructs of Aspect Oriented Programming that help to separate crosscutting concerns (Rashid and Loughran, 2003).	-	Rashid and Loughran (2003)
14) Business rules	Access	Chisholm (2003)
15) **UML Model Measurement Tool (UMMT)**: UML class-models and state models. *Comments*: The system reads data from XMI files and populates a database in order to make possible queries based on the models.	MySQL	Lavazza and Agostini (2005)
16) **EA WebModeler**: Commercial system for the management of architecture artifacts.	-	EA Web Modeler (2006)

R^2 and FUNSON use ORDBMS$_{SQL}$ Oracle. We placed descriptions of these systems to the list of systems that use a RDBMS because they do not use object-relational features of Oracle.

System RASES (Relational Algebraic System Entity Structure) (Park et al., 1994) helps to manage models of electronic schemas and simulation. It is an example of an engineering system (other than software engineering) that uses a RDBMS (INFORMIX). In addition, Blanning (1982) and Tsai (2001) suggest that it is possible and reasonable to record models in a relational database. These models are abstract representations of some real-world problems. Examples of such models are transportation or production optimisation problems. Authors do not present working system, but rather investigate possibility of using RDBMSs for recording and managing these models.

2.2.4.2 Systems that Use an OODBMS

Ditrich et al. (2000) present an overview of the systems that use an OODBMS in order to manage engineering artifacts. Table 13 contains *some* examples of this kind of systems.

Table 13 Examples of software engineering systems that use an OODBMS

ID) **Name:** Content. Comments.	OODBMS product	Reference
1) **SPADE-1**: Process models and process artifacts. *Comments:* SPADE-1 is Process-centered Software Engineering Environment.	O_2	Ambriola et al. (1997)
2) Design level specification of enterprise workflow models.	ObjectStore	Liu et al. (1996)
3) **SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) design repository**: Design-level data that is extracted from the source code. "The schema of the design repository is based on an extended version of the UML metamodel 1.1." (Keller et al., 2001)	POET	Keller et al. (2001)

An example of an engineering system that is not software engineering system and uses an OODBMS is NIST Design Repository (Szykman et al., 2000). Its database contains design artifacts and data about them. The system is built by using ObjectStore OODBMS. The paper contains an example of an artifact that is the result of the mechanical engineering process.

2.2.4.3 Systems that Use an ORDBMS

Bernstein (2003) writes about the implementation of a model management system and concludes: "Given technology trends, an object-relational system is likely to be the best choice, but an XML database system might also be suitable." ORDBMS in this case is a DBMS, the underlying data model of which is OR_{SQL}. It allows us to define user-defined types (UDTs) and user-defined routines (UDRs), including user-defined functions (UDFs). Table 14 refers to the systems that use an ORDBMS in order to manage artifacts.

Table 14 Examples of software engineering systems that use an ORDBMS

ID) **Name:** Content. Comments.	ORDBMS product	Reference
1) **Knowledge and Data Base for Software Systems**: program sources, symbol tables, abstract syntax trees. *Comments*: System is able to read C program code and produce symbol table and abstract syntax tree.	POSTGRES	Miguel et al. (1990)
2) CommonKADS models. *Comments*: Database schema is extended version of the generic schema for object modeling called the Defence Command and Army Data Model (DCADM).	Oracle	Allsop et al. (2002)
3) **SFB-501 Reuse Repository**: Software experiences that are analysed and packaged according to the Quality Improvement Paradigm (QIP) steps 5-6 as well as complete experiment documentations, structured in accordance with QIP steps 1-4. *Comments*: This system is designed to support the Enterprise Factory approach (Basili et al., 1994). It uses extreme extending (X2) approach that means that entire application logic as well as major parts of the presentation layer are implemented using the extensibility infrastructure of an ORDBMS.	Informix IDS/UDO	Feldman et al. (2000), Mahnke and Ritter (2002)
4) **SERUM (Generating Software Engineering Repositories using UML)**: Design artifacts. *Comments*: SERUM provides framework for building customized repository managers for the management of different types of artifacts. A repository designer has to create UML specification of a new repository manager by using domain guidelines (specified in OCL), design patterns and templates (Härder et al., 2000). The system records specification and generates code for creating the repository manager and database. The repository database is built up by using typed tables. UDFs implement reading and recording (CRUD) services (Kovse et al., 2002).	Informix IDS/UDO	Härder et al. (2000), Kovse et al. (2002)
5) **UML Repository**: UML models. *Comments*: The schema of its database is created based on the UML metamodel. The repository database is built up by using typed tables. Demuth and Hussman (1999) explain that it is possible to generate relational database constraints from the OCL constraints. This system is an example of that because "OCL invariants defined in the UML meta-model are mapped to SQL constraints" (Ritter and Steiert, 2000).	Informix IDS/UDO	Ritter and Steiert (2000)

ID) **Name:** Content. Comments.	ORDBMS product	Reference
6) **ORIENT (Object-based Relationship Integration ENvironmenT)**. This experimental system extends ORDBMS in order to allow us to preserve the semantics of relationships in a database. The authors use management of software artifacts as an example of a field that can take advantage of such system.	Informix IDS/UDO	Zhang et al. (2001)

An example of an engineering system that is not software engineering system and uses an ORDBMS is DUCADE (Domain-Unified Computer Aided Design Environment) (Montero et al., 2002). Its database contains design features of mechanical and electric engineering domains and couplings between these features. The database is created by using Oracle8i ORDBMS.

2.3 Problems of Using the Relational Model and RDBMS$_{SQL}$s in Engineering Systems

Next, we classify problems of the relational model and RDBMS$_{SQL}$s based on the literature study and briefly analyse them in terms of The Third Manifesto. The study covers many papers that are referenced in the previous section (see section 2.2). They refer to the problems of the relational data model and RDBMS$_{SQL}$s. We remind, that the relational model in this case is the underlying data model of SQL:1992 or previous standards. Researchers and developers present these problems as reasons why a relational database is not the best type of database that can be used in the (software) engineering systems. On the other hand, this dissertation tries to show that the relational model (OR$_{TTM}$ model) is suitable for the engineering systems. Therefore, we have to be familiar with the criticism towards the relational model and systems that implement it.

Often the researches point only to some problems that are the most important in their opinion. This study is different because it presents references to many problems. Table 15 presents problems of the relational model that are mentioned in the literature.

Some researchers have raised the issues that are actually orthogonal to the relational model. We adopt the approach taken by Date and Darwen (2000, p. 21): "The question as to what data types are supported is orthogonal to the question of support for the relational model." Transaction model is also in our view orthogonal to the relational model. Date and Darwen (2000) have requirement for nested transactions in the section of the Other *Orthogonal* Prescriptions.

Hierarchic and networked data can be represented relationally (Pascal, 2000, chap. 7). The issue of making queries based on data that represents graph structure is addressed in The Third Manifesto. The Relational Model Very Strong Suggestion no. 6 (Date and Darwen, 2006) requires that a relational language should provide shorthand for expressing generalized transitive closure operation. The paper of Agrawal and Jagadish (1987) is an example of the work that presents and compares algorithms for computing the transitive closure of large database relations. They emphasize the importance of transitive closure as a primitive database operator. The latest versions of the SQL standard, which specify "WITH RECURSIVE" phrase (Melton, 2003), also allow us to create recursive queries.

Fragmentation increases complexity to the user of a database according to Gray (1997). Virtual relvars (views) help to overcome this problem in ORDBMS$_{TTM}$s. A view expression can join values of relvars that contain data about an object. The view can have relation-valued attributes, the values of which are calculated using the relational operator GROUP that provides relation "nest" capability (Date and Darwen, 2000).

Table 15 Problems of the relational model according to literature

Problem	Authors who mention that problem
It is not powerful, flexible and expressive enough.	Hardwick and Spooner (1989), Constantopoulos et al. (1995), Ma et al. (2005)
Fragmentation. Data about the object is in different relations (tables).	Kemper et al. (1987), Liu et al. (1996), Gray (1997)
Performance problems due to fragmentation.	Kemper et al. (1987), Hardwick and Spooner (1989), Gray (1997)
Super/sub typing is not supported	Liu et al. (1996)
Lack of powerful type system that could allow "complex" types. For example, Hardwick (1984) thinks that the relational model is invented for flat, homogeneous entities.	Hardwick (1984), Taylor et al. (1988), Emmerich et al. (1992), Liu et al. (1996), Gray (1997)
An inability to represent heterogeneous relationships.	Hardwick (1984)
Poor support to data that represents graph structures (including hierarchies). Lack of facilities for making queries based on such data including finding transitive closure.	Hardwick (1984), Katz (1990), Emmerich et al. (1992), Gray (1997), Lange et al. (2001), Yen et al. (2001)
Inappropriate transaction models for the engineering systems.	Hardwick and Spooner (1989)
Detailed semantics of the relvars have to be captured outside the relational database.	Wurden (1997), Engle (2003)
Lack of possibility to preserve the semantics of relationships.	Zhang et al. (2001)

Gray (1997) writes that fragmentation may cause performance problems. However Stonebraker et al. (1991) and Date (2005) think that performance is not a data model issue but an implementation issue. In addition, some researches about using relational databases in order to record engineering artifacts do not see performance as a problem. Allsop et al. (2002) write: "We are confident that there is no performance problem in extracting data from the database using complex PL/SQL queries."

The internal predicate a relvar is the conjunction of all the constraints that apply to this relvar (Date, 2003). The DBMS has to understand and enforce this predicate. "Internal predicates are (loosely) what the data means to the system" (Date, 2003, p. 262). A RDBMS should provide means for defining constraints.

Database users have to understand internal predicates. However, it is possible to construct an informal description of the relvar that helps to explain the meaning of the relvar to the human user. Date (2003) calls this description "external predicate". For example, section 3.2.4.1 contains some examples of external predicates. External predicates could well be recorded in the database catalog (see section 1.3.3).

Table 16 presents problems of RDBMSs that are mentioned in the literature.

Table 16 Problems of the RDBMSs according to literature

Problem	Authors who mention that problem
Limited amount of data types (it is not possible to record "complex objects" without fragmentation).	Haskin and Lorie (1982), Miguel et al. (1990), Barghouti et al. (1996)
It is not possible to define functional interface of a data type wholly within the schema using SQL In other words, it is not possible to specify operators/function that allow us to perform operations with the data values.	Barghouti et al. (1996)
Fragmentation. Data about the object is in the different relations (tables).	Barghouti et al. (1996)
Views (including updateable) are inadequately supported.	Haynie (1981), Emmerich et al. (1992), Emmerich (1995), Barghouti et al. (1996)
Database language (SQL) does not have the power to express transitive closure and path traversal queries	Miguel et al. (1990), Consens et al. (1992)
Performance is not satisfactory	Haskin and Lorie (1982), Linton (1984), Miguel et al. (1990), Chen et al. (1990), Barghouti et al. (1996), Lange et al. (2001)

Problem	Authors who mention that problem
Inappropriate transaction models.	Haskin and Lorie (1982), Katz (1990), Emmerich et al. (1992), Barghouti et al. (1996), Gray (1997)
Inadequate concurrency control mechanisms (like two-phase locking).	Constantopoulos et al. (1995)
Lack of versioning facilities.	Emmerich et al. (1992), Emmerich (1995), Barghouti et al. (1996), Gray (1997)
Lack of facilities for maintaining consistency between data structure definitions in the schema and operation definitions in a host programming language with embedded SQL.	Emmerich (1995), Barghouti et al. (1996)
Lack of access control on a level of single tuples in a relation	Emmerich et al. (1992)
Lack of distributed and multi-database architectures	Gray (1997)
Lack of configuration management	Constantopoulos et al. (1995), Gray (1997)
Lack of possibilities to have cooperative work processes	Gray (1997)
It is difficult to integrate an existing tool with RDBMS if the source code of the tool is not available.	Barghouti et al. (1996)

Some of these problems are also present in the list of problems of the relational model (see Table 15). The first five problems in Table 16 (problems with the data types, views and transitive closure queries) are caused by the inadequate implementation of the relational model by RDBMS$_{SQL}$s. The fragmentation problem is caused by the limited support to viewed tables by RDBMS$_{SQL}$s. All other problems are *orthogonal* to the relational *model*.

The problems that are mentioned in Table 15 and Table 16 should primarily cause improvement of the implementation and standards but not necessarily the invention of new data models.

Barghouti et al. (1996) evaluate RDBMSs in order to find the shortcomings that limit their use in software engineering systems. One difference from the present research is that they do not present separately shortcomings of the relational data model and implementation of the model (DBMSs). They also do not point to all the shortcomings that are mentioned in the literature.

During the literature study we discovered that the existing researches do not always make clear whether they describe the problems of the relational *model* or *implementation* of this model by some standard and DBMS. For example, Hardwick and Spooner (1989), Emmerich et al. (1992) and Singh and Han (1996) point to the shortcomings of "relational *technology*".

2.4 Summary

This chapter gives a short literature-based overview of existing software engineering systems and requirements to their information management components. We are most interested in systems where the information management component uses a RDBMS or an ORDBMS. There exist overviews of software engineering systems that use some other type of DBMSs. For example, Tombros and Geppert (1995) present a list of software development environments that use an OODBMS and Dittrich et al. (2000) present a list of special-purpose software engineering database and object management systems. However, they refer to only few software engineering environments that use a RDBMS or an ORDBM. Guo and Luqi (2000) present a survey of software reuse repositories that are one type of software engineering systems. However, the comparison part of their survey does not contain information whether these systems are built on top of a DBMS or not.

This chapter contains a more thorough list of software engineering systems, that use a RDBMS or an ORDBMS, than the existing overview papers. It illustrates the fact that quite a lot of researchers and developers have decided to use general purpose DBMSs in order to build up a software engineering system. Dittrich et al. (2000) thinks that the object-oriented data model is also one of the most prominent general-purpose data models. In line with this view, we consider OODBMSs as "general-purpose" systems in this work. However, the creation of OODBMSs was partly triggered by the needs of specific types of applications like CASE and CAD and they are not as widely used as RDBMSs or ORDBMSs. Bernstein (1998) writes: "Indeed, many object-oriented database systems have been marketed primarily as support for software tools."

The systems in this chapter help to manage analysis specifications, design specifications, program code, experience elements and patterns. "Appendix A: Some properties of existing software engineering systems that use the help of a DBMS" is a table that gives an overview of the content of their repositories. It shows that most of the first systems that came into existence helped to manage program code. They were followed by systems that helped to manage other software engineering artifacts. Harrison et al. (2000) note the same thing by writing: "The first significant efforts in producing tightly integrated development environments were those in the area of programming support environments (PSEs)."

All the systems in this chapter that use a RDBMS or an ORDBMS use actually a $\text{RDBMS}_{\text{SQL}}$ or an $\text{ORDBMS}_{\text{SQL}}$, respectively. We did not find any software engineering system that uses an $\text{ORDBMS}_{\text{TTM}}$. The reason is probably the lack of stable and easily usable $\text{ORDBMS}_{\text{TTM}}$s. Currently *Alphora Dataphor*, which is a federated DBMS with integrated application development environment, is the only commercial implementation of the principles of The Third Manifesto. An example of a prototypical system is the free and open source DBMS *Rel* (Voorish, 2005).

The referenced research literature points to numerous problems with the relational data model and RDBMSs. Despite that, we found many examples of software engineering systems that use a RDBMS. Papers about these systems are a good source of comments about shortcomings of $\text{RDBMS}_{\text{SQL}}$s and their underlying data model. The problem is that different papers refer to different problems and there is no comprehensive list of all possible problems. The novel results of this chapter are lists of the SQL and $\text{RDBMS}_{\text{SQL}}$ problems together with the references to the papers that mention these problems. We also shortly analysed these problems in terms of The Third Manifesto.

We found more systems that use a $\text{RDBMS}_{\text{SQL}}$ than systems that use an $\text{ORDBMS}_{\text{SQL}}$. This can be explained by the fact that $\text{ORDBMS}_{\text{SQL}}$s came into existence much more lately. Papers that describe systems, which use an $\text{ORDBMS}_{\text{SQL}}$, point to several advantages of these systems:

- "Advanced data, object, and knowledge (rules) services" (Miguel et al., 1990)
- " Access to External Data" (Ritter et al., 1999)
- "Infrastructure for Access via WWW" (Ritter et al., 1999)
- "The enhanced type system" (Ritter and Steiert, 2000)
- "The powerful SQL facilities" (Ritter and Steiert, 2000)
- "Extensibility features of ORDBMSs" (Ritter and Steiert, 2000)
- "Allow the mapping of important concepts of object models, such as class hierarchy, to the repository schema" (Kovse et al., 2002)
- Row type can be used in order to "store values of table relationships so that the join operations can be reduced or eliminated" (Pardede et al., 2003)

We agree that these features give more options to database designers and programmers compared to $\text{RDBMS}_{\text{SQL}}$s. However, these papers pay little attention to the possible problems of $\text{ORDBMS}_{\text{SQL}}$s. For example, Mahnke and Ritter (2002) acknowledge in their final sentence that the use of extreme extending (X^2) approach by using the extensibility infrastructure of the ORDBMS will definitely lead to "a whole bunch of problems, e. g., concerning system performance and robustness as well as ease of development." It seems that this comment is about X^2 approach, but not about the ORDBMSs or their underlying data model. Major parts of the presentation layer (GUI) reside within the database server in accordance with the X^2 approach. Section 3.3.2.1 refers to some problems that the researchers have discovered when they tried to implement whole-part relationships in an $\text{ORDBMS}_{\text{SQL}}$ database.

We think that the investigation of possible problems of OR_{SQL} and $\text{ORDBMS}_{\text{SQL}}$s is also important. The results may help to decide whether to use an $\text{ORDBMS}_{\text{SQL}}$ in a software engineering system. We also think that it is worth to investigate, whether the use of an $\text{ORDBMS}_{\text{TTM}}$ in the engineering systems is more advantageous compared to $\text{ORDBMS}_{\text{SQL}}$s. For example, one reason to prefer an $\text{ORDBMS}_{\text{TTM}}$ is the lack of orthogonality in OR_{SQL} (see section 1.3.8).

3 REPOSITORY DATABASE DESIGN

Bernstein et al. (2000) suggest that a model management system can take advantage of a specialized DBMS, the underlying data model of which is domain specific and treats models, model mappings and model management operations as its first-class elements.

This dissertation, on the other hand, investigates how it is possible to use the features of some of the "general purpose" data models - OR_{TTM} and OR_{SQL} - in order to record software engineering data.

In sections 3.1-3.4 we describe the designs that are usable in an OR_{TTM} database. We use the concepts of the OR_{TTM} data model, if not stated otherwise. We present examples of database language statements. They have been written in *Tutorial D* relational language and have been mostly tested in the prototypical $ORDBMS_{TTM}$ Rel (ver. 0.0.13 Alpha) (Voorish, 2005). We have not tested the statements that use (a) TCLOSE operator; (b) outer joins; (c) user-defined types; (d) THE_ operators. Unfortunately, Rel does not support them yet or supports partially. Tutorial D language has been proposed in The Third Manifesto (Date and Darwen, 2000) and a dialect used by Rel is based on that proposal.

3.1 Design Alternatives of Database Schema of a Software Engineering System

We investigate only the design alternatives according to which an artifact is recorded entirely in a database. These alternatives allow us to use all the features of a DBMS and its underlying data model for the artifact management.

Artifacts are created by using some language. A language allows us to create one or more types of artifacts. An example of a modeling language is UML (OMG formal/03-03-01) and an example of a pattern writing language is Pattern and Component Markup Language (ObjectVenture).

How can we specify a language? The description of a semi-formal language (like, for example, UML) contains descriptions of abstract syntax, well-formedness rules and semantics. The abstract syntax is specified using a metamodel. The well-formedness rules are expressed using OCL constraints (OMG formal/2006-05-01). The semantics is described using free-form text.

A repository system permits management of artifacts that are created using a language that belongs to the set of its supported languages. The repository system should allow us to add new languages to this set in order to be most useful. Each repository has an *information model* that "specifies a model of the structure and semantics of the artifacts that are stored in the repository." (Bernstein, 1998) This information model contains a general part and a metamodel specific part. We have to create the latter part of an information model based on the metamodels of the languages and their well-formedness rules. We can implement a repository as an ORDBMS$_{TTM}$ database by creating a set of types, relvars, operators and integrity constraints. There are different approaches to build up a repository and we explain them in the following sections.

Each metamodel is an instance of a meta-metamodel. If we implement repository by using a DBMS, then components of the underlying data model of this DBMS correspond to the meta-metamodel.

3.1.1 Encapsulated Artifact Types

We implement each artifact type by using:

1. Scalar type ST that corresponds to the artifact type. Values that belong to this type are artifacts. This type could be a user-defined type. However, if we use the OR$_{TTM}$ data model as underlying data model of an Engineering DBMS, then this type and its associated operators could also be built-in in this EDBMS.
2. Exactly one real relvar RR with the type RELATION {K, A}. An artifact is recorded as a tuple that is part of the value of this relvar. K and A are pairs of attribute name and type name. A represents the attribute that corresponds to the artifact type. This attribute has the scalar type ST.
3. Scalar type that is specified in K. It could be the built-in scalar type INTEGER or a user-defined scalar type.
4. Possible representations of scalar types (in this case from pairs K and A) have components. We need a set of operators that allow us to select and modify values of these components (see section 1.3.2).
5. Constraints that the attribute of the relvar RR that corresponds to K is a candidate key and a foreign key that refers to relvar *Artifact*.
6. It is not necessary to record the same artifact more than once. Therefore, we also need a constraint that the attribute of relvar RR that corresponds to A is a candidate key.
7. Exactly one virtual relvar VR that joins values of relvars *Artifact* and RR. If we assign a new value to VR, then the system should assign a new value to relvars *Artifact* and RR.
8. Candidate key attribute of relvar *Artifact* could be a surrogate key, which means that the values of this attribute are generated by the system by using system function SERIAL (see section 1.3.3.1).

We need relvar *Artifact* because if we want to record additional metadata about the artifacts in general (for example, events with them), then we have to create additional relvars and associate them with relvar *Artifact*. A possible naming convention could be that the names of RR and VR are almost the same except that the name of RR has prefix "_", but the name of VR does not.

Figure 45 presents fragment of the information model of a repository that allows us to record Use Case Diagrams (UCD) and State Transition Diagrams (STD). In case of entity type *UCD* in Figure 45 we have to create real relvar *_UCD* with the type RELATION {artifact_id# INTEGER, model UCDType}.

In addition, we have to create virtual relvar *UCD* with the type RELATION {artifact_id# INTEGER, model UCDType}.

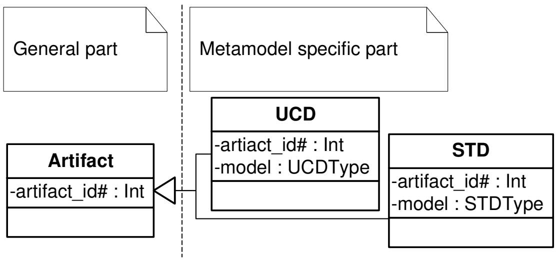


Figure 45 Example of structure of a repository that uses encapsulated artifact types

Next, we present examples of statements for creating relvars. For example, we create the real relvar that corresponds to entity type *UCD* (1).

VAR _UCD BASE RELATION {artifact_id# INTEGER, model UCDType} KEY {artifact_id#} KEY {model} FOREIGN KEY {artifact_id#} REFERENCES Artifact; (1)

In addition, we create the following virtual relvar (view) (2) because there is a generalization relationship (see section 3.3.1). Supertype *Artifact* represents a more general concept and subclass *UCD* more specialized one.

VAR UCD VIEW (_UCD JOIN Artifact) {artifact_id#, model}; (2)

If a user assigns a new value to virtual relvar *UCD*, then the system assigns new values to real relvars *_UCD* and *Artifact*. The precondition of this design is that a DBMS must allow us to update the value of a virtual relvar so that *all* its underlying real relvars get a new value.

We can call this design "encapsulated artifact type" because each artifact type has a corresponding scalar type. Well-formedness rules of the artifacts have to be implemented as type constraints. We need a set of scalar- and relational read-only operators as well as update operators in order to perform operations with the values of these types (artifacts). For example, a possible representation of type *UCDType* could contain components *Use_Case*, *Actor*, *Include* that all have a relation type. We need read-only and update operators in order to expose these components (see section 1.3.2). If we want to make it easier to make queries, then we have to create virtual relvars, which present data in unencapsulated way. For example, if we assume that the relation type of component *Use_Case* has only one attribute – *name* (with the type CHAR), then the type of the following virtual relvar (3) is RELATION {artifact_id# INTEGER, name CHAR}. Its value contains names of use cases.

Operator *THE_Use_Case* exposes component *Use_Case* of *UCDType*. Operator UNGROUP is used in order to "unnest" an attribute that has a relation type. Why do we have to use this design if we need so complex virtual relvars?

```
VAR Use_case VIEW ((EXTEND _UCD ADD THE_Use_Case          (3)
 (model) AS Use_Case) {artifact_id#, Use_Case}) UNGROUP
                    Use_Case
```

In general, types should correspond to properties and relvars to entities (Date and Darwen, 2000, Appendix C). Singh and Han (1996) have the same position. They note that the coarse grained representation of documents in an object-oriented repository make it difficult and inefficient to manipulate individual document components. For example, the little modification of the document would mean recording entire document in order to preserve old version of it. An alternative is to use the design where an artifact is recorded by using many real relvars. The design "encapsulated artifact type" does not follow the suggestion of Date and Darwen (2000).

3.1.2 Non-encapsulated Artifact Element Types

An artifact consists of artifact elements. Each element has a type. We implement each artifact element by using:

1. Exactly one real relvar RR with the type RELATION {K, P_1,...,P_n}. K, P_1, ..., P_n are pairs of attribute name and type name. Each attribute in the pairs P_1,...,P_n corresponds to one property of the artifact element type. K represents the surrogate key attribute.
2. Scalar types that are used in the pairs K, P_1,..., P_n. These types are either built-in or user-defined.
3. Set of operators that allow selecting and modifying components of the possible representation of these scalar types.
4. Let us assume that entity types ET_1, ..., ET_n in the information model are organized into a class hierarchy. Let us assume that ET_k is the supertype and ET_{k+1} is its direct subtype ($1 \leq k < n$). Real relvar RR_k that corresponds to ET_k and real relvar RR_{k+1} that corresponds to ET_{k+1} are associated using a foreign key. For each ET_i where $i>1$ we have to create a corresponding virtual relvar. For example, element type ET_{k+1} has corresponding virtual relvar VR_{k+1} that joins values of real relvars RR_1,..., RR_{k+1}. If one assigns a new value to VR_{k+1}, then the system should be able to assign a new value to all the real relvars RR_1,...., RR_{k+1}. Again, we could use naming convention that names of RR_i and VR_i are almost the same except that the name of RR_i has prefix "_", but the name of VR_i does not have this prefix.

In this case, an artifact element is recorded as a tuple in a relation and artifact (as a whole) is represented by the set of tuples in the different relations. If a new artifact is added to a repository, then it must be broken into elements so that it is possible to record these elements.

We illustrate a database structure that follows this design by using two software design languages:

- Simple software design language SimpleM that was originally presented by Serrano (1999) in order to introduce VCt specification language.

SimpleM specifies one diagram (visual model) type. We can use it in order to create simple state diagrams.

- Unified Modeling Language (UML). We consider its part that specifies use-case diagrams (OMG formal/05-07-04, p. 570).

Figure 46 presents a fragment of the information model of a repository that allows us to record the state models, which are created by using SimpleM as well as use case models, which are created by using UML. We have simplified UML language specific part of the information model for the presentation purposes by adding attribute *name* to entity types *Classifier*, *Use Case* and *Include*. In reality, these entity types have this attribute through inheritance. We also do not consider extension relationships and extension points in this example.

Artifact, *Element_in_artifact* and *Element* are not part of UML or SimpleM metamodel and correspond to the generic part of the repository.

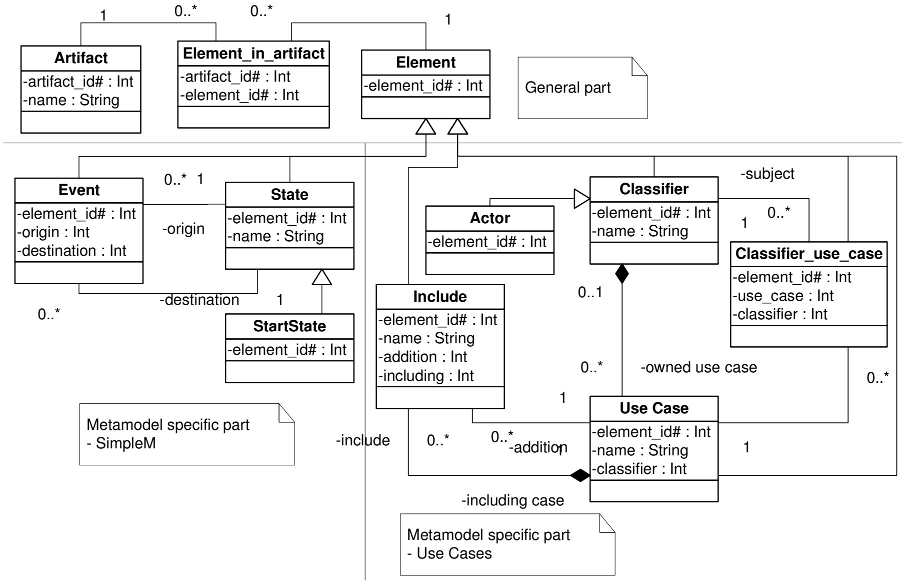


Figure 46 Example of structure of a repository that uses non-encapsulated artifact element types

For example, based on entity type *Actor* we have to create real relvar *_Actor* with the type RELATION {element_id# INT} and virtual relvar *Actor* with the type RELATION {element_id# INT, name CHAR}. Relational expression of this virtual relvar joins relations *_Actor*, *_Classifier* and *Element.*

3.1.3 Encapsulated Artifact Element Types

It is also possible to use the design that combines previous two designs (see sections 3.1.1 and 3.1.2). In this case, each artifact element type ET has corresponding scalar type ST and real relvar RR with an attribute that has type ST. In addition, relvars should contain candidate key attributes, the values of which are generated by the system and foreign key attributes. An artifact is recorded as a set of tuples that are part of the values of more than one relvar. This design uses user-defined scalar data types as the "Encapsulated Artifact Types" approach. This design is also similar to the "Non-encapsulated Artifact Element Types" approach because an artifact will be recorded as a value of more than one relvar.

This kind of design does not follow the suggestion of Date and Darwen (2000, Appendix C): "types should correspond to properties and relvars to entities." The design that is presented in this section does not remove complexity from the repository design. On the contrary, we need type constraints as well as database constraints in order to enforce well-formedness rules. We also need virtual relvars that expose components of the scalar types that correspond to the element types.

3.1.4 Universal Data Model

Next, we investigate suitability of using the database design "Universal Data Model" in order to build up a repository database. In this case, the concept "data model" has the meaning 2 (see Introduction). We use the concepts of the OR_{SQL} data model in this section because existing literature about this design uses these concepts.

Nowadays the results of the research of Baskerville and Pries-Heje (2001) should not be surprise to anyone. Their research shows that two important properties of the system development methodologies of Internet time are: (a) constant time pressure to the developers and (b) vague requirements that often change. Designers of a repository must also take into account these factors because requirements to a repository database schema evolve. One tempting solution seems to be the use of a highly generic database design that has different names: "Universal Data Model" (Hay, 1996, p. 254-256), "The entity-attribute-value representation with classes and relationships (EAV/CR)" (Chen et al. 2000), "Generic data model" (Kyte, 2003, p. 34-36). The following diagram (see Figure 47) presents the general idea of this design. We note that diagrams in this section present conceptual data models and therefore they do not contain foreign key attributes.

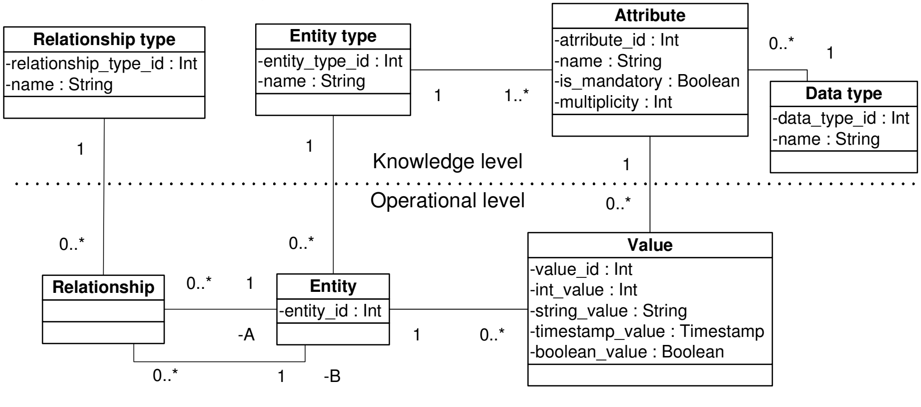


Figure 47 Conceptual data model of "Universal data model"

This diagram uses the modeling principle according to which a model should be explicitly divided into operational and knowledge levels (Fowler, 1997, p. 26). "The knowledge level objects define legal configuration of operational level objects" (Fowler, 1997, p. 25) Data about an object system is recorded at the operational level in terms of the entities, their attributes and relationships. Entity type *Value* has a set of attributes with the general form: <<*data_type_name*>>*_value data_type_name*. These attributes allow us to record values that have different types. Amount of these attributes and their data types depend on a DBMS where this database is created. Data at the knowledge level determines the legal values that can be associated with an entity at the operational level. The knowledge level contains data about relationship types, entity types and their attributes, and data types of the attributes. Some of the variations of the "Universal Data Model" are:

- The word "entity" can be replaced with the words "object" or "thing".
- Hay (1996, p. 254-256) proposes entity type *Attribute_assignment* that models an association between *Attribute* and *Entity type* (conceptually many-to-many relationship between *Attribute* and *Entity_type*).
- Entity types *Attribute* or *Attribute_assignment* could have associated entity type *Legal_value* (or *Domain_element*) that allows us to specify legal values of the attributes (Hay, 1996, p. 255).
- Entity type *Attribute* could have an attribute or even associated entity type *Format* in order to permit recording of a format for the values of an attribute (Hay, 1996, p. 255).
- Each supported data type should have corresponding table for recording values with this type (see Figure 48) according to EAV/CR approach (Chen et al., 2000). This is different from the solution in Figure 47, where is one generic entity type (and therefore also a table) *Value*.
- Another possibility to extend the design is to allow us to record permissible relationship types between the entity types at the knowledge level. This data determines permitted relationships between entities at the operational level.

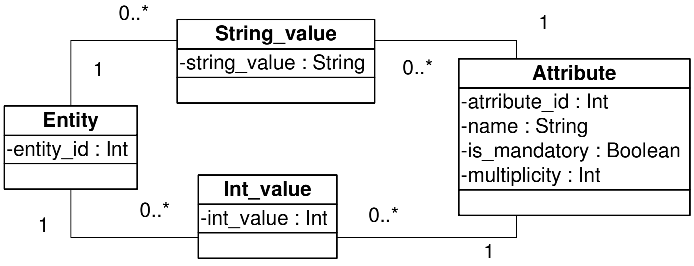


Figure 48 Fragment of EAV/CR design

We use the concept "universal design" in order to refer to the database design according to the "Universal Data Model" (see Figure 47). We assume that each entity type in the conceptual data model (see Figure 47) has a corresponding table and each attribute has a corresponding column. The names of the tables and columns are the same as the names of the entity types and attributes, respectively.

We use the concept "regular design" in order to refer to the design where each entity type and attribute in a conceptual data model (meaning 2) has a corresponding table and column in a database schema, respectively.

At first glance, the "universal design" seems like the easy way to quick success. However, it also has many serious problems. For example, Kyte (2003, p. 34-36) points to the problems with the query complexity and query speed. Do and Rahm (2004) also acknowledge complexity of the queries. These are not the only problems. It seems that there is a lack of consensus about this design and no comprehensive discussion about all its shortcomings. Researchers and developers have tried to use it repeatedly in order to achieve maximum flexibility.

Systems that use "universal design" have to manage large amounts of data. Some bioinformatics systems use a database that is designed according to EAV/CR approach: (a) Subsystem of system net-TRIAL that helps to manage procedures and laboratory results of the clinical trials (Hageman and Reeves, 2001); (b) SenseLab database for recording neuroscience data (Marenco et al., 2003); (c) System PhD for web-based management of phenotype data (Li et al., 2005). System GenMapper that helps to integrate heterogeneous molecular-biological annotation data (Do and Rahm, 2004) uses database that is designed according to Generic Annotation Model that is a variation of the "Universal Data Model". It contains a source level (knowledge level) and an object level (operational level). Some systems in the area of software engineering also use "universal design". Bernstein et al. (1997) describe the Microsoft repository that uses a $\text{RDBMS}_{\text{SQL}}$ database in order to provide persistent storage for the different software tools. This database contains generic tables *Object* and *Relationship* among others. These tables correspond to entity types *Entity* and *Relationship* in the "Universal Data Model", respectively. Habela (2002) proposes flattened metamodel that resembles the "Universal Data Model" (see section 1.4.1). Habela (2002) envisage that the schema of a metadata database could be designed based on this metamodel. Bednárek et al. (2005) describe the data integration system DataPile that records data in a repository, which is designed according to the "universal design".

The advantages of the "universal design" are:

1. It is possible to extend the repository without executing DDL (Data Definition Language) statements. Instead, a user has to modify data at the knowledge level and the system has to execute DML (Data Manipulation Language) statements. Ideally, even the users who are not database designers or programmers could do that. Question remains – why it is the better approach compared to the *generation* and *execution* of DDL statements based on the instructions of a user?
2. These changes do not require corresponding changes in the user interface that is provided to database users, if there is one to one mapping between the columns in the tables and fields in the forms.

3. If a value of an attribute is missing, then in case of EAV/CR approach there is no need to use NULL's (Ahnøj, 2003).
4. A query for finding all the data about an entity has to access only one table (*Value*) and does not need reprogramming then attributes of an entity type change (Ahnøj, 2003). If an entity has attributes with the different types, then more than one table has to be accessed in case of EAV/CR approach.

On the other hand, there are many problems with the "universal design" in the following areas:

1. Database schema evolution
2. Expressiveness of a database schema
3. Constraints
4. Compensating actions
5. Default values
6. Query complexity
7. Missing information
8. Dependencies of database objects
9. Query performance
10. Size of data
11. Access control
12. Concurrency control
13. User interface of a data management program

***Database schema evolution*:** A database that is designed according to this design may still need schema changes in the future because of the data types that are usable in a DBMS. Each data type could have a corresponding column in table *Value* or even a separate table in case of EAV/CR approach. The set of predefined data types in a DBMS may change from release to release. Some of these changes are caused by the changes in standards. For example, SQL:1999 introduced the predefined type BOOLEAN (Gulutzan and Pelzer, 1999). SQL:2003 deprecated the data types BIT and BIT VARYING (Melton, 2003, p. 1173). ORDBMSs provide data type constructors in addition to the predefined data types. Therefore, a large amount of data types could be used in a database. If new requirements stress the need for having an attribute with a data type that has no corresponding column in table *Value* or no corresponding separate table, then the database structure has to be changed. It seems reasonable to use most popular predefined data types at the beginning and gradually add support to the data types. The result of the application of this kind of design could be the use of the limited amount of simple data types (for example, VARCHAR and INTEGER) as column types. This, on the other hand, limits and complicates operations with the data values. An application that uses this database must perform type conversions.

All the data values that otherwise would be part of different tables are now in table *Value* (or in the separate tables that correspond to the data types in case of EAV/CR approach). A comment about the implementation – a DBMS usually locks a table exclusively in case of changing its structure. If someone changes structure of table *Value,* then it locks a very large portion of a database.

Therefore, all the schema changes have to be done at the times, when the use of the *entire* database is as minimal as possible. Corruption of a database table or its indexes or modifications of its data or structure have far greater consequences compared to the "regular design".

Expressiveness of a database schema**:** External predicate of a relvar is an informal construct that specifies what the data in a database means to the user (Date, 2003, p. 263). In this case, external predicates of tables do not give any information about the object system, the data of which is recorded in a database. For example, table *Value* could have the following external predicate (4). The parameters of the predicate correspond to the columns and they are written in *italics*.

Entitity *entity_id* has an associated value *value_id* of an attribute *attribute_id,* which is either an integer value *int_value*, string value *string_value*, timestamp value *timestamp_value* or Boolean value *boolean_value*. (4)

We need a special tool in order to present database conceptual schema based on the data at the knowledge level (Marenco et al., 2003).

Constraints**:** It is more difficult to enforce constraints to the data values than in case of the "regular design". For example, the data at the knowledge level could state that entity type ET has mandatory attribute A with the multiplicity "1". Attribute A has data type DT. Therefore, each entity E, that has type ET, must have exactly one associated value V that is associated with attribute A. Which attribute (*int_value*, *string_value*, *timestamp_value*, *boolean_value* etc.) of V has a value, depends on the data type that is associated with A.

The SQL standard permits creation of assertions and the use of subqueries in the CHECK constraints. However, some well-known DBMSs (like PostgreSQL (PostgreSQL, 2005) and Oracle (Oracle, 2005)) do not follow the standard in this regard. Türker and Gertz (2001) note in the review of integrity constraints in the different DBMS-s: "assertions are in general not available and are unlikely to be offered in the near future". Therefore, triggers have to be created in order to enforce these rules at the database level. These triggers must react to the following events: (a) Creation of a *Value* instance; (b) Modification of a *Value* instance; (c) Deletion of a *Value* instance. If each data type has a corresponding separate table like in case of EAV/CR approach, then each of these tables must have these triggers. If data changes at the knowledge level, then triggers have to be created/altered/dropped as well. This means that the system has to generate and execute DDL statements after all (see the advantages of "universal design"). In addition, the system has to check whether the existing data violates new rules and in case of violation prohibit the changes. These triggers do the work that is implicitly done by a DBMS in case of the "regular design".

Let us assume that we want to enforce two rules in a pattern repository:

1. Name of a pattern cannot be an empty string or a string that consists of spaces.

2. Creation time of a pattern must be smaller or equal than its last modification time.

Let us assume that table *Pattern* in the "regular design" contains columns *name*, *creation_time*, *last_modification_time* among others. Both rules can be enforced by using table level check constraints of table *Pattern* ((5) and (6)):

CONSTRAINT chk_pattern_name CHECK(Trim(name)<>'') (5)

CONSTRAINT chk_pattern_creation_modification
CHECK(creation_time<=last_modification_time) (6)

Function *Trim* removes spaces from the beginning an end of a string.

The first rule can be enforced by a table level constraint of table *Value* in case of the "universal design" (7):

CONSTRAINT chk_pattern_name CHECK(attribute_id= *val1* AND
Trim(string_value)<>'') (7)

The value *val1* identifies attribute *pattern_name*. It is possible to enforce the second rule by using the assertion (8):

CREATE ASSERTION chk_pattern_creation_modification CHECK
((SELECT Count(*) AS amt FROM Value INNER JOIN Value AS
Val_1 ON Value.entity_id = Val_1.entity_id WHERE
(Value.attribute_id= *val3*) AND (Val_1.attribute_id= *val2*) AND
(Value.timestamp_value<Val_1.timestamp_value))=0); (8)

The values *val2* and *val3* identify attributes *creation_time* and *last_modification_time*, respectively. Note that expression of this constraint contains self join of a table that probably contains the biggest amount of rows in a database. If for some reasons the values *val1*, *val2*, or *val3* change in a database (someone modifies attribute identifiers in table *Attrbute*), then the constraints have to be rewritten as well. Otherwise, they will enforce incorrect rules.

In addition, assertions are not supported by current DBMSs. We also remind that one expected benefit of the "universal design" was that even the users who are not database experts could extend the database. Question remains – who creates these constraints? If these constraints are generated by the system, then the system has to create more complex constraints in case of the "universal design" than in case of the "regular design". Another possible solution is to use as little constraints as possible and permit recording of inconsistent and incomplete data. It is possible to use queries in order to find inconsistencies and incompletenesses, but the query expressions are also more complicated compared to the "regular design".

Columns in table *Value* must be as "flexible" as possible. Examples:

- All the columns in table *Value* that correspond to the different supported data types must be optional (permit NULLs) (see Figure 47). For example, if a row in table *Value* contains a value that corresponds to column *int_value*, then columns *string_value*, *timestamp_value*, *boolean_value* etc.

must have NULLs in this row. EAV/CR approach prevents the use of such large amount of NULL's because each supported data type has a separate table (see advantage 3).

- The specification of the maximum length, in characters, of acceptable values in column *string_value* (it has type VARCHAR), must be as big as possible in particular DBMS. For example, this column could contain names of the patterns that should contain less than 50 characters and problem statements that should contain less than 4000 characters. It also means that it is possible to record names of the patterns that consist of approximately 4000 characters. We could create a constraint in order to prevent that.

Column *entity_id* of table *Entity* contains identifiers of entities. This column is a primary key column. The values in this column are probably system generated and do not prevent real data duplication in a database. It is difficult if not impossible to declare that a set of attributes of an entity type must have unique values in case of the "universal design". For example, let us assume that a repository that uses the "universal design" has to store data about the patterns. Let us assume that the patterns are uniquely identified by their names (name has the type VARCHAR). The following constraint of table *Value* (9) does not give the desired result because the column *string_value* contains values of many different attributes. For example, maybe we want to record data about the documents and the name of a document can be the same as the name of some pattern.

```
CONSTRAINT ak_document UNIQUE(string_value)                    (9)
```

Sometimes it is possible to use proprietary solutions in order to solve this problem. For example, in PostgreSQL we could use the following statement (10) in order to declare the key that consists of one attribute. In this case, *val4* is identifier of attribute *name* that belongs to entity type *Pattern*.

```
CREATE UNIQUE INDEX idx_document ON Value (string_value)       (10)
                    WHERE attribute_id= val4;
```

We note that the SQL standard does not specify indexes and therefore this solution is not universal. In this case, we do not declare database constraints that belong to the conceptual level of a database, but indexes that are constructs of the database internal level. We have the problem (as with the constraints (7) and (8)) that if someone changes the identifier of attribute *name* (in table *Attribute*), then this index enforces incorrect rule.

It is possible that the complexity of defining constraints leads to a database with few constraints. Constraint checking, if any, is done by the application that uses a database. It is likely that many constraints are not checked at all because they need complex queries (12).

Compensating actions. A DBMS can sometimes resolve constraint violations as they arrive by executing a compensating action. We have to implement some compensating actions by using triggers. For example, if we wish that deletion of an entity with the type ET1 should cause cascading deletion of all the related entities (see entity type *Relationship* in Figure 47) with the type ET2, then the use of "ON DELETE CASCADE" option in the foreign key declaration is not enough and we have to create a trigger.

***Default values*:** SQL permits *declaration* of *one* default value for a column. This feature is not always usable in case of "universal design". For example, attributes of patterns *registration_time* and *next_revision_time* can have the default values *Date()* and *Date()+'3 months'*, respectively. The values of these attributes are in the same column *timestamp_value* of table *Value* in case of the "universal design". Therefore, we have to use the triggers in order to use the default values. For example, if we decide to create one trigger, then it must contain a set of if-then statement, each of which specifies a default value of an attribute. Values *val6* and *val7* are identifiers of attributes *registration_time* and *next_revision_tine*, respectively.

```
IF new.attribute_id==val6 THEN new.timestamp_value:=Date;          (11)
    ELSE IF new.attribute_id == val7 THEN
    new.timestamp_value:=Date()+'3 months';
```

An alternative is to create many triggers, each of which specifies default value of only one attribute. If a database user specifies new attributes of an entity type or modifies the existing ones, then the system may have to generate and execute DDL statements for creating, replacing or removing triggers.

***Query complexity*:**

```
SELECT A.name FROM (SELECT Entity.entity_id, Val.string_value       (12)
AS name FROM ((Entity_type INNER JOIN Attribute ON
Entity_type.entity_type_id = Attribute.entity_type_id) INNER JOIN
Entity ON Entity_type.entity_type_id = Entity.entity_type_id) INNER
JOIN Val ON (Entity.entity_id = Val.entity_id) AND
(Attribute.attribute_id = Val.attribute_id) WHERE
Entity_type.name='Pattern' AND Attribute.name='name') AS A INNER
JOIN (SELECT Entity.entity_id, * FROM (Entity_type INNER JOIN
Attribute ON Entity_type.entity_type_id = Attribute.entity_type_id)
INNER JOIN Entity ON Entity_type.entity_type_id =
Entity.entity_type_id WHERE Entity_type.name='Pattern' AND
Attribute.name='solution' AND NOT EXISTS (SELECT * FROM Val
WHERE Val.attribute_id=Attribute.attribute_id AND
Val.entity_id=Entity.entity_id )) AS B ON A.entity_id=B.entity_id;
```

Let us assume that a database that is created based on the "universal design" contains data about the patterns (entity type *Pattern*) that have a name, a problem description and a solution description (attributes *name, problem* and *solution*, respectively). The query (12) finds names of patterns that have no solution description:

In case of the "regular design", we could solve the same problem with the query (13):

SELECT name FROM Pattern WHERE solution IS NULL; (13)

It is possible to simplify the query writing task by creating operators (Do and Rahm, 2004) or views. However, after performing the view resolution a DBMS still has to execute a complex query even in case of the simple problems. It causes *performance problems* that are reported, for example, by Kyte (2003) and Wang et al. (2004). Chen et al. (2000) propose to use combinations of simpler queries and temporary tables in order to speed up the queries. In this case, a user of a database looses an advantage of a DBMS according to which a user can make a (complex) query and a DBMS decides how to execute it. In this case, a user has to describe a procedure how to retrieve the desired results.

Size of data. Chen et al. (2000) write: "The EAV/CR representation consumed approximately four times the storage of our conventional schema." Conventional schema is created according to the "traditional design".

***Missing information*:** If value of an attribute is missing, then one possibility is not to record a row in table *Value*. However, there are many reasons why value of an attribute could be missing (Date, 2003, p. 577). It is a useful data that could be recorded in a database. Existing research about the "universal design" does not deal with this problem. We have several options:

1. D. McGoveran (in Date, 1998, p. 381) suggests to record reason for missing data in a special metadata table.
2. Date and Darwen (2000) propose that the definitions of a scalar type could specify the special values that represent different reasons, why a value is missing. Currently it is not possible to declare such special values in SQL type definitions.
3. Darwen (2003) proposes to use vertical and horizontal decomposition of tables in order to prevent combination of multiple meanings in a single table.

Figure 49 presents a conceptual data model of a possible solution to this problem in case of "universal design".

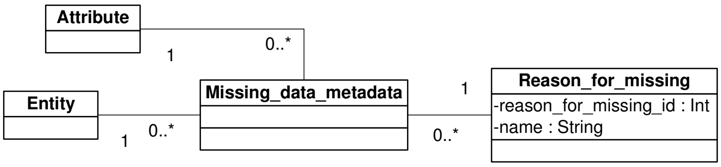


Figure 49 A possible solution for recording reasons of missing data

Table *Missing_data_metadata* has the following external predicate (14):

Value of an attribute *attribute_id* of entity *entity_id* is missing because of the reason *reason_for_missing_id*. (14)

This design could be extended further by the many-to-many relationship between *Attribute* and *Reason_for_missing*. This relationship models the fact that different attributes can have different possible reasons, why the attribute value might be missing.

The following rule also has to be enforced: if a row in table *Entity* has an associated row in table *Value*, then it cannot have the associated row in table *Missing_data_metadata*. If a value is recorded in a database, then corresponding row in table *Missing_data_metadata* has to be deleted.

***Dependencies between database objects*:** Database objects like triggers, declarative constraints, conditional indexes and views depend on the specifications at the knowledge level (see Figure 47). Data changes at this level can cause creation, modification or removal of these database objects. Dependencies between the database objects are automatically recorded in a database catalog by a DBMS. A database has to contain explicitly defined tables in order to record the dependencies in case of the "universal design".

***Access control*:** Access control mechanisms that are provided by a DBMS are not sufficient in case of the "universal design". SQL provides statements for granting privileges that allow us to perform a given action on a specified table or column. Let us assume that a database contains data that corresponds to entity types ET1, ET2 and ET3. Let us also assume that user U1:

- has right to SELECT and UPDATE data that correspond to ET1,
- has right to SELECT data that corresponds to ET2,
- does not have rights to use the data that corresponds to ET3.

It is unreasonable to grant to user U1 direct access to tables *Entity* and *Value* because these tables contain data about all the entities. Instead, we have to create two views that present data about entity types ET1 and ET2, respectively. Then we can give to the user U1 rights to use these views in order to see or modify data. We have to bear in mind that in some DBMSs (like PostgreSQL 8.0) it is not possible to modify data in base tables through views without further programming. The result might be that the systems, which use a database that follows the "universal design", do not use the security mechanisms of a DBMS, in order to restrict access to the data.

***Concurrency control*:** Locking is widely used mechanism for concurrency control by DBMSs. Examples of the situations that need special care in case of the "universal design":

- Explicit locking of all the attribute values of an entity. For example, two separate users could change a pattern concurrently so that one modifies the problem statement and another modifies the description of solution. The result of these modifications could be an incorrect pattern.
- Explicit locking of all the data values that correspond to an entity type.
- Modifications at the knowledge level that influence the operational level should restrict concurrent data changes at the operational level. For example, if data type of an attribute changes, then at the same time system should not allow us to register new values of this attribute or to change existing value of the attribute.

According to the most basic locking strategy, if a transaction requests a read lock (shared lock) on a data item, then other transactions cannot request a write lock (exclusive lock) on the same data item in order to update it (Weikum and Vossen, 2002, p. 131). Therefore, making query about an entity prevents the concurrent updates of its attribute values. Some DBMSs like PostgreSQL and Oracle use the multiversion concurrency control mechanism (Weikum and Vossen, 2002, p. 185). Versions of the modified data are kept by a DBMS in order to allow us to answer to the queries even if the data is modified at the same time. In this case reading of data does not block its concurrent updates and vice versa. Therefore, making query about the entity does not prevent the concurrent updates of its different attribute values. We have to use SELECT statement with special syntax in order to lock the necessary data.

***User interface design*:** Marenco et al. (2003) acknowledges that data in a database that follows the "universal design" "must be transiently converted (‘‘pivoted’’) into a conventional representation through fairly elaborate metadata-driven code." (Marenco et al., 2003) Conventional representation means that each data element is presented in the separate field with the meaningful label.

In conclusion, we can say that the "universal design" advocates building a DBMS on top of a DBMS. Knowledge level of a schema is actually a database catalog – addition to the one that is automatically created by a DBMS. Designers have to work out many *ad hoc* solutions and do redundant work instead of relying on the built-in features of DBMSs. Many features that are present in a DBMS have to be duplicated in the applications.

Database is "a collection of true propositions" (Date, 2003, 15). A DBMS cannot enforce truth, but as an approximation it can check that all the data values are consistent (i.e., conform to the integrity constraints) (Date, 2003). Not all the consistent propositions are correct, but all the correct propositions must be consistent. In our view, the "universal design" advocates a database where the consistency is not the most important property (due to the difficulties to enforce the integrity rules).

3.2 Checking of the Well-formedness Rules

Current CASE tools provide mostly built-in constraint checks that are implemented in their code. A Meta-CASE tool uses constraints that are embedded in its code and run-time constraints that are enforced by a constraint manager (Gray and Welland, 1999). Dittrich et al. (2000) note that one advantage of using DBMSs in the software engineering environments is their integrity control mechanisms. It can help to "automate manual consistency checks, provide information about tool-provoked errors, and simplify the logic implemented in software tools". A constraint manager that ensures enforcement of the constraints is component of a DBMS. Constraints that are enforced by a DBMS constraint mechanism are called *run-time constraints*.

Rasmussen (2005) describes different policies of constraint checking in CASE tools. These policies are applicable in software engineering systems in general. Next, we describe how it is possible to implement some of them in an ORDBMS$_{TTM}$ database.

3.2.1 Ignore All Well-formedness Rules

Relvars with the minimal possible set of constraints must be created in case of the policy that ignores all well-formedness rules.

What constraints belong to the minimal possible set of constraints? By definition, each attribute of a relvar must have a type. This means enforcement of an attribute constraint. Each relvar must by definition have at least one candidate key. In addition, foreign key attributes must have foreign key constraints that enforce the referential integrity rule.

One could say that this policy is useful if the repository is used for storing artifacts that have been created by tools, which are not integrated with a repository. One could say that these tools and not the repository are responsible for checking the artifacts. However, the constraint checking in a repository is a second defence-line that prevents spreading of the artifacts that are incomplete, incorrect and inconsistent.

There are situations when this design is most appropriate. Lavazza and Agostini (2005) describe UML Model Measurement Tool (UMMT) that calculates metrics values based on UML models. This tool must be able to calculate metrics values in case of incomplete and inconsistent models as well.

3.2.2 Automatically Resolve Constraint Violations as they Arrive

It can be achieved in a database by using compensating actions, which occur if a DBMS discovers constraint violation. "ON DELETE CASCADE" and "ON UPDATE SET DEFAULT", which are used in the definition of the foreign keys, are examples of the compensating actions.

In current ORDBMS$_{SQL}$s compensating actions can be implemented by using triggered procedures. The Third Manifesto does not prohibit triggers and therefore it is also possible in an ORDBMS$_{TTM}$ database.

It would be useful if a DBMS could allow us to specify a compensating action that is associated with a constraint. This is actually a special kind of trigger, the triggering event of which is the occurrence of a constraint violation. If a DBMS discovers the constraint violation, then it must execute a set of operations and check the constraint again after these operations have been performed.

3.2.3 Disallow an Operation and Inform a User

Another approach is to disallow an operation that leads to the violation of a well-formedness rule and inform a user about that. In this case, we have to use the integrity constraints (type, database and transition constraints). If a new value is assigned to a relvar, then a DBMS checks whether this value is correct in terms of the integrity constraints and rejects changes that are incorrect. This policy also requires that the certain operations must be atomic. For example, the task of recording data about the whole instance and its two mandatory part instances would need to be done as a single atomic operation. It can be done in an ORDBMS$_{TTM}$ by using multiple assignment operations (Date, 2003).

The problem of this policy is that an artifact can be initially "incorrect" and only at some point must become correct. However, a DBMS does not allow in a database data that does not conform to the integrity constraints. Next, we present examples of statements for creating integrity constraints.

3.2.3.1 Example with SimpleM

In this section, we illustrate the "Disallow an Operation and Inform a User" approach by using a simple software design language SimpleM (see Figure 46). Serrano (1999) describes well-formedness rules of the language. We have modified rules R1 and R2.

1. (R1): Both *StartState* and *State* have a label with a name that is unique amongst all other states.
2. (R2): There is at most one *StartState* in a repository.
3. (R3): "The *StartState* can only be connected to *States* by outgoing *Events*." (Serrano, 1999)
4. (R4): "Any pair of *States* is connected at most by two *Events*, one in each direction." (Serrano, 1999)
5. (R5): "Loop *Events*, i.e. *Events* that connect a *State* to itself, are not allowed." (Serrano, 1999)

Next, we present the database constraints that implement these rules.

Each relvar must have at least one candidate key. We can enforce some well-formedness rules by creating appropriate key constraints.

Rule R1 can be enforced by creating the constraint KEY{name} in relvar *_State*.

Rule R4 can be enforced by creating the constraint KEY{origin, destination} in relvar *_Event*.

Constraint KEY {artifact_id#, element_id#} in relvar *Element_in_artifact* ensures, that each element can participate only once in an artifact. Together with constraint C_2 (15) they guarantee that each diagram (artifact) can contain at most one *StartState*.

Rules R2, R3 and R5 are enforced by database constraints C_2 (15), C_3 (16) and C_5 (17), respectively. C_3 is created based on the reformulation of R3 to the equivalent rule R3'. (R3'): "*StartState* cannot be destination of any event."

CONSTRAINT C_2 (COUNT(_StartState)<=1); (15)

CONSTRAINT C_3 IS_EMPTY ((_Event RENAME (element_id# AS el_id#, destination AS element_id#) JOIN _State) JOIN _StartState); (16)

CONSTRAINT C_5 (IS_EMPTY (_Event WHERE origin=destination)); (17)

IS_EMPTY (<relation exp>) is a scalar operator that evaluates to TRUE if body of the relation denoted by <relation exp> contains no tuples (Date et al., 2003). IS_EMPTY is built-in operator in Rel.

Database constraint C_6 (18) ensures that each element is part of at least one artifact. It uses relational operator SEMIMINUS (Date, 2003) in order to find tuples of one relation that have no counterpart in another.

CONSTRAINT C_6 IS_EMPTY (Artifact_element SEMIMINUS Element_in_artifact); (18)

Constraints C_3, C_5 and C_6 could also be created using Count operator (Count(<relation exp>)=0). However, the use of IS_EMPTY operator allows DBMS to optimise execution of <relation exp> (see section 3.5.2)

3.2.3.2 Example with Use Cases

In this section, we illustrate the approach by using a fragment of UML language (see Figure 46). UML 2.0 specification (OMG formal/05-07-04) states the following rules about the elements of the metamodel fragment:

1. (R1) A *UseCase* must have a name.
2. (R2) *UseCases* can only be involved in binary *Associations*.
3. (R3) *UseCases* cannot have *Associations* to *UseCases* specifying the same subject.
4. (R4) A use case cannot include use cases that directly or indirectly include it (see Figure 50).

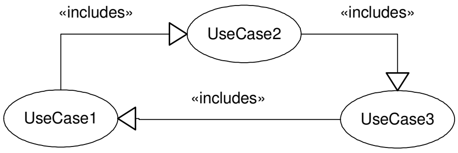


Figure 50. Illegal relationships between use cases

5. (R5) An *Actor* can only have associations to *UseCases*, *Components*, and *Classes*. Furthermore, these associations must be binary.
6. (R6) An *Actor* must have a name.
7. (R7) We also conclude based on the specification that inclusion relationship must be between two *different* use-cases.

Next, we present the constraints that implement these rules. Rules R1 and R6 are by default enforced by an ORDBMS$_{TTM}$ because all attributes in a relation must have a value. If there is a possibility that attribute cannot have a value, then a designer of the type of the attribute must define "special values" for dealing with the missing information.

Some rules are enforced by the structure of the relvars.

Rule R2 is enforced by the structure of relvar *_Include*. It has two foreign key attributes and therefore allows only to record data about the binary relationships.

Rule R5 is enforced by the database structure because currently actors (or in general classifiers) can only have associations with use-cases.

Rule R3 can be enforced by the following database constraint (C_3) (19). Subsection of the constraint with **bold** font finds pairs of use cases (actually their identifiers) that are associated with the inclusion relationship. We have to use RENAME operator because a relation cannot have two attributes with the same name. Subsection of the constraint with *italic* font finds pairs of use cases (actually their identifiers) that are associated with the same subject.

CONSTRAINT C_3 (IS_EMPTY(((**_Include RENAME (addition AS use_case)) RENAME (including AS uc) {use_case, uc}**) INTERSECT (19)
((((_Classifier_use_case RENAME (use_case AS uc)) JOIN (_Classifier_use_case RENAME (classifier AS cl))) WHERE classifier=cl AND use_case<>uc) {uc, use_case})));

We use intersection operation in order to find the pairs that belong to both these sets. The set of pairs that belong to both these sets must be empty. *Tutorial D* does not provide explicitly Cartesian product operator and it makes syntax of the constraint C_3 more complicated.

Built-in transitive closure operator TCLOSE (Date, 2003, p. 203) can be used in order to enforce rule R4. Constraint C_4 (20) also enforces rule R7.

CONSTRAINT C_4 (IS_EMPTY((TCLOSE (_Include {including, addition})) WHERE including=addition)); (20)

3.2.4 Allow Everything Initially and Search Errors Later

Gray and Welland (1999) call the constraints, the violation of which is initially allowed "soft constraints". On the other hand, constraints that must be immediately satisfied (see section 3.2.3) are called "hard constraints". In case of this policy, we propose two approaches.

Approach 1: Create relvars with the minimal possible set of constraints. Create a set of queries in order to find violations of the well-formedness rules. It is possible to create more than one query based on a rule. One query checks whether an artifact follows this particular rule or not and returns a Boolean value. Another query presents information about the artifact elements that violate this rule. The third one could find the artifact elements, which satisfy the rule. User of a system can execute the queries at any time in order to check artifacts. A repository database must contain specific real relvars in order to record these queries.

Approach 2: Create two sets of relvars. The first set (let us call it A) uses the minimal possible set of constraints. The second set (let us call it B) of relvars is accompanied with the integrity constraints that help to enforce all well-formedness rules. If a user wants to check an artifacts, then the system must read the artifact from the relations in the set A and try to record it in the relations in the set B. It must be done as a single atomic operation by using multiple assignment operations. If the artifact is correct in terms of the constraints, then this operation succeeds otherwise it does not succeed. The problem is that a DBMS reports only the first error, even if there are more errors in the artifact. The user has to fix this error and check the artifact again in order to find the next error. The user does not know in advance how many more improvements the artifact will need.

3.2.4.1 Integrated Approach with Versioning

Next, we propose the design that is synthesized based on the schema design approach "Non-encapsulated Artifact Element Types" (see section 3.2.4.1), the constraint checking approach "Allow Everything Initially and Search Errors Later" (see approach 2 in section 3.2.4) and the proposal about how to keep temporal data in an $\text{ORDBMS}_{\text{TTM}}$ database (Date et al., 2003). A repository database should consist of five sets of relvars:

1. Values of relvars (relations) in the set S present *current data* that may or may not be validated.
2. Relations in the set $S_{\text{validated}}$ present *current and validated* data.
3. Relations in the set S_{history} present *historical* data that may or may not be validated.
4. Relations in the set $S_{\text{validated\_history}}$ present *historical and validated* data.
5. Relvars in the set $S_{\text{no\_history\_and\_version\_control}}$ are the so-called "regular" relvars. In case of them, we do not want to know historical data and do not want to validate data by using approach 2 from section 3.2.4. Creation of the relvars that belong to this set is optional and depends on the needs of a system that uses this database.

Values of the relvars in S and S_{history} present so-called initial artifacts. Values of the relvars in $S_{\text{validated}}$ and $S_{\text{validated\_history}}$ present so-called validated artifacts. Now, we shortly explain the procedure of data registration and validation. Firstly, data is registered by using relvars in S. If we modify data in a relation in S, then the system automatically registers the historic attribute values by using the corresponding relvars that belong to S_{history}. If an artifact (the data of which is in the relations in S) is successfully validated, then the values of the corresponding relvars in $S_{\text{validated}}$ will change. The system automatically registers historic and validated attribute values by using relvars in $S_{\text{validated\_history}}$.

We explain the proposed approach by using a small example. Let us assume that a repository has to contain information about states (that are part of a state-transition model). Next, we list *external* predicates of the real relvars. Relvar STATE in S has predicate (21). Relvar STATE_VALIDATED in $S_{\text{validated}}$ has predicate (22). Relvars STATE_DURING and STATE_NAME_DURING in S_{history} have predicates (23) and (24), respectively. Relvars STATE_VALIDATED_DURING and STATE_VALIDATED_NAME_DURING in $S_{\text{validated\_history}}$ have predicates (25) and (26), respectively. The parameters of the predicates are written in capital letters. In this example, we assume that all states belong to one artifact. In general, the repository can contain more than one artifact.

State ELEMENT_ID# has been in the initial artifact ever since ELEMENT_ID#_SINCE (and not the time point immediately before ELEMENT_ID#_SINCE), and has been named NAME ever since NAME_SINCE (and not the time point immediately before NAME_SINCE). (21)

Validated state ELEMENT_ID# has been in the validated artifact ever since ELEMENT_ID#_SINCE (and not the time point immediately before ELEMENT_ID#_SINCE), and has been named with valid name NAME ever since NAME_SINCE (and not the time point immediately before NAME_SINCE). (22)

From the time point that is the beginning point of DURING (and not on the point immediately before that point) to the time point that is the end point of DURING (and not on the point immediately after that point), inclusive, state ELEMENT_ID# was in the initial artifact. (23)

From the time point that is the beginning point of DURING (and not on the point immediately before that point) to the time point that is the end point of DURING (and not on the point immediately after that point), inclusive, state ELEMENT_ID# in the initial artifact had name NAME. (24)

From the time point that is the beginning point of DURING (and not on the point immediately before that point) to the time point that is the end point of DURING (and not on the point immediately after that point), inclusive, validated state ELEMENT_ID# was in the validated artifact. (25)

From the time point that is the beginning point of DURING (and not on the point immediately before that point) to the time point that is the end point of DURING (and not on the point immediately after that point), inclusive, validated state ELEMENT_ID# had valid name NAME. (26)

A designer has to select granularity of time points that are used in a system. Date et al. (2003 p. 62) write that time points are "time units that are relevant for some particular purpose, which might be days or months or milliseconds". For example, if designer decides the granularity is one *second*, then it means that the system has to consider it as an indivisible point. The model of timeline for computing purposes consists of discrete points that have this granularity.

The repository database contains constraints that implement well-formedness rules of artifacts. Relvars in the different sets have different constraints. A repository designer has to choose, which constraints to create only in $S_{validated}$ and which constraints to create in both S and $S_{validated}$. If we want to allow violation of a well-formedness rule in an initial artifact, then we have to create the corresponding integrity constraint in $S_{validated}$ and not in S. If we want to prohibit violation of a well-formedness rule in an initial artifact, then we have to create the corresponding integrity constraints in both S and $S_{validated}$.

Gray and Welland (1999) describe hard and soft constraints. If a rule has the corresponding constraints in both S and $S_{validated}$, then we can say that this rule is enforced by using a *hard runtime constraint*. The word "hard" means that this constraint cannot be temporarily violated. The word "runtime" means that the constraint is enforced by a constraint manager of a DBMS and the constraint is not directly embedded in the code of the run-time system (SES). If a rule has a corresponding constraint in $S_{validated}$ (but not in S), then we can say that this rule is enforced by using a *soft runtime constraint*. The word "soft" means that this constraint can be temporarily violated.

Let us define a rule that name of a state must be at least four characters long. Let us decide that the system must enforce this rule by using a soft run-time constraint. We have to create the constraint in $S_{validated}$. This could be a database constraint that is associated with relvar STATE_VALIDATED or a type constraint that is associated with the type of attribute NAME of relvar STATE_VALIDATED.

(t01)

STATE

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t01	Accepted	t01

(t02)

STATE_VALIDATED

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t02	Accepted	t02

(t03)

STATE

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t01	Accpt.	t03

STATE_NAME_DURING

ELEMENT_ID#	NAME	DURING
1	Accepted	[t01:t02]

(t04)

STATE_VALIDATED

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t02	Accpt.	t04

STATE_VALIDATED_NAME_DURING

ELEMENT_ID#	NAME	DURING
1	Accepted	[t02:t03]

(t05)

STATE

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t01	Accpt.	t03
2	t05	B	t05

(t06)

STATE

ELEMENT_ID#	ELEMENT_ID#_SINCE	NAME	NAME_SINCE
1	t01	Accpt.	t03

STATE_DURING

ELEMENT_ID#	DURING
2	[t05:t05]

STATE_NAME_DURING

ELEMENT_ID#	NAME	DURING
1	Accepted	[t01:t02]
2	B	[t05:t05]

Figure 51 Creation, modification and validation of a new state

The scenario that is illustrated by Figure 51 is following:

1. A user defines the state "Accepted" at the time point t01. New tuple is added to relation STATE.
2. A user wishes to validate the state "Accepted" at the time point t02. The system tries to add a new tuple to relation STATE_VALIDATED because

this state is validated the first time. If it succeeds, then relvar STATE_VALIDATED obtains a new value. The system must use multiple assignment operations in order to enforce atomicity of the validation operation. If the DBMS rejects an assignment operation because its result does not conform to the integrity constraints, then it must roll back all operations that are part of this multiple assignment operation.

3. A user renames the state "Accepted" to "Accpt." at the time point t03. The system updates tuple in relation STATE. It changes values of attributes NAME and NAME_SINCE. Modification of the tuple in relation STATE causes the system to automatically add tuple with the historic name to relation STATE_NAME_DURING. The value of attribute DURING shows the period when the value of attribute NAME was the name of the state.
4. A user validates the renamed state at the time point t04. The existing tuple is updated in relation STATE_VALIDATED because this state (with the same element ID) has already been validated and the element with the identifier value 1 is already in the relation. Modification of the tuple in relation STATE_VALIDATED causes the system to automatically add the tuple with the historic name to relation STATE_VALIDATED_NAME_DURING.
5. A user defines a new state at the time point t05. Such value of relvar STATE is allowed because it has no constraint about the length of the name. If user chooses to validate this new state, then this state is rejected because of the integrity constraint in $S_{validated}$.
6. A user deletes the state "B" at the time point t06. The system deletes tuple from relation STATE and assigns new values to relvars STATE_DURING and STATE_NAME_DURING.

Date et al. (2003) distinguishes the concepts "stated time" and "logged time". According to Date et al. (2003), "stated times are the times when, according to our current beliefs, something is, was, or will be true." According to Date et al. (2003), "logged times are the times when the database said we believe something is, was, or will be true." Times that are used in proposed versioning approach (in our example, values of all the relvar attributes, the name of which contains "SINCE" or "DURING") are logged times and must be recorded by the system. The system must not allow us to change the logged times (Date et al., 2003). Users cannot modify a validated artifact directly. They must modify the initial artifact and validate the changes. The system must also forbid all the changes of relvar values in $S_{history}$ and $S_{validated\_history}$ that are not caused by the changes in S and $S_{validated}$, respectively.

The proposed approach allows us to find old versions of the artifact elements and artifacts. For example, the system can give an answer to the question: "What was the validated name of the state with the element_id#=1 at the time point t02?" A version of an artifact (either initial or validated) can be found by using the set of queries. The system can reconstruct an artifact, by finding the values of artifact elements that were current at the given time point. Therefore, this system supports intensional versioning. In case of intensional versioning, a version is constructed in response to some query (Conradi and Westfechtel, 1998).

If we want to explicitly identify different versions, then each version must have unique *version identifier* (VID) that could be system generated (Conradi and Westfechtel, 1998). In this case, we should create separate real relvar *Version* that belongs to $S_{no\_history\_and\_version\_control}$. This relvar could have type: RELATION {version_id INTEGER, artifact_id INTEGER, version_time DATE}. Creation of a new version means insertion of new tuple to relation *Version*. In this case, we determine explicitly the time point that must be used in the queries in order to restore the artifact.

We can say that data about the entity types that have corresponding relvars in S, $S_{history}$, $S_{validated}$ and $S_{validated\_history}$ is put under the version control. All the relvars that must belong to these sets must have additional constraints because they contain temporal data. Thorough discussion of these constraints is presented by Date et al. (2003, chapter 11, 12). Example of the constraints:

- The fact, that a validated artifact element had some name *n* at time point *t*, can be recorded only in one tuple in the database.

We have to create much more relvars and constraints than in case of "regular design". This design approach needs supporting development environment that is able to generate DDL statements for creating necessary relvars, types and constraints.

3.3 Preserving the Semantics of Relationships in a Database

A conceptual data model that is created, for example, in UML (OMG formal/03-03-01) can contain aggregation, composition and generalization relationships between entity types. These types of relationships are also often used in the class diagrams that present abstract syntax of modeling language (and that are the basis for the creation of repository information model). For example, the class diagram that specifies abstract syntax of use-cases (OMG formal/05-07-04, p. 570) in UML2.0 contains six different generalization relationship instances and five different composition relationship instances. This section explains how to preserve semantics of these relationships in an $ORDBMS_{TTM}$ database.

A DBMS does not "understand" semantics of a relationship the same way as humans do - based on the names of a relationship and its participants (Date and McGoveran, 1994). However, a DBMS is able to enforce structural and operational properties of the relationships and objects, which participate in these relationships (Zhang et al., 2001). These properties depend on the type of relationship.

3.3.1 Generalization Relationships

Date and Darwen (2000, p. 397) describe possible $ORDBMS_{TTM}$ database design approach in case of generalization relationship. If a conceptual data model contains entity types ET_{super} and ET_{sub} where ET_{super} and ET_{sub} are supertype and subtype, respectively, then a database should contain real relvars RR_{super} and RR_{sub} that correspond to ET_{super} and ET_{sub}, respectively. In addition, a database should contain a virtual relvar that joins relations RR_{super} and RR_{sub}. Value of the virtual relvar must be updateable and updates must propagate to the values of real relvars (Date and Darwen, 2000). Relational language could have special statement (as shorthand) in order to create relvar that is conceptually associated with other relvar through generalization relationship (Pascal, 2000). This kind of statement causes creation of necessary real- and virtual relvars.

3.3.2 Whole-Part Relationships

Different authors have done a lot of research about the semantics of the aggregation and composition relationships. Examples of the recent research are works of Barbier et al. (2003) and Guizzardi (2005). Their view is that UML (at least prior to the version 2.0) does not define the semantics of this kind of relationships precisely enough. Therefore, we use instead the concept "whole-part relationship".

3.3.2.1 Related Works

Some researchers have investigated how to preserve semantics of whole-part relationships in an $ORDBMS_{SQL}$ database.

The first approach is to extend the OR_{SQL} data model with the relationships as first class objects. For example, extension module ORIENT (Zhang et al., 2001) extends Informix $ORDBMS_{SQL}$ by providing CREATE RELATIONSHIP statement and means for recording and using relationship data.

The second approach tries to add support to the relationships by using existing facilities of DBMSs and their underlying data models. SQL:2003 defines type constructors ROW, ARRAY, REF and MULTISET and permits creation of the user defined structured types (UDTs) (Melton, 2003). We could use these types in order to implement whole-part relationships. If we look the *picture* of the table that has a column with a complex data type, then we see that data about the whole instance contains data about its associated part instances. Hammer and Mc Leod (1981) describe Semantic Database Model: "The constructs of the database model should provide for the explicit specification of a large portion of the meaning of a database." Researchers have already suggested to implement whole-part relationships in an $ORDBMS_{SQL}$ database by using array- or table types (the latter is interpretation of a multiset type in Oracle DBMS) (Marcos et al., 2001), indexed clusters or table types (features in Oracle DBMS) (Rahayu and Taniar, 2002) or multiset- or row types (Pardede et al., 2005). Data about the part instances can be recorded in the columns that have complex data types and hence data about the whole instances and their part instances can be recorded in one table at the conceptual level. The use of clusters in Oracle means that data about the whole- and part instances can be recorded together at the internal level, but they remain in the separate tables at the conceptual level.

Our comment about the array types is that array is a collection in which elements have a defined order and the same element can be in the collection more than once. Tuples in the body of a relation are unordered and relations cannot contain duplicated tuples (Date and Darwen, 2000). Therefore, we cannot use arrays in order to implement relationships if we want to treat their participants in a uniform way.

Proponents claim that the object-relational features help to implement relationships in more natural and semantics-preserving ways. However, researches have also identified problems of using collections in conceptual modeling (Halpin and Bloesch, 2000) and in database schemas. "A collection is a composite value comprising zero or more elements, each a value of some data type DT." (Melton, 2003, p. 45) Halpin and Bloesch (2000) note that collections make harder to express constraints (which typically occur on members, not collections) in a conceptual model. If it is difficult to use declarative language like OCL in order to express constraints in a conceptual model, then it is also difficult to express declarative constraints and queries on collections in a database. Smith and Smith (1977) propose to use complex types as domains for the attributes in relations in order to record semantically important information about an aggregation of objects in a relational database. They also identify possible problems that include restrictions to ways how user can access data and duplication of data. The latter causes waist of storage space as well as introduces problems of possible inconsistency. One solution could be the use of pointers, but "Pointers are objects which have no real-world analogy and serve to dramatically increase the complexity of database interactions." (Smith and Smith, 1977) Soutou (2001) has also identified this problem and writes: "Collections should model relationships when there are no strong integrity constraints and when there is a particular data access (via a separate relation)." Collections offer little performance gain according to experience of Halpin and Bloesch (2000). "Collections can provide better performance than a standard relational database, but require more complex queries for data retrieving." (Smith and Smith, 1977) Comment to the last observation is that performance is an implementation issue, not a model issue (Date, 2003) and should not be a criterion for evaluating different data models.

Date (2003, p. 374) present the guideline (not strict law) that real relvars without relation-valued attributes should be preferred because they have a simpler logical structure that simplifies operations with the data. His discussion of using attributes with the relation- or tuple types is limited and he gives few examples. Therefore, we think that it is necessary to study more thoroughly the implications of using complex data types in real relvars.

3.3.2.2 Possible Designs

In this section, we present some possible designs of an ORDBMS$_{TTM}$ database structure. For the illustrative purposes, we assume that we have a conceptual data model with entity types *Whole* and *Part*. They are associated with a generic binary whole-part relationship. Entity type *Whole* has the attributes *a* and *b* and *Part* has attributes *c* and *d*. Values of attributes *a* and *c* are unique identifiers of the Wholes and Parts, respectively. We also assume that attributes *a*, *b*, *c* and *d* have type INTEGER (INT).

Declarations of the relvar types (see Table 17) consist of the pairs of attribute and type identifiers. The phrase "part TUPLE {c INT, d INT}" in Table 17 means that the relvar has attribute *part* with a tuple type. Phrases "part RELATION {c INT, d INT}" and "part RELATION{part ST}" mean that the relvar has attribute *part* with a relation type. Type ST is a scalar type that is created based on entity type *Part*. Its possible representation contains components that correspond to attributes *c* and *d*. All relvars that are presented in Table 17 are *real relvar*s.

Table 17 contains illustrations of the values of the relvars. Some designs have the same illustration. The reader must bear in mind that the designs are different because they use the different types.

Designs 1 and 6 are similar to the ones that Rahayu et al. (1998) propose to use in RDBMS$_{SQL}$ databases in case of the collection type *set* in an object-oriented conceptual model. Designs 2-5 use relvar attributes that have *complex types*. They are similar to some of the designs that the researchers (Marcos et al., 2001; Soutou, 2001; Zhang et al., 2001; Pardede et al. 2004) recommend to use in the ORDBMS$_{SQL}$ databases.

Table 17 Design alternatives for implementing a whole-part relationship

<table>
<tr><th>ID</th><th>Types of the real relvars
(<i>relvar name</i> : relvar type)</th><th>Pictures that illustrate values of the relvars</th></tr>
<tr><td>1</td><td><i>Whole</i> : RELATION {a INT, b INT}
<i>Part</i> : RELATION {c INT, d INT, a INT}</td><td>Whole<table><tr><th>a</th><th>b</th></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>4</td></tr></table>Part<table><tr><th>c</th><th>d</th><th>a</th></tr><tr><td>1</td><td>5</td><td>1</td></tr><tr><td>2</td><td>5</td><td>2</td></tr></table></td></tr>
<tr><td>2</td><td><i>Whole</i> :
RELATION {a INT, b INT, part ST}</td><td rowspan="2">Whole<table><tr><th>a</th><th>b</th><th>part</th></tr><tr><td>1</td><td>2</td><td>1, 5</td></tr><tr><td>2</td><td>4</td><td>2, 5</td></tr></table></td></tr>
<tr><td>3</td><td><i>Whole</i> : RELATION {a INT, b INT,
part TUPLE {c INT, d INT}}</td></tr>
<tr><td>4</td><td><i>Whole</i> : RELATION {a INT, b INT,
part RELATION {c INT, d INT}}</td><td rowspan="2">Whole<table><tr><th>a</th><th>b</th><th>part</th></tr><tr><td>1</td><td>2</td><td>1, 5</td></tr><tr><td rowspan="2">2</td><td rowspan="2">4</td><td>2, 5</td></tr><tr><td>3, 6</td></tr></table></td></tr>
<tr><td>5</td><td><i>Whole</i> : RELATION {a INT, b INT,
part RELATION{part ST}}</td></tr>
<tr><td>6</td><td><i>Whole</i> : RELATION {a INT, b INT}
<i>Part</i>: RELATION {c INT, d INT }
<i>PartOfWhole</i>: RELATION {a INT, c INT}</td><td>Whole<table><tr><th>a</th><th>b</th></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>4</td></tr></table>PartOfWhole<table><tr><th>a</th><th>c</th></tr><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>2</td></tr><tr><td>2</td><td>3</td></tr><tr><td>1</td><td>3</td></tr></table>Part<table><tr><th>c</th><th>d</th></tr><tr><td>1</td><td>5</td></tr><tr><td>2</td><td>5</td></tr><tr><td>3</td><td>6</td></tr></table></td></tr>
</table>

Relvar *Part* has one foreign key (attribute *a*) and relvar *PartOfWhole* has two foreign keys (attributes *a* and *c*) in case of designs 1 and 6, respectively. For example, foreign key *a* refers to relvar *Whole* in case of design 1.

All these designs (1-6) require additional constraints depending on the secondary characteristics of the relationship that they help to implement (see next section).

3.3.2.3 Choosing Between the Designs

In this section, we evaluate the designs (1-6) in terms of some of the secondary characteristics of the whole-part relationships (see Table 18): shareability (SH), lifetime dependency (LD), existential dependency (ED) and separability (SP). We refer to these characteristics in column *"Values of the characteristics"* in Table 18 by using the abbreviations that are in brackets. For example, Barbier et al. (2003) and Guizzardi (2005) explain the meaning of these characteristics. See also "Appendix B: Some Secondary Characteristics of Whole-part relationships" that explains some of these characteristics. Pictograms in column *"Relationship constraints"* in Table 18 illustrate the participation and cardinality constraints of the relationships that are imposed by the values of the secondary characteristics. "[W]" and "[P]" denote "Whole" and "Part", respectively.

We give marks (0-4) to these designs based on the possible values of the characteristics. The marks depend on the participation and cardinality constraints and characterize whether it is reasonable to use the design and how much effort it requires. We assume that a database designer wants to enforce consistency of data by using integrity constraints.

If the design is unreasonable because it will cause data redundancy, then we give mark *0*. For example, we could use designs 4 or 5 in case of the relationship: [W]<>-**0..n**----**0..n**-[P]. Pardede et al. (2004) propose to use similar design in case of the shareable parts. However, data about some part instances would be repeatedly recorded (see Figure 52) and it will cause *update anomalies*.

Whole

a	b	part
1	2	1, 5
		2, 5
2	4	***2, 5***
		3, 6
3	9	4, 1
		2, 5
?	?	1, 4

Figure 52 Relation that contains redundant data

Mark 2 means that the design can be used, but besides candidate key and foreign key constraints we have to create additional database constraints.

Mark 3 means that a database designer has to ensure that attributes can have special values for dealing with the "missing information". Additional constraints like in case of mark 2 are not needed. For example, we could use designs 1-5 in case of the relationship: [W]<>-**0..1**-----**0..n**-[P]. In this case, relation *Whole* must contain exactly one tuple with the special values (see "?" in Figure 52). This tuple corresponds to a missing whole instance. The Third Manifesto envisages that declarations of the scalar types can be accompanied by the declarations of the special values, which represent information that is missing or unknown for some reasons (Date and Darwen, 2000) (see section 1.3.2). We can use an empty relation as a special value in case of a relation type. In case of a tuple type, we have to declare that scalar types of attributes of the tuple type permit special values.

Table 18 Comparison of the designs

ID	Values of the	D	e	s	i	g	n	Gr	Relationship
	characteristics	6	1	2	3	4	5	p	constraints
1	LD: lifetime dependency – cases 1, 2, 4, 5.	-	-	-	-	-	-	*5*	[W]<>-1..------[P]
2	LD: lifetime dependency – cases 3, 6, 7, 8, 9.	-	-	-	-	-	-	*5*	[W]<>-0..------[P]
3	SH, SP: locally exclusive part with optional wholes.	4	3	3	1	1	1	*4*	[W]<>-0..1----[P]
4	SP: whole with no more than one optional part.	4	4	3	3	1	1	*4*	[W]<>----0..1-[P]
5	SH: globally exclusive (non-shareable) part.	4	4	4	4	4	4	*4*	-
6	SH: globally shareable part.	4	4	0	2	2	0	*3*	-
7	SH, SP: locally exclusive part with mandatory wholes. ED, SH: inseparable and locally exclusive part.	2	4	4	2	2	2	*3*	[W]<>-1..1----[P]
8	SP: whole with exactly one mandatory part. ED: whole with exactly one essential part.	2	2	4	4	2	2	*3*	[W]<>----1..1-[P]
9	SP: whole with more than one mandatory part. ED: whole with more than one essential part.	2	2	0	0	2	2	*2*	[W]<>----1..n-[P] n>1
10	SP: whole with more than one optional parts.	2	2	0	0	1	1	*2*	[W]<>----0..n-[P] n>0
11	SH: locally shareable part.	2	0	0	0	0	0	*1*	[W]<>-m..n---[P] n>1 n>=m
12	SP, SH: mandatory whole with locally shareable parts. ED, SH: inseparable and locally shareable part.	2	0	0	0	0	0	*1*	[W]<>-1..n----[P] n>1
13	SP, SH: optional whole with locally shareable parts.	2	0	0	0	0	0	*1*	[W]<>-0..n----[P] n>0
∑		**30**	**25**	**18**	**16**	**15**	**13**		

Mark 1 means that we have to use additional constraints (mark 2) as well as special values (mark 3).

Mark 4 means that the design can be used by just creating relvars. Each relvar has by definition one or more candidate keys and can have foreign keys – additional constraints (mark 2) and special values (mark 3) are unnecessary.

In the description of lifetime dependency, we use nine cases proposed by Barber et al. (2003) (see Figure 53) that compare lifetime of the part to the lifetime of the whole.

We do not give marks in case of this characteristic (see Table 18) because cardinality constraints are not specified. These constraints determine possibility of using one or another design (designs 1-6) and necessary additional constraints.

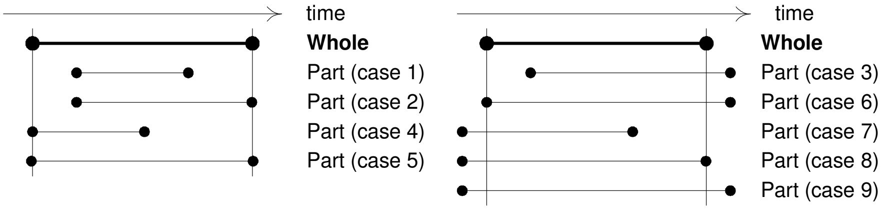


Figure 53 The cases of lifetime dependency (Barber et al. 2003)

If the notation of the cardinality constraint value is n, then we assume that it is some finite number that a designer can specify.

We used the "minus technique" algorithm (Võhandu et al., 2006) for ordering the data table (see Table 18) in order to see typical and fuzzy parts of the data. This algorithm *reorders* rows and columns in a table based on the frequencies of the data values (marks in this case). It also finds groups of the relationship characteristic values that have a similar usability (marks) in terms of the designs (1-6) (see column *Grp* in Table 18).

Next, we give examples of the *integrity constraints* that are necessary in case of the designs (1-6) in the context of the values of the characteristics. Our goal is not to present all the possible constraints that correspond to all the characteristics of the whole-part relationships. Instead, we want to present examples in order to illustrate decisions what we had to make during the creation of Table 18.

Firstly, we investigate the case where the cardinality constraint of the relationship determines that a whole must have at most one part (ID=4 in Table 18). In case of designs 1 and 6, we do not need additional relvar and database constraints besides candidate key and foreign key constraints. In case of design 1, attribute *a* of relvar *Whole* must be the candidate key. Relvar *Part* must have two candidate keys – attribute *c* as well as the foreign key attribute *a*. In case of design 6, attribute *a* of relvar *Whole*, attribute *c* of relvar *Part* and foreign key attribute *a* of relvar *PartOfWhole* must be candidate keys. In case of designs 2-5, attribute *part* of relvar *Whole* can have the values that represent missing information. In addition, in case of designs 4 and 5 we have to limit the amount of tuples that can be part of a value of attribute *part*. We could create database constraint with the following expression (27):

```
IS_EMPTY((SUMMARIZE (Whole UNGROUP part) PER Whole {a}     (27)
          ADD Count AS card) WHERE NOT (card<=1));
```

If the cardinality constraint of the relationship requires that a whole must have exactly one part (ID=8 in Table 18), then attribute *part* cannot have a value that represents missing information in case of designs 2-5. In addition, the previous relvar constraint must have the condition card=1 in case of designs 4 and 5. In case of designs 1 and 6, we have to *additionally* create database constraints with the expressions (28) and (29), respectively:

```
IS_EMPTY(Whole SEMIMINUS Part);                  (28)

IS_EMPTY(Whole SEMIMINUS PartOfWhole);           (29)
```

A SEMIMINUS B is the relational operation, the result of which contains tuples of A that have no corresponding tuple in B (Date 2003). If we want to assign new values to relvars *Whole*, *PartOfWhole* and *Part* in case of constraints (28) and (29), then we have to use multiple assignment operation.

Next, we investigate the case where a part must have exactly one associated whole (ID=7 in Table 18). Attribute *c* must be the candidate key of relvar *PartOfWhole* in case of design 6 and we need similar constraint to the constraint (29) that refers to relvar *Part* instead of relvar *Whole*. In case of designs 2-5, we have to prevent the possibility that the data about the same part is recorded repeatedly – as part of the different tuples in relation *Whole*. For example, in case of design 3 we could create database constraint with the following expression (30) in order to assure that the value of attribute *c* is not recorded repeatedly in the relation. Therefore, values of user-visible attribute c must be unique across relation *Whole*. The attribute *c* is the unique identifier attribute of entity type *Part*.

```
IS_EMPTY((SUMMARIZE (Whole UNWRAP part) PER Whole        (30)
   UNWRAP part {c} ADD COUNT AS cnt) WHERE cnt>1);
```

UNWRAP is the relational operator that forms a relation, the heading of which contains attributes that correspond to the attributes in the heading {H} of the tuple type, instead of one attribute with the type TUPLE{H} (Date 2003). In case of design 2, we have to declare that attribute *part* with a scalar type is a candidate key. It is not enough to declare that attribute *part* is a candidate key in case of designs 4 and 5. Two distinct values with the same relation type can contain the same tuple. Therefore, in case of designs 4 and 5 we have to use similar constraint to the previous one (30) where the operator UNWRAP is replaced with the operator UNGROUP. UNGROUP is the relational operator that "unnests" an attribute that has a relation type.

The constraint is even more complex than (30) if we assume that whole-part relationship has the following participation and cardinality constraints: [W]<>-**1..1**-------**0..1**-[P]. Such relationship exists, for example, in case of mandatory wholes with no more than one locally exclusive part. Let us assume that "?" is a special value that represents a missing value in case of integers (see Figure 54) and UNK_INT() is the operator that returns this special value.

Whole

a	b	part
1	2	?, ?
2	4	2, 5
3	5	7, 1
4	9	?, ?

Figure 54 Value of relvar Whole that contains special values

The constraint with the following expression (31) ensures in case of design 3 that data about the same part instance is not recorded across all the values of attribute *part* more than once.

```
IS_EMPTY(((SUMMARIZE Whole UNWRAP part PER Whole          (31)
        UNWRAP part {c} ADD COUNT AS cnt) WHERE
                    c<>UNK_INT()) WHERE cnt>1);
```

The idea of this constraint is to "unwrap" attribute *part* and count how many times each value of attribute *c* participates in the result. The set of *c* values, *except special value returned by UNK_INT()*, that is in the result more than once must be empty.

One characteristic of the whole-part relationships is shareability of parts. Object type can be related through whole-part relationship type to another whole object type in case of globally shareable parts (see Figure 55). We use the same neutral notation (a dotted line diamond) as Barbier et al. (2003) for presenting general whole-part relationship.

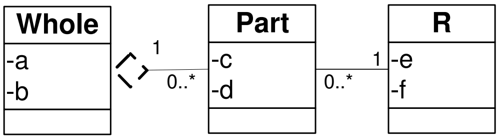


Figure 55 Globally shareable part

It is possible to implement the relationship type between entity types *Part* and *R* in case of designs 1, 3, 4 and 6. In case of designs 1 and 6, we can use foreign key constraints for maintaining the referential integrity rule, but we need special database constraints in case of designs 3 and 4. For example, expression of the constraint in case of design 3 is following:

```
IS_EMPTY((Whole UNWRAP part) SEMIMINUS R);          (32)
```

Attribute *part* of relvar *Whole* has tuple- or relation type in case of designs 3 and 4. We assume that the headings of these types contain the foreign key attributes. In addition, we assume that the candidate key attributes in relvar *R* (that is created based on entity type *R*) have the same names as previously mentioned foreign key attributes. In this case, we do not have to use RENAME operator. The problem with this kind of approach is that we cannot specify easily the compensating action (for example, ON DELETE CASCADE) in order to overcome possible referential integrity violation.

In case of designs 1 and 6, we can use virtual relvars (views), which have attributes with complex types, in order to present data the same way as in case of designs 2-5. The following expression (33) is an example of the relational expression of a virtual relvar that can be used in case of design 1. The virtual relvar with such relational expression has the relation type: RELATION {a INT, b INT, part RELATION {c INT, d INT}}.

"A mandatory FILL clause specifies the contents for non-matching tuples, thus avoiding the need for NULLs" (Voorish, 2005). We assume that "?" is the special value for unknown data in case of type INTEGER.

(Whole LEFT JOIN Part FILL {c "?", d "?"}) GROUP {c, d} AS part; (33)

Date (2003, p. 301) describes *The Principle of Interchangeability* according to which there must be no arbitrary and unnecessary distinctions between real- and virtual relvars. Therefore, an $\text{ORDBMS}_{\text{TTM}}$ allows us to change the values of real- and virtual relvars the same way. For example, if we delete tuple from the value of the virtual relvar (33), then the system must change the values of the underlying real relvars by deleting corresponding tuples from the relations *Whole* and *Part*. This behaviour is needed in case of the inseparable parts.

Next, we draw some conclusions based on Table 18. Designs that have attributes with complex data types in *real relvar*s are unsuitable to use in case of the relationship characteristic values which impose restriction that the cardinality constraint at the relationship end connected to the whole is *bigger than one*. It is traditionally seen as property of the aggregation relationship. In addition, designs 2 and 3 that use an attribute with a tuple type or a user-defined scalar type are not usable if the cardinality constraint at the relationship end connected to the part is *bigger than one*. Designs 2 and 3 are well usable and do not require additional constraints if the multiplicity at the both ends of the relationship is 1..1. Design 6 is usable in case of any characteristic value, but sometimes requires additional constraints. Design 1 is not usable only in case of the relationship characteristic values, which impose restrictions, that the cardinality constraint at the relationship end connected to the whole is *bigger than one*.

We summarize marks by designs (see row ∑ in Table 18). Generally, the bigger the sum is, the smaller are the usage restrictions of the design and the need for the accompanying constraints and special values. As we can see in Table 18, designs 1 and 6 have bigger results than designs 2-5. These values do not mean that it is prohibited to use attributes with complex types in the real relvars. For example, Date (1998, p. 55) presents example where the use of this kind of attribute in a relvar is reasonable (see section 3.4.4). However, these values should help to make a good and reasonable decision.

3.3.3 Advantages of Attributes with Complex Types in Real Relvars

Possible advantage of designs 2-5 (see section 3.3.2.2) is that data about a whole instance and its associated part instances can be accessed by only accessing one relvar (*Whole*). A user has to retrieve only one tuple and has to send only one request to a DBMS instead of many requests. It reduces network load. However, in case of designs 1 and 6, we could use virtual relvars for the same purpose. They provide even more flexibility because the user can decide which relvar to use and how complex tuple to retrieve.

We have to agree with Date (2003) that one advantage of designs 2-5 is that database users do not have to write expressions that contain the outer join operator in order to retrieve data about the whole and part instances together. However, if users ask the values of virtual relvars (see (33)), then they do not have to write outer-join queries themselves.

Encapsulation is one of the secondary characteristics of the whole-part relationships (Barbier et al., 2003) and is rooted in the object-oriented software engineering. In this context, it means that database users have to use special access operators in order to access data about parts. However, tuples in relations are not encapsulated and have user-visible components. Data about parts is encapsulated in case of design 2 and 5 because we need special operators that accompany scalar type ST, in order to access and modify data about parts. Data about parts is not encapsulated in case of designs 1, 3, 4 and 6. One could emulate encapsulation of parts by using user-defined relation-valued operators (RM Prescription 20) (Date and Darwen, 2000). The operator (34) can be used in case of design 1, but similar operator could be used in case of other designs as well.

```
OPERATOR Part(w INT) RETURNS (RELATION {c INT, d INT})        (34)
        RETURN ((Part WHERE a=w) {c, d}); END;
```

The argument of this parameterized operator is a whole instance identifier and it returns relation that contains data about the associated part instances of this whole instance. The idea of using stored parameterized queries is not unique to The Third Manifesto. For example, Levy et al. (1996) propose to use parameterized views.

Designs 2-5 do not necessarily reduce the amount of relvas in a database because we may create additional virtual relvars. They allow us to access directly data about parts and provide better support to ad hoq queries. For example, in case of design 4, we can create the following virtual relvars. A value of the virtual relvar that has expression (35) contains only data about wholes. Its relation type is RELATION {a INT, b INT}.

```
Whole {ALL BUT part};        (35)
```

A value of the virtual relvar that has expression (36) contains data about parts. Its relation type is RELATION {a INT, c INT, d INT}.

Whole UNGROUP part {a, c, d}; (36)

Why do we have to use real relvars that have attributes with complex types if we have to create such virtual relvars?

Designs 2-5 could make it easier to implement some features that are orthogonal to data models. Designs 2-5 make naive implementation of versioning easier. If we modify a tuple, then the versioning system must preserve old version of it. The problem is that even the smallest change causes recording of old version of the entire tuple and therefore data about a whole instance as well as its associated part instances. It causes data redundancy and increases the need for storage space.

Designs 2-5 can make it easier to implement concurrency control by database vendors because it is possible to use existing functionalities of DBMSs. For example, if a DBMS records data about parts and wholes together at the internal level and uses locking, then only one tuple in the implementation of a real relvar (at the internal level) has to be locked in order to lock data about a whole instance and its associated part instances. However, locking is method of concurrency control that belongs to the implementation level of the system. In addition, not all DBMSs are recording data about the whole and part instances together at the internal level. For example, a table with a column that has a table type in Oracle (Oracle, 2005) is recorded internally as two separate tables (Kyte, 2001).

There are opinions that the use of attributes with the complex data types in real relvars helps to improve performance because data about an object is in this case not fragmented (see the problem "Performance problems due to fragmentation" in Table 15). However, storage of the data that is presented in relations does not have to reflect the structure of relvars. If there are two separate real relvars at the conceptual level, then at the internal level their data can be recorded together as if there is one relvar. This approach has already been used in the real systems. DBMS Oracle (Oracle, 2005) permits creation of indexed- or hash clusters for this purpose. Skatulla and Dorendorf (2003) investigate how to optimize storage structures of complex types in ORDBMSs. They propose Physical Representation Definition Language (PRDL) and implement a prototype system in Oracle9i. A PRDL-specification is a strictly separated part of a DDL statement. Skatulla and Dorendorf (2003) acknowledge that: "fully separated definition with adequate references would be possible." This is in line with the work of Date and Darwen (2000) who propose "storage structure definition language" that is distinct from the data definition and data manipulation languages.

In addition, if we want to keep logical distinction of model and implementation, then "easy implementation" should not be argument that forms and reshapes the model. Date (2003, p. 301) describes *The Principle of Interchangeability* according to which there must be no arbitrary and unnecessary distinctions between real- and virtual relvars. Therefore, an $\text{ORDBMS}_{\text{TTM}}$ should allow update virtual relvars the same way as real relvars.

Such update propagates to the underlying real relvars of this virtual relvar and causes locking of the relevant tuples that are part of their values.

3.3.4 Disadvantages of Attributes with Complex Types in Real Relvars

Designs 2-5 (see section 3.3.2.2) cause the problem of asymmetry (Date, 1998, p. 53) because we have to access, retrieve and modify data about whole instances and their part instances differently.

Section 3.3.2.3 demonstrates that if we use complex types in real relvars, then we need more complex integrity constraints in order to preserve integrity of data in a database. Queries about part instances will also be more complex and we need additional virtual relvars in order to simplify their writing task. Special values of scalar types are needed in order to deal with the missing data.

It is unreasonable to use designs 2-5 in case of some cardinality constraints of the whole-part relationship (see Table 18). For example, we could record data about the same part instance repeatedly if the cardinality constraint in the relationship end connected to the whole is bigger than one. However, it would cause data duplication and update anomalies.

Designs 2-5 make it more difficult to discover data redundancy across different relvars. For example, we could create real relvars with the following types:

- RELATION***{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}***
- RELATION{contractno CONTRNO_TYPE, creation_time TIME_TYPE, supervisor TUPLE ***{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE }***}.

Values of both these relvars can contain data about the same employee. The principle of orthogonal database design helps to discover data redundancy across different relvars. The extended version of the principle that takes into account the use of complex data types in real relvars (designs 2-5) (Eessaar, 2006b) is more complicated than the original principle (Date and McGoveran, 1994) that does not take it into account. See the next section for more thorough discussion of this principle. See also section 3.4.4 that provides additional illustrations of problems that occur if we use relation-valued attributes in a real relvar.

The use of complex data types does not have such clear advantages that one could conclude based on the existing research like (Zhang et al., 2001) and (Pardede et al., 2005). It does not remove the complexity, but repositions it within the system.

3.4 Additional Guidelines for Database Design

Bigger selection of data types means that a database designer has more design options but also more possibilities to come up with a bad design. For example, an entity type in a conceptual data model could be implemented as a separate real relvar or in some cases as an attribute of a real relvar. This attribute has a scalar or non-scalar (tuple- or relation type) type. Soutou (2001) presents a simple conceptual data model (meaning 2) with six entity types and four relationship types. He offers 384 different designs in order to implement such data model in an ORDBMS$_{SQL}$ database. Many of these designs use collections and pointers. He does not take into account solutions that have multiple nesting levels.

We need guidelines that help to avoid bad design decisions. One obvious guideline is that we should design our database in a way that prevents redundant data in it. Normalization and dependency theory deals with the formal guidelines about how to eliminate data redundancy within the value of each relvar. Vincent (1998) offers formal definition of the redundancy and explains informally that: "an occurrence of a data value in a relation is semantically redundant if it is implied by the other data values in the relation and the constraints which apply to the relation." Mok et al. (1996) also define redundancy and take into account nested relations. However, both these definitions consider redundancy within the value of *one* relvar.

What about data redundancy *across* different relvars?

A relevant guideline that helps to prevent data redundancy *across* different relvars is The *Principle of Orthogonal Design (POOD)* (Date and McGoveran, 1994). General idea of POOD is "no cross-table duplication, which means no two tables should have rows representing the same entity (or propositions about the same entity)"(Pascal et al., 2005). If this principle is violated, then the same data about the same entity could be recorded as part of the value of more than one real relvar and it means data redundancy. It causes update anomalies and therefore possible violations of data integrity and makes more difficult to construct queries and understand data in a database. However, the original version of this principle does not take into account that real relvars could have attributes that have complex types.

3.4.1 The Principle of Orthogonal Design

In this section, we describe original version of The *Principle of Orthogonal Design (POOD).*

Connolly and Begg (2002) present guidelines for the logical design of a relational database. They describe how to create tables in a database if a conceptual data model contains generalization relationships between the entity types. If the participation constraint is "Mandatory" and disjointness constraint is "Disjoint" (see Figure 56), then they suggest to create a table (relvar) for each combined entity sybtype/supertype. The integrity constraints should ensure that if we add a tuple to one relation, then we cannot add a tuple with the same data to another relation. These tables violate POOD without such constraint. Unfortunately, Connolly and Begg (2002) do not state that this kind of constraint is necessary.

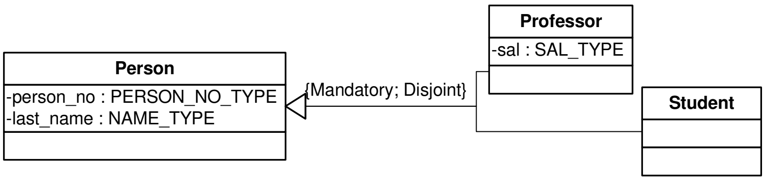


Figure 56 Example of generalization relationship with associated constraints

In this case, we have to create the following real relvars ((37) and (38)) according to the guideline of Connolly and Begg (2002).

VAR Professor BASE RELATION {person_no PERSON_NO_TYPE, last_name NAME_TYPE, sal SAL_TYPE} KEY {person_no}; (37)

VAR Student BASE RELATION {person_no PERSON_NO_TYPE, last_name NAME_TYPE} KEY {person_no}; (38)

The Principle of Orthogonal Design: "Let A and B be distinct base relvars. Then there must not exist nonloss decompositions of A and B into A1, A2, ..., Am and B1, B2, ..., Bn (respectively) such that some projection Ai in the set A1, A2, ..., Am and some projection Bj in the set B1, B2, ..., Bn have overlapping meanings." (Date, 2003, p. 397)

Nonloss decomposition of relvar R into projections R1, R2, ..., Rn means that R is equal to the join of R1, R2, ..., Rn and no projection is redundant (if any of these projections is missing, then the join of others is not equal to R). (Date, 2003, p. 355) For example, if we have the following projections of relvar *Professor* {person_no}, {person_no, last_name}, {person_no, sal}, then the first one ({person_no}) is not needed in order to restore relvar *Professor*. The nonloss decompositions of relvar *Professor* are:

- {person_no, last_name}, {person_no, sal}
- {person_no, last_name, sal}

What is an overlapping meaning of the relvars? A DBMS does not "understand" the meaning of a relvar the same way as humans do – based on the name of the relvar and the names of its attributes (Date and McGoveran, 1994). Even humans may have difficult to understand it if the names are not properly selected. Operations with the relvars do not depend on their names. However, a DBMS knows the integrity constraints that are associated with a relvar. The *relvar predicate* for relvar R "is the logical AND or conjunction of the constraints that apply to - in other words, mention relvar R" (Date, 2003, p. 259). Let us assume that R1 and R2 are two relvars, with associated relvar predicates R1A and R1B, respectively. The meanings of R1 and R2 are said to overlap if and only if it is possible to construct some tuple t so that R1A(t) and R1B(t) are both true (Date and McGoveran, 1994). In other words, if relvars R1 and R2 have overlapping meanings, then tuple t could be part of the value of both these relvars.

Relvar (let us call it *Professor''*) that is a possible projection of relvar *Professor* has the relation type:

- RELATION {person_no PERSON_NO_TYPE, last_name NAME_TYPE}

Relvar (let us call it *Student''*) that is a possible projection of relvar *Student* has the relation type:

- RELATION {person_no PERSON_NO_TYPE, last_name NAME_TYPE}

Both these relvars have the following predicate (39):

```
p.person_no PERSON_NO_TYPE AND                    (39)
  p.last_name NAME_TYPE AND
(IF p.person_no=r.person_no THEN p.last_name=r.last_name)
```

The first two rows of the predicate show that both relvars have two attributes and these attributes have the same types. The last row of this predicate indicates that both these relvars have one candidate key and a candidate key attribute has in both relvars the same type. Relations *Student''* and *Professor''* can both contain a tuple with the same data even if the names of attributes are different in different relvars. We conclude that in this case, relvars *Professor* and *Student* have overlapping meanings and they do not follow POOD guideline. It is possible to record same data about the same person by using both these relvars. We need the following constraint (40) in order to prevent that:

```
CONSTRAINT C IS_EMPTY (Professor JOIN Student);     (40)
```

Join operation uses the attributes in both relvars that have same name and type– *person_no* and *last_name*. If the names in the relvars are different, then we have to use additionally RENAME operator. In this case relvar *Professor''* has the predicate (41):

```
p.person_no PERSON_NO_TYPE AND                    (41)
  p.last_name NAME_TYPE AND
(IF p.person_no=r.person_no THEN p.last_name=r.last_name) AND
IS_EMPTY (Professor'' SEMIJOIN Student)
```

and relvar *Student''* has the predicate (42):

```
p.person_no PERSON_NO_TYPE AND                    (42)
  p.last_name NAME_TYPE AND
(IF p.person_no=r.person_no THEN p.last_name=r.last_name) AND
IS_EMPTY (Student'' SEMIJOIN Professor)
```

Date (2003, p. 196) writes: "the semijoin of a with b is the join of a and b, projected over the attributes of a." These predicates are different and therefore relvars *Professor* and *Student* have no overlapping meanings and they follow POOD guideline.

Let us assume that we create a real relvar based on each entity type that is in Figure 57. All these relvars have two attributes – one of them has built-in type

INT and another has built-in type CHAR. An attribute that has type INT is a candidate key.

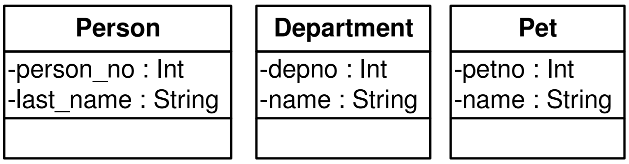


Figure 57 Example of difficulties in using POOD

We want to record different data by using the different relvars. However, a DBMS does not know that. For example, tuple with the values <1, 'test'> could be part of the value of all these relvars. If we apply POOD, then we find that these relvars violate it.

If we take the position that design in Figure 57 is satisfactory and use only built-in "simple" types (INT, CHAR, DATE etc.) in a database, then we cannot effectively automate the checking of POOD. The software would find a lot of pairs of projections that formally have overlapping meaning. However, without manual intervention of a user, the system does not know for sure, what was the exact intention of designer and whether these relvars suppose to help to record the same data or not. In this case, this principle is only intuitive guideline to the database designer, who manually inspects the data model (meaning 2).

It is an argument in support of using user-defined types in a database. For example, attributes in relvar *Person* could have types PERSON_NO (base type INTEGER) and PERSON_NAME (base type CHAR) and attributes in relvar *Department* could have types DEPT_NO (base type INTEGER) and DEPT_NAME (base type CHAR). The possible representations of these types have only one component.

Albrecht et al. (1998) describe a database design method that allows us to derive database structure and constraints from a natural language description. A database designer has to enter real world data in order to find candidates for valid and not-valid semantic constraints. According to "Heuristic Rules to Search for Analogue Attributes" (Albrecht et al., 1998), it is possible to find the cases when different tables contain the same data by finding attributes that have the same meaning (they are called "analoga"). "All attributes of a database having the same type and similar length are checked for being analoga." (Albrecht et al., 1998) Examples of the *heuristic* rules that help to determine whether the attributes have the same meaning: these attributes have same or similar (synonyms) attribute names; the same values in the sample data; the same or similar number of distinct possible values. Differences with POOD are that this approach requires sample data and uses the names of attributes. However, it is possible that two attributes that have the same meaning have names that are not similar. It is possible that the registered sample values of two attributes that have the same meaning are different. In addition, this approach does not seem to take into account the use of complex types.

The use of complex data types opens up new opportunities to the database designers.

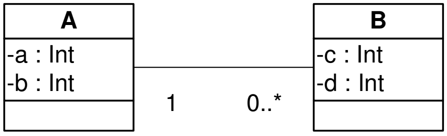


Figure 58 Example of one-to-many relationship

Next, we present some possible designs of *real relvar*s that we could create based on this model (see Figure 58). We also show names of the relvars (A or B) (INT is abbreviation of INTEGER).

Design 1: A: RELATION {a INT, b INT}
B: RELATION {a INT, c INT, d INT}

Design 2: A: RELATION {a INT, b INT, B RELATION {c INT, d INT}}

Design 3: A: RELATION {a INT, b INT, B RELATION {B B_TYPE}}

B_TYPE is a scalar type that is created based on entity type *B*. Its possible representation contains two components that correspond to attributes *c* and *d*.

Design 4: B: RELATION {A TUPLE {a INT, b INT}, c INT, d INT}

Design 5: B: RELATION {A A_TYPE, c INT, d INT}

A_TYPE is a scalar type that is created based on entity type *A*. Its possible representation contains components that correspond to attributes *a* and *b*.

Design 6: A: RELATION {a INT, b INT}
B: RELATION {c INT, d INT}
AB: RELATION {a INT, b INT}

We note that in this case Soutou (2001) proposes 12 solutions based on OR_{SQL}. One could say that the use of collection types is reasonable only in case of whole-part relationships. However, Soutou (2001) presents different designs and some of them use collection types in case of one-to-many relationships (*which are not necessary whole-part relationships*). Pardede et al. (2004) also suggest to use collection types in case of one-to-many relationships.

Data that corresponds to entity type *A* is duplicated within the value of relvar *B* in case of designs 4 and 5 if an entity with the type *A* is associated with more than one entity with type *B*. In case of designs 2 and 3, we need constraints, which enforce the rule that an entity with type *B* is associated with only one entity with type *A*. Otherwise we could register data about the same entity with type *B* within more than one tuple of relation *A*. This means that entity types *A* and *B* are associated with many-to-many relationship.

Now let us assume that requirements to the database evolve. The database must allow us to register data about entity types *C* and *D* (see Figure 59).

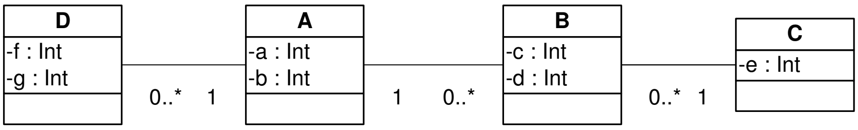


Figure 59 Multiple occurrences of one-to-many relationship

Next, we present two *examples* of names and types of possible real relvars that implement this model. The first example is:

- B: RELATION {**A TUPLE {a INT, b INT}**, C TUPLE {e INT}, c INT, d INT}
- D: RELATION {**A TUPLE {a INT, b INT}**, f INT, g INT}

In this case, data about an entity that has type *A* is duplicated in the values of different relvars if it is associated with an entity that has type *B* as well as with an entity that has type *D*. Another example is:

- A: RELATION {a INT, b INT, **B RELATION {c INT, d INT},** D RELATION {f INT, g INT}}
- C: RELATION {e INT, **B RELATION {c INT, d INT}**}

In this case, data about an entity that has type *B* is duplicated in the values of different relvars if it is associated with an entity that has type *A* as well as with an entity that has type *C*.

Now we present the motivating example, which shows that the original principle of orthogonal design does not take into account the use of complex data types. Figure 60 presents conceptual data model, which shows that an employee can be supervisor of orders as well as contracts.

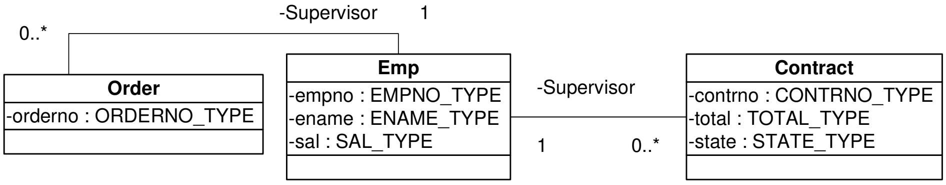


Figure 60 Conceptual data model with entity types Emp, Contract and Order

One could come up with the following database design (we present names and types of the *real relvar*s):

- *Emp*: RELATION {**empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE**, orders RELATION {orderno ORDERNO_TYPE}}

The candidate key of relvar *Emp* is attribute *empno*.

- *Contract*: RELATION {supervisor TUPLE{**empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE**}, contrno CONTRNO_TYPE, total TOTAL_TYPE, state STATE_TYPE}

The candidate key of relvar *Contract* is attribute *contrno*.

Relvar *Emp* is created based on design 2 and relvar *Contract* is created based on design 4. The following expressions show how to create relvar *Contract* (43) and assign a new value to it (44). "sal(1000)" is an example of invocation of scalar selector operator.

VAR Contract BASE RELATION {supervisor TUPLE{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}, contrno CONTRNO_TYPE, total TOTAL_TYPE, state STATE_TYPE} KEY {contrno}; (43)

INSERT Contract RELATION {TUPLE {supervisor TUPLE {empno empno(1), ename ename('JOHN'), sal sal(1000)}, contrno contrno(1), total total(50000), state state(10)}}; (44)

Examples of possible values of relvars *Emp* and *Contract* can be seen in Figure 61.

Emp

			order
empno	ename	sal	orderno
1	JOHN	1000	1
2	BOB	1500	9 14
3	ANN	2000	2
4	LISA	2500	21

Contract

supervisor					
empno	ename	sal	contrno	total	state
1	JOHN	1000	1	50000	10
2	BOB	1500	2	40000	15
1	JOHN	1000	3	25000	10
3	ANN	2000	4	100000	20

Figure 61 Examples of values of relvas Emp and Contract

Data about the employees is duplicated in the values of the different relvars. Intuitively this kind of database design does not seem right. Can we show it by using POOD? One possible projection of relvar *Emp* has the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}

One possible projection of relvar *Contract* has the following type:

- RELATION {emp TUPLE{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}, contrno CONTRNO_TYPE}

These relvars are not *isomorphic*. Two tables A and B are isomorphic "if and only if there exists an one-to-one correspondence between the columns of A and the columns of B, say A1:B1,..., An:Bn, such that in each pair of columns Ai:Bi (i = 1, ..., n) the two columns are defined on the same domain." (Date and McGoveran, 1994) One of the relvars has three attributes and another has two attributes. The types (domains) of their attributes are also different. Date and McGoveran (1994) write: "Two tables cannot possibly have overlapping meanings if they are not isomorphic." Therefore, original version of POOD is insufficient in this case.

3.4.2 The Extended Principle of Orthogonal Database Design

In this section, we present the extended version of POOD and examples of its usage.

The Extended Principle: "The type is "complex type" if it is: (a) a relation type, (b) a tuple type, (c) a scalar type where the possible representation has more than one component, (d) a scalar type where the possible representation has one component but this component has one of the types (a)-(c). Let A and B be distinct real relvars. Let A' and B' be distinct virtual relvars where the expressions of A' and B' "flatten" the structure of A and B, respectively. It means that the headings of the relation types of virtual relvars A' and B' cannot contain an attribute with the declared type being "complex type". Then there must not exist nonloss decompositions of A' and B' into A'1, A'2, ..., A'm and B'1, B'2, ..., B'n (respectively) such that some projection A'i in the set A'1, A'2, ..., A'm and some projection B'j in the set B'1, B'2, ..., B'n have overlapping meanings."

How to construct virtual relvars A' and B'?

- If relvar *R* has attribute *t*, which has a tuple type, then the following relational expression unwraps this attribute: R UNWRAP t.
- If relvar *R* has attribute *t*, which has a relation type, then the following relational expression unnests this attribute: R UNGROUP t.
- If relvar *R* has attribute *t*, which has a scalar type ST and this scalar type has a possible representation with the components c1,...,cn, then the following relational expression exposes these components and removes attribute t from the result: (EXTEND R ADD (THE_c1 (t) AS c1, THE_c2 (t) AS c2, ..., THE_cn (t) AS cn)) {ALL BUT t}

The reader must bear in mind that real relvars could have multiple levels of nesting and real relvars could have more than one attribute that has a complex type. For example, an attribute in the heading of a tuple type can have a relation type or an attribute in the heading of a relation type could have a user-defined scalar type, the component of possible representation of which has again a relation type. The relational expressions of the virtual relvars must take it into account.

3.4.2.1 Discussion and Examples

Let us continue with the motivating example that is at the end of section 3.4.1. Based on relvar *Emp,* we can create a virtual relvar with the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, orderno ORDERNO_TYPE}

This virtual relvar has the following expression (45):

Emp UNGROUP orderno; (45)

Based on relvar *Contract,* we can create a virtual relvar with the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, contrno CONTRNO_TYPE, total TOTAL_TYPE, state STATE_TYPE}

This virtual relvar has the following expression (46):

Contract UNWRAP emp; (46)

One possible projection of both these virtual relvars has the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}

The candidate key of this relvar is attribute *empno*. These projections have overlapping meanings because they both have the same predicate (47):

Therefore design of relvars *Emp* and *Contract* is not correct in terms of the extended version of POOD.

e.empno EMPNO_TYPE AND (47)

e.ename ENAME_TYPE AND

e.sal SAL_TYPE AND

(IF e.empno=f.empno THEN e.ename=f.ename AND e.sal=f.sal)

Let us see another example, which show that existing design guidelines do not always take POOD into account. We use entity types *Emp* and *Contract* from Figure 60 as an example. Next, we present the names and types of real relvars based on a possible database design in case of one-to-many relationship (Soutou, 2001):

- Emp: RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, contracts RELATION {**contrno CONTRNO_TYPE**}}
- Contract: RELATION {**contrno CONTRNO_TYPE**, empno EMPNO_TYPE, total TOTAL_TYPE, state STATE_TYPE}

As you can see, contract numbers are duplicated in the different relvars. We can create the virtual relvar with the following type based on relvar *Emp*.

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, contrno CONTRNO_TYPE}

This virtual relvar has the following expression (48):

Emp UNGROUP contracts; (48)

We can create the virtual relvar with the following type based on relvar *Contract*.

- RELATION {contrno CONTRNO_TYPE, empno EMPNO_TYPE, total TOTAL_TYPE, state STATE_TYPE}

This virtual relvar has the following expression (49):

Contract; (49)

One possible projection of both these virtual relvars has the following type:

- RELATION {empno EMPNO_TYPE, contrno CONTRNO_TYPE}

The candidate key of this relvar is attribute *contrno*. These projections have overlapping meanings because they both have the same predicate (50):

e.empno EMPNO_TYPE AND (50)

e.contrno CONTRNO_TYPE AND

(IF e.contrno=f.contrno THEN e.empno=f.empno)

Therefore design of relvars *Emp* and *Contract* is not correct in terms of the extended version of POOD.

Predicate of a relvar is the conjunction of the constraints that apply to this relvar. A human user found the predicates in the previous examples. It would be very useful if a DBMS or some separate tool would be able to determine the predicates of virtual relvars and check whether a database follows POOD guideline. The Third Manifesto states: "the constraints that apply to the result of evaluating an arbitrary relational expression shall be well defined and known to both the system and the user." (Date and Darwen, 2006)

Next, we present some examples about how it is possible to determine the predicate of a virtual relvar.

An attribute constraint (like "empno EMPNO_TYPE") determines the set of possible values that attribute could have (Date, 2003) by specifying the type of the attribute. Some attributes of a virtual relvar and their types could be derived from the headings of the tuple type, relation type or from the components of possible representation of the scalar types.

Let us consider key constraint as an example of database constraint. Expression "(IF e.empno= f.empno THEN e.ename=f.ename AND e.sal=f.sal)" states that attribute *empno* is a candidate key. Date and Darwen (2000) suggest that a DBMS should be able to determine candidate keys of virtual relvars if it knows about candidate keys in real relvars. They acknowledge the possibility that implementations might find proper *superkeys* rather than true candidate keys or might not discover some candidate keys at all. Possible ways how a DBMS could find the key constraints of a virtual relvar:

- By using the input of a user who explicitly specifies the key constraints of the virtual relvar.
- By deducing the information about the keys from the existing constraints to the real relvars. For example, Date and Darwen (1992, p. 133-154) explain how to find functional dependencies in derived relations that are formed by executing some relational expression.
- By analysing the existing value of a relvar. For example, algorithm TANE (Huhtala et al., 1999) finds functional dependencies in a relvar by analysing its value.

A DBMS must have as much as possible information about the data that is in a database in order to fulfil this task. We can give to the DBMS information about nature of the data by creating constraints.

Constraint that helps to enforce rule 1 (see section 3.4.3) determines a candidate key (attributes bi, bj, ..., bk) of a virtual relation that "unnests" an attribute with a relation type. Constraint that helps to enforce rule 2 (see section 3.4.3) determines a candidate key (attributes bi, bj, ..., bk) of a virtual relation that "unwraps" an attribute with a tuple type.

3.4.3 Heuristic Rules for Reducing Data Redundancy within the Value of One Real Relvar

In this section, we present two *heuristic* rules that help to prevent data redundancy within the value of one real relvar. These rules take into account the fact that a database designer can use relvar attributes that have complex types. These rules should be seen as guidelines but not as law. The word "heuristic" means that these rules are often, but not always, usable. For example, these rules are not mandatory if heading {H} of a tuple- or relation type contains one attribute that has a scalar type, the each possible representation of which has only one component.

Rule 1: Let there be real relvar R having attributes a1,..., an and relation-valued attribute r with relation type RT that has heading {H}. Let us assume that if we could have relvar R' with type RT, then the set of attributes bi, bj, ..., bk that are a subset of attributes b1,..., bm in {H} would be a candidate key of relvar R'. Then each possible value of relvar R must satisfy the constraint that has the following expression (51):

```
IS_EMPTY((SUMMARIZE R UNGROUP r PER R UNGROUP r          (51)
    {bi, bj, ...,bk} ADD COUNT AS card) WHERE card>1)
```

Loosely speaking, attributes bi, bj, ...,bk should also be involved in a candidate key of the relation where the relation-valued attribute is "flattened" by using UNGROUP operator.

Rule 2: Let there be real relvar R having attributes a1,..., an and tuple-valued attribute t with tuple type TT that has heading {H}. Let us assume that if we could have relvar R' with the type with heading {H}, then the set of attributes bi, bj, ..., bk that are a subset of attributes b1,..., bm in {H} would be a candidate key of relvar R'. Then each possible value of relvar R must satisfy the constraint that has the following expression (52):

```
IS_EMPY((SUMMARIZE R UNWRAP t PER R UNWRAP t {bi,         (52)
    bj, ..., bk} ADD COUNT AS card) WHERE card>1)
```

Loosely speaking, attributes bi, bj, ...,bk should also be a candidate key of the relation where the tuple-valued attribute is "flattened" by using UNWRAP operator.

These rules can be enforced as database constraints. If the creation of these constraints is unacceptable, then a designer has to seriously consider, whether the database design has to be changed or whether the existing design is a special case (see section 3.4.3.1) that does not need changes.

If we can enforce these constraints in case of attribute *t* of real relvar R, then we could also state that attribute *t* is a candidate key of relvar R. However, if we just state that attribute *t* is a candidate key, then a DBMS may have incomplete information about constraints. For example, without constraint (51) a DBMS will not have information that contract number (attribute *contrno*) is unique identifier of contracts (see Figure 63).

3.4.3.1 Discussion and Examples

Next, we will present an example about the use of the rule 1.

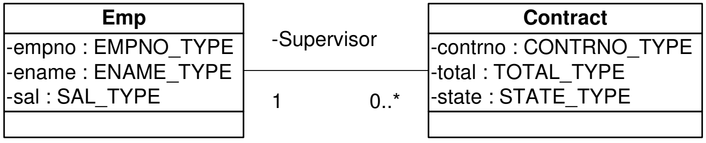


Figure 62 Conceptual data model with entity types Emp and Contract

Figure 63 presents possible value of relvar *Emp* that is created based on the conceptual data model (see Figure 62) and follows design 2 (see section 3.4.1). This relvar has the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, **contracts RELATION{contrno CONTRNO_TYPE, total TOTAL_TYPE, state STATE_TYPE }**}

Attribute *empno* is the candidate key of this relvar. *Contrno* is used in order to uniquely identify contracts. Result of the SUMMARIZE operation (see Figure 63) shows that data about the contract with the *contrno* 1 is duplicated. It will cause update anomalies. The constraint for avoiding data redundancy according to rule 1 is (53):

CONSTRAINT C_emp_contract (IS_EMPTY((SUMMARIZE Emp (53)
UNGROUP contracts PER Emp UNGROUP contracts {contrno}
ADD COUNT AS card) WHERE card>1));

Emp

			contracts		
empno	ename	sal	contrno	total	state
1	JOHN	1000	1	50000	10
			3	25000	10
2	BOB	1500	2	40000	15
			1	50000	10
3	ANN	2000	4	100000	20

contrno	card
1	2
2	1
3	1
4	1

Figure 63 Sample value of relvar Emp and result of the SUMMARIZE operation

The similar constraint is also usable in case of design 3. In this case, we need equality comparison operator for comparing values that have a scalar type. The Third Manifesto prescribes that "The equality comparison operator "=" shall be supported for every type." (Date and Darwen, 2000, p. 139) This operator should be created automatically if a new user-defined scalar-type is created. For example, in case of using design 3 we could create relvar *Emp* that has the following type:

- RELATION {empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE, contracts RELATION{contract CONTRACT_TYPE}}

The constraint for avoiding data redundancy according to rule 1 is (54):

CONSTRAINT C_emp_contract (IS_EMPTY((SUMMARIZE Emp (54)
UNGROUP contracts PER Emp UNGROUP contracts {contact}
ADD COUNT AS card) WHERE card>1));

A DBMS must check equality of values with type CONTRACT_TYPE by using equality comparison operator.

Figure 64 presents possible value of relvar *Contract* that is created based on the conceptual data model (see Figure 62) and follows design 4. This relvar has the following type:

- RELATION {**supervisor TUPLE{empno EMPNO_TYPE, ename ENAME_TYPE, sal SAL_TYPE}**, contrno CONTRNO_TYPE, total TOTAL_TYPE, state STATE_TYPE}

Contract

supervisor					
empno	ename	sal	contrno	total	state
1	JOHN	1000	1	50000	10
2	BOB	1500	2	40000	15
1	JOHN	1000	3	25000	10
3	ANN	2000	4	100000	20

Figure 64 Sample value of relvar Contract

Attribute *contrno* is the candidate key of this relvar. *Empno* is used in order to uniquely identify employees. Data about the employee with the *empno* 1 is duplicated.

The constraint for avoiding data redundancy according to rule 2 is (55):

CONSTRAINT C_contract_emp (IS_EMPTY((SUMMARIZE Contract UNWRAP supervisor PER Contract UNWRAP supervisor {empno} ADD COUNT AS card) WHERE card>1)); (55)

Soutou (2001) and Pardede et al. (2004) use collection types in order to implement one-to-many relationships in an ORDBMS$_{SQL}$ database. They do not use this kind of constraints. A reason is that it is not possible to *declare* such constraints in current ORDBMS$_{SQL}$s. Therefore, an application has to enforce these constraints instead of a DBMS.

As we said earlier, rules 1 and 2 are heuristic rules. Now we present *counterexamples* that show that we do not always have to follow these rules.

Each employee has exactly one salary number and name. One could create relvar *Emp* that has an attribute with a tuple type for recording name and salary:

- RELATION {empno EMPNO_TYPE, **emp TUPLE {ename ENAME_TYPE, sal MONEY_TYPE**}}

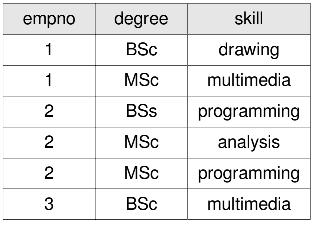
We could have different employees who have the same name and salary. Therefore, the use of the constraint that is proposed in the rule 2 is not suitable. However, such design would make it more difficult to change/retrieve data and enforce constraints and therefore we do not advise to use it.

Let us assume that we have relvar *Emp_background* with the following type:

- RELATION {empno EMPNO_TYPE, degree DEGREE_TYPE, skill SKILL_TYPE}

We want to record only names of degrees and skills. Therefore, both types have one possible representation with one component – name of the degree or skill, respectively. Name has the built-in type CHAR. Let us also assume that relvar *Emp_background* is not in the fourth normal form and contains multi-valued dependencies: *empno→→degree* and *empno→→skill.*

In Figure 65 is a sample value of relvar *Emp_background.* Astrova (2003) presents the algorithm that transforms relational database schema to the object-database schema. This algorithm maps relvar with the multi-valued dependencies to the class that has encapsulated attributes with the set types.



empno	degree	skill
1	BSc	drawing
1	MSc	multimedia
2	BSs	programming
2	MSc	analysis
2	MSc	programming
3	BSc	multimedia

Figure 65 Sample value of the relvar that contains multivalued dependenies

Therefore, by using analogy we could replace relvar *Emp_background* with the relvar that has the following type:

- RELATION {empno EMPNO_TYPE, degrees RELATION {degree DEGREE_TYPE}, skills RELATION {skill SKILL_TYPE}}.

Many employees could have the same degree or skill and no degree and skill is associated with the same employee more than once. We cannot enforce the constraint that corresponds to rule 1. We may choose to ignore it because values of attributes degrees and skills contain minimal amount of repeating data. Only names are repeatedly recorded. The problem is that if names change, then we may have to update more than one tuple.

If we want to start to record textual descriptions about the different types of degrees, then we could add new component *description* with the type CHAR to the possible representation of type DEGREE_TYPE. In this case, name as well as description would be repeatedly recorded in the different tuples and we should consider changing the design so that rule 1 is satisfied.

3.4.4 An Example about the Suitability of Relation-Valued Attribute

Date and Darwen (1992, p. 86) and Date (2003 p. 374-375) present *one* example about the situation where a relation-valued attribute (a relvar attribute that has a relation type) makes sense. Each relvar must have at least one candidate key. Date (2003) proposes that a database *catalog* (see section 1.3.3) must contain a *real relvar Rvk*. Its value "lists the relvars in the database and their candidate keys" (Date, 2003, p. 375). For each candidate key (*ck*) of a relvar (*rvname*), it presents the names of the attributes (*attrname*) that participate in this key. Date (2003) proposes to create real relvar *Rvk* with the following type:

- RELATION {rvname RVNAME_TYPE, ck RELATION {attrname ATTRNAME_TYPE}}

The candidate key of this relvar is the combination {*rvname, ck*}. An advantage of such design is that the candidate key constraint already enforces the rule (R1): A relvar cannot have two distinct candidate keys that involve exactly the same attributes (Date and Darwen, 1992, p. 87). Next, we present *some* examples of possible designs that extend the original design (relvar *Rvk*). Common problems to all these designs:

  - the use of them requires complex query expressions and hence also complex expressions in order to enforce integrity rules or change values of relvars;
  - violations of the extended version of POOD.

If a database user executes a DDL statement, then a DBMS must change values of some of the relvars that belong to the catalog. A relvar can have more than one candidate key. Therefore, a value of relvar *Rvk* can contain more than one tuple with the same relvar name. This redundancy means that if a database designer wants to rename a relvar, then the system must search *Rvk* to find every tuple with the old name (and change it). If the system does not make changes in all the tuples but only in some of them, then the relation presents incorrect data. If we want to avoid this problem and at the same time still use relation-valued attributes, then we must use multiple nesting levels. We could create a real relvar with the following type:

- RELATION {rvname RVNAME_TYPE, cks RELATION {ck RELATION {attrname ATTRNAME_TYPE}}}

The candidate key of this relvar is attribute *rvname*. Attribute *cks* cannot be a candidate key because it is possible that two distinct relvars have exactly the same amount of candidate keys and names of the attributes that participate in these keys are the same. Unfortunately, in this case, the rule R1 is not enforced by the candidate key constraint and we have to enforce it by using a separate database constraint.

Some relvars can have overlapping keys (an attribute is part of more than one key). Data about these attributes is duplicated within the value of relvar *Rvk*. If a database designer changes name of a candidate key attribute, then the system has to find every tuple with the old name (and change it) or otherwise the catalog contains inconsistent data.

Let us continue with the original proposal of Date (2003). A database catalog could contain more information about the attributes. For example, it could contain the information about the type of each attribute. In addition, not all the attributes of a relvar are part of a candidate key. We could create additional real relvar *Non_candidate_key_attribute* with the following type in order to record data about the attributes that are not involved in any candidate key.

- RELATION {rvname RVNAME_TYPE, attrname ATTRNAME_TYPE}

The candidate key of this relvar is the combination {*rvname*, *attrname*}. Without the additional constraint (56), relvars *Rvk* and *Non_candidate_key_attribute* violate the extended version of POOD (see section 3.4.2). It means that without this constraint the database could contain the proposition that an attribute is part of a candidate key as well as not part of any candidate key. Other interpretation of the relvars that do not have this constraint is that it is permitted to create a relvar that has a type where the heading {H} contains two or more attributes that have the same name. However, "No two distinct pairs in {H} shall have the same attribute name." (Date and Darwen, 2006)

CONSTRAINT C_relvar_attribute (IS_EMPTY(Relvar UNGROUP ck JOIN Non_candidate_key_attribute)); (56)

If a database designer wants to drop a candidate key, then the DBMS may have to assign new values to two relvars because data about the non-candidate key attributes must be added to the value of real relvar *Non_candidate_key_attribute*.

Another possibility is to create relvar *Rvk* with the following type:

- RELATION {rvname RVNAME_TYPE, cks RELATION {ck RELATION {attrname ATTRNAME_TYPE}}, non_ck_attributes {attrname ATTRNAME_TYPE}}

Attribute *non_ck_attributes* has a relation type and its value contains the names of non-candidate key attributes of a particular relvar. We also have to create a database constraint (for the same reasons as (56)). However, the necessity of this constraint is not detected by the extended version of POOD because both these attributes (*cks* and *non_ck_attributes*) belong to one real relvar.

If we also want to record data about the types of attributes, then we could come up with the two real relvars that have the following types:

- Relvar *Rvk*: RELATION {rvname RVNAME_TYPE, ck RELATION {attrname ATTRNAME_TYPE, typename TYPENAME_TYPE}}
- Relvar *Non_candidate_key_attribute*: RELATION {rvname RVNAME_TYPE, attrname ATTRNAME_TYPE, typename TYPENAME_TYPE}

Without additional constraint, these relvars also violate the extended version of POOD. Examples of situations that require reading and possibly changing values of both these relvars:

- changing the name of a type;
- changing the type of a candidate key that participates in a foreign key. We have to change the types of associated foreign key attributes as well.

We could also create the relvars:

- Relvar *Rvk*: RELATION {rvname RVNAME_TYPE, ck RELATION {attrname ATTRNAME_TYPE}}
- Relvar *Rvk_attribute*: RELATION {rvname RVNAME_TYPE, attrname ATTRNAME_TYPE, typename TYPENAME_TYPE}

A value of relvar *Rvk* contains names of candidate key attributes and a value of relvar *Rvk_attribute* contains data about all the attributes (including attributes that are involved in candidate keys). These two relvars violate the extended version of POOD and we cannot use a constraint in order to prevent that. Creation, renaming or deletion of an attribute that participates in a candidate requires that the system has to read and change values of both these relvars.

In conclusion, we can say that even the example that should demonstrate the advantages of relation-valued attributes in certain real relvars has not clear advantage if we extend the database.

3.5 View to ORDBMS$_{SQL}$s

In this section, we firstly explore some of the database design options that we cannot use in an ORDBMS$_{TTM}$ database. Secondly, we analyse whether it is possible to use the designs that are usable in the ORDBMS$_{TTM}$ databases in the ORDBMS$_{SQL}$ databases as well.

3.5.1 SQL-specific Solutions

Some of the designs that are usable in case of the OR$_{SQL}$ data model are not usable in case of the OR$_{TTM}$ data model.

Some systems, like for example UML repository (Mahnke and Ritter, 2002) use typed tables and inheritance between the typed tables. It is interesting to note that Definition Schema of SQL (Melton, 2003c) does not implement the generalization relationships by using typed tables.

It is not possible to use typed tables in ORDBMS$_{TTM}$ (see Chapter 1). The authors of The Third Manifesto quite strongly oppose the use of typed tables and inheritance relationship between typed tables. They claim that it adds unnecessary complexity to the data model without actually providing benefits. OR$_{TTM}$ allows us to implement generalization relationship by using virtual relvars (see section 3.3.1). We do not want to repeat the extensive discussion of this topic and interested reader could look, for example, the work of Date (2003, chapter 26) and Date and Darwen (2000, Appendix E).

Bernstein and Dayal (1994), Feldman et al. (2000) and Mahnke and Ritter (2002) advise to record representations of experience elements as a set of CLOBs (Character Large Objects) or BLOBs (Binary Large Objects) in order to keep original storage formats of the different tools. Each experience element is recorded as a big uninterpreted "chunk" without dividing it into fine-grained components. In addition, an experience element is associated with an experience characterization vector (CV) that describes the element.

Barghouti et al. (1996) and Rashid and Lougharn (2003) argue against recording software code in the columns that have CLOB or BLOB data types because a DBMS provides limited means to formulate queries based on this data and modify this data. DBMSs provide only some basic built-in scalar operators and functions (equality comparison, concatenation, substring etc.) for these types. The developers can create user-defined routines (UDRs) or use application code in order to retrieve and modify subcomponents of the artifacts. "The queries become too complex in case of CLOBs and are hard to formulate due to the lack of a formal structure e.g. a relational schema." (Rashid and Lougharn, 2003). Therefore, it is also difficult to create constraints to the columns with these types. The system has to record additional metadata in order to allow us to make queries about the artifacts.

3.5.2 Usability of the Designs and Guidelines in ORDBMS$_{SQL}$s

This section describes problems of OR$_{SQL}$ and ORDBMS$_{SQL}$s. Some problems are caused by the shortcomings of the SQL standard (and OR$_{SQL}$) and some are caused by the incomplete implementation of the standard in the ORDBMS$_{SQL}$s. These problems make it more difficult to use the designs that were proposed in the previous sections (3.1-3.4) in the ORDBMS$_{SQL}$ databases. The appropriate terminology that describes OR$_{SQL}$ based design can be found by using the mapping between the OR$_{SQL}$ and OR$_{TTM}$ data model elements (see section 1.3). We pay attention to two ORDBMS$_{SQL}$s – Oracle10g (Oracle, 2005) and PostgreSQL8.0 (PostgreSQL, 2005).

Current ORDBMS$_{SQL}$s make it difficult to use declarative constraints in a database in order to enforce *well-formedness rules* of the artifacts. Let us assume that names of patterns cannot be empty strings or strings that contain only spaces or underscores. According to OR$_{TTM}$, we can create a new scalar type with the appropriate type constraint. On the other hand, in OR$_{SQL}$ we can choose between the creation of a domain or a type. Domains allow us to specify declarative constraints, but we cannot achieve strong typing because domain is not a data type in OR$_{SQL}$. If the base types of two domains are the same, then we can perform operations (that we have not explicitly specified) with the values that belong to these domains. In addition, Türker and Gertz (2001) evaluate seven DBMSs that use SQL language and write that only one of them allows us to create domains. If we want to define a new type in OR$_{SQL}$, based on a predefined type and achieve strong typing, then a distinct type has to be created. We cannot use constraint *declarations* there. We can use methods of user-defined types in order to implement the constraints by using some *imperative* language.

Date and Darwen (2000) treat concepts "operator" and "function" as synonyms but use the term "operator". SQL (starting from SQL:1999) specifies statement for creating *user-defined functions* but does not specify statement for creating operators. It is not possible to determine more convenient infix, prefix or postfix notation that could be used in order to call this function. On the other hand, SQL dialect of PostgreSQL (PostgreSQL, 2005) and Oracle (Oracle, 2005) allow us to create user-defined operators as well as user-defined functions.

Date et al. (2003, p. 22) introduces the read-only *scalar operator* IS_EMPTY that could be a built-in operator. Currently there is no such built-in operator or function in SQL (Melton, 2003). Argument of IS_EMPTY should be a SELECT statement. It returns a Boolean value *TRUE* if the result of this query contains at least one row and returns *FALSE* if the result of this query contains no rows. Important implementation detail is that it should stop its execution and return the result as soon as first row that belongs to the resultset of the query is found. SQL provides EXISTS predicate, but we cannot write the statement SELECT EXISTS(<<subquery>>). One could say that it is possible to achieve the same results as expected from IS_EMPTY by using the Count aggregate function. The problem is that in this case DBMS has to execute a query, find all the rows that belong to the resultset and count them. It is inefficient because we are interested in existence of at least one row in the resultset but not about the exact amount of rows. It is also possible to implement

IS_EMPTY as a generic user-defined function that uses *dynamic* SQL. However, DBMS vendors probably have better means to optimize this function and prevent its misuse. For example, programs that use dynamic SQL are vulnerable to SQL-injection problem (Boyd and Keromytis, 2004) that could be used in order to attack a database.

Current $\text{ORDBMS}_{\text{SQL}}$s have problems with the *database constraints*. A database constraint can be implemented as a *CHECK constraint*. There exists ORDBMSs like PostgreSQL and Oracle that do not allow us to use *subqueries* in the CHECK constraint although the SQL standard permits that. Database constraints can be implemented using *assertions* that constrain the set of valid values for one or more base tables in SQL (Gulutzan and Pelzer, 1999). For example, the following assertion (57) implements rule R2 from section 3.2.3.1:

```
                    CREATE ASSERTION C_2                    (57)
      CHECK ((SELECT COUNT(*) FROM StartState)<=1);
```

Unfortunately Ceri et al. (2000) note that many $\text{RDBMS}_{\text{SQL}}$s do not support assertion objects. Türker and Gertz (2001) write in the review of integrity constraints in the different DBMS-s: "assertions are in general not available and are unlikely to be offered in the near future."

Alternative method for enforcing constraints in the current $\text{ORDBMS}_{\text{SQL}}$s is to use imperative programs in the *SQL-invoked routines* or *triggers* that were both first time standardized in SQL:1999.

We can create an *SQL-invoked function* that accesses data that is in different tables and returns a scalar value. We can use this function in a CHECK constraint by determining that a value that is returned by this function must satisfy some condition. Currently it is permitted in PostgreSQL but not in Oracle.

If data in a database is changed using SQL-invoked routines, then these routines can enforce the well-formedness rules. In this case, routines must be the only means for modifying data. Systems like UML-repository (Ritter and Steiert, 2000) and business-rule enforcer (Zimbrão et al., 2003) use declarative OCL constraints in order to specify database constraints. They cannot use assertions in order to implement these constraints and have to generate triggers that are written in a proprietary imperative language. Some problems of using triggers: (a) the creation of a trigger does not cause automatic evaluation of the existing data; (b) the SQL standard does not permit to defer execution of the trigger to the end of a transaction; (c) "Semantic query optimization is not possible if the declarative semantics are hidden in triggers." (Cochrane et al., 1996); (d) application generators cannot find easily the data integrity rules. In addition to a DBMS, a generated application could also check whether these rules are satisfied; (e) instead of one declarative constraint, we need many triggers in order react to all the events that can cause invalidation of the constraint.

Let us assume that we have created tables *Whole* and *Part* based on the conceptual data model (see Figure 66) and we want to enforce the structural constraint that each whole instance must be all the time associated with between two and six part instances. We need triggers that react to the insertion of a new row to table *Whole*, insertion of a new row to table *Part*, modification of a part identifier in table *Part* and deletion of a row from table *Part*.

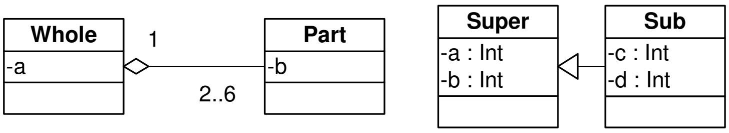


Figure 66 Examples of whole-part and generalization relationships

For example, if a new row is added to table *Whole*, then we have to associate it with the data about the part instance in table *Part*. These operations must be part of one transaction and a DBMS must check the data at the end of it. If any of the checks fails, then the transaction should be rolled back. It can be implemented in PostgreSQL by using *not-standardized* constraint triggers. They allow us to defer execution of the trigger procedure to the end of transaction. In contrast, The Third Manifesto states that constraints must be satisfied at *statement boundaries* and relational language must have *multiple form of the assignment operation* in which several individual assignments to relvars are performed as a single logical operation (Date and Darwen, 2000).

In PostgreSQL, we cannot use a CHECK constraint that uses a user-defined function in order to enforce this constraint. Currently PostgreSQL permits us only to defer checking of foreign key constraints.

Lloyd (1994) shows advantages of the *declarative* programming languages compared to imperative languages which include easier teaching, clearer semantics, improved programmer productivity and better support to meta-programming and parallelism. Cochrane et al. (1996) write: "declarative constraints should be used in lieu of triggers whenever possible." Leff and Rayfield (2006) are also convinced in an advantage of declarative statements and write: "Dramatic improvements in productivity might be achieved if programmers could fully define applications declaratively." They propose Relational Blocks approach that presents business logic in relational algebra and allows us to create an application by using only declarative statements.

Cochrane et al. (1996) thinks that RDBMSs do not support assertions because they are "extremely expensive to support". Maybe it means that an assertion reduces performance of a system? Performance is an issue of the implementation of the data model. If *triggers* have sufficient performance compared to assertions and table CHECK constraints, then creation of a *declarative constraint* at the model level could cause automatic creation of the imperative programs (triggers) at the implementation level. For example, in case of Relational Blocks approach application code in Java is created based on the declarative specification of model, view and controller part of the system.

A DBMS should have information about types of relationships (generalization, whole-part) between entity types (that correspond to tables) in order to be able enforce properties of the relationships and answer to the queries

(Zhang et al., 2001). We have to use virtual relvars in order to implement this kind of relationships in an ORDBMS$_{TTM}$ database (see section 3.3). It must be possible to change data in a database by assigning new value to a virtual relvar. The data change must propagate to the values of all its underlying real relvars.

Generalization relationship is an example of a generic relationship that is often used in the metamodels (see Figure 66). In an ORDBMS$_{TTM}$ database we could create real relvars *Super* and *_Sub* and virtual relvar *Sub* that joins relations *Super* and *_Sub*. If we want to register data about a new entity with the type *Sub*, then we add new tuple to the value of virtual relvar *Sub* and a DBMS adds corresponding new tuples to the values of the real relvars *Super* and *_Sub*.

Is it possible to change data in an ORDBMS$_{SQL}$ database through views? SQL:1992 and earlier standards do not allow us to use joins in the *updateable views* (Date, 2003). Starting from SQL:1999 views defined as one-to-one or one-to-many join of two base tables are updateable (Date, 2003). Unfortunately, there are ORDBMS$_{SQL}$s that do not support the SQL standard in this regard. For example, in PostgreSQL all views are not-updateable without further programming. In Oracle, a DML statement must affect only one underlying table of an updateable join view. In our case, system needs to add data to all the base tables that participate in the join. An alternative is to use not-standardized features like *rules* (PostgreSQL, 2005) or *instead-of triggers* (Oracle, 2005) that are associated with a view in order to achieve its updatability. Creation of these objects requires additional programming.

There are other ways to implement generalization relationship in an ORDBMS$_{SQL}$ database. However, we do not need them (and related complexity in the data model) in the DBMSs that that fully support the relational model.

The SQL standard defines language constructs for creating *subtables and supertables* that seem suitable in order to implement this kind of relationship. Both subtable and supertable must be *typed tables* and a *structured type* on which subtable is defined must be subtype of a structured type on which supertable is defined (Date, 2003). As stated earlier, it is not possible to use declarative constraints in the structured type specification. However, it is possible to use table constraints in order to restrict values in a typed table. PostgreSQL uses non-standard approach according to which a subtable and supertable are not typed tables (PostgreSQL does not support typed tables). PostgreSQL implementation of subtable-supertable feature is immature because the subtable does not inherit all the declarative constraints of the supertable.

Another possibility is to use *triggers* that are associated with the *base tables* (Pokrajac et al., 2004). Let us assume that table *B* is a subtable and *A* is its supertable. For example, we could create delete and update triggers that are associated with *B*. Their task is to delete corresponding row from *A* then row in *B* is deleted and update primary key of *A* then corresponding foreign key is updated in *B*. This approach also requires insertion of new rows into *A* and *B* within one transaction using two different statements.

Users of the current ORDBMS$_{SQL}$s must often manually create additional database objects if some object is created in a database. For example, by default it is not possible to add a key constraint to a column that has a row type in a PostgreSQL database. We have to create first an operator class and a b-tree support function that compares two values that have a row type. Another example is creation of rules or instead-of triggers in order to make a view updateable. Ceri et al. (2000) notes that *handcrafted triggers* are error-prone and triggers should be created by the system. We think that ideally a DBMS should create all these objects automatically. At least the system should allow us to use the schema triggers (like in Oracle) in order to allow us to create the generation program that is executed, when database schema changes. SQL:2003 does not permit such triggers (Melton, 2003).

Standardization of some important features that are required by The Third Manifesto has begun in SQL:1999 or SQL:2003. It takes time before ORDBMS$_{\text{SQL}}$s start to fully implement the standard in this regard.

Recursive queries (introduced in SQL:1999) based on data that represents a graph structure. Information about the associations between artifacts as well as associations between elements of artifacts could be recorded in a repository. Associations and associated elements form a graph structure. Example of a query that is needed then a pattern is modified: Find all patterns in pattern language PL that directly or indirectly depend on pattern P.

Types that are constructed by using the type constructor *ROW* (introduced in SQL:1999) and *Multiset* (introduced in SQL:2003) could be used in the views that allow us to present artifact to a user without fragmentation. It is also possible to use columns with these types in the base tables in order to implement whole-part relationships (see section 3.3.2).

Examples of the constraints that may be necessary in case of using constructed types:

1. A field of a row type or an attribute of a table type must be mandatory (they should not permit NULL's).
2. A row that is part of the value of a row- or a table type must satisfy a predicate.
3. A value with the multiset type cannot contain some element (value) repeatedly.
4. A value with the multiset type has minimum and/or maximum cardinality. It is also possible, that there are gaps in the sets of permitted cardinalities.
5. If a column has a multiset type, then an element (value) can be part of at most one multiset in this column.
6. An attribute of a table type is also a foreign key attribute.
7. Column with a constructed type can have primary key or uniqueness constraint.

A multiset can contain repeating elements. It is not consistent with The Third Manifesto that prohibits duplicate tuples in a body of a relation. Developer who wants to use sets of rows instead of multisets must be continuously aware that *most* (but not all) of SQL statements must explicitly state it. Constraint (3) would be automatically enforced, if we could use type constructor SET. Unfortunately it is not present in SQL:2003. However, we can create a view where duplicate elements that are part of the multiset value are removed by using the function SET.

SQL specifies UNNEST operator that allows us to present elements of a multiset as rows of a virtual table. In theory, we could create declarative constraints to the values of the constructed types that are in some column by using table- or database constraints. These constraints should use, for example, UNNEST operator or MEMBER, SUBMULTISET or SET predicates, introduced in SQL:2003. In practice, we cannot create them because of the limited support to the declarative constraints in the ORDBMS$_{SQL}$s. The same problems are with the database constraints that reference more than one table. It is possible to use triggers for checking these constraints (1-7).

Table-functions (introduced in SQL:2003) allow us to implement parameterized relational operators and return multiset (bag) of rows. They help to implement queries that search artifacts or statistical information from the repository.

Sequence generators (introduced in SQL:2003) generate values for the candidate keys.

Table 19 contains comparison of SQL:2003, The Third Manifesto and two existing systems - Oracle 10g and PostgreSQL 8.0. In Table 19 "Y" means that the feature is supported, "P" means that the feature is "partially supported" according to our evaluation and "N" means that the feature is not supported.

Column *Design* refers to the identifiers of possible designs of whole-part relationship (see section 3.3.2) where such feature is needed.

Sometimes there is no exact correspondence between OR$_{SQL}$ and OR$_{TTM}$ constructs. For example, relations are sets but tables are multisets. Special care is needed in SQL in order to prevent and eliminate duplicated rows. In this case, we use the most similar features in a comparison. Last row of Table 19 summarizes support to the features.

Big amount of "P"-s and "N"-s in the table is consistent with the finding of Barghouti et al. (1996) who evaluated RDBMS$_{SQL}$s and OODBMS$_{SQL}$s and concluded: "there is no single commercial system that completely satisfies the PSEE data management requirements." Unfortunately, the situation has not dramatically changed during the last ten years.

One could say that an ORDBMS$_{SQL}$ allows us to achieve the same results as an ORDBMS$_{TTM}$ if we bear in mind good database design principles and that much of the ORDBMS$_{TTM}$ features are present in the current ORDBMS$_{SQL}$s. Our study shows that the existing ORDBMS$_{SQL}$s do not allow us to implement many aspects of the designs that are presented in the previous sections (see sections 3.1-3.4). Other researches have noticed similar problems. For example, Pardede et al. (2004) try to implement whole-part relationship in an ORDBMS$_{SQL}$ database by using the collection data types. Pardede et al. (2004) write: "At present, we cannot use SQL to embed integrity constraint checking in ORDB collection." Our research result supports this finding and brings attention to the practical need to improve the existing standards and systems.

Table 19 Some features that could help to implement software engineering repository

	The Third Manifesto	Designs	SQL:2003	Oracle 10g	PostgreSQL8.0
1	tuple type generator	3	Y: row type constructor	N	Y
2	relation type generator (relation is a set)	4, 5	P: *multiset* type constructor	P: table type – supports the multiset feature but not the syntax	N
3	user-defined scalar type	2, 5	Y: user-defined type (UDT)	P: supports user-defined structured types (UDST) but not distinct types	N
4	attribute with a complex type can be a key	2, 3, 4, 5	Y: no reference that it cannot be	N	P: needs programming
5	attribute with a complex type can be mandatory	2, 3, 4, 5	Y: no reference that it cannot be	P: yes in case of UDSTs. No in case of the table types	Y
6	complex *declarative* relvar and database constraints	1, 2, 3, 4, 5, 6	Y: CHECK constraint with a subquery	N: CHECK constraint cannot contain a subquery	
			Y: UDF in a check constraint	N	Y
			Y: assertion object	N: not possible to create assertions	
7	*declarative* constraints to the values of the complex types if we use these types as declared types of columns	2, 3, 4, 5	Y: attributes and fields have data types. Possible to declare others constraints by using, for example, UNNEST function.	P: attributes of UDSTs have types. If UDST is a column type, then relvar constraints that its attributes are mandatory.	P: fields have data types. Relvar constraints to the values of row types.
8	it is possible to change value of *multiple* relvars through a virtual relvar (view), the expression of which contains a join	1, 6	P: yes in case of one-to-one join. No in case of one-to-many join - only "many side" is updateable (Date, 2003, p. 322).	P: yes if not-standardized instead-of triggers (Oracle) or rules (PostgreSQL) are programmed.	

	The Third Manifesto	Designs	SQL:2003	Oracle 10g	PostgreSQL8.0
9	automatically generated = and ≠ *operators* for comparing values with a complex type	2, 3, 4, 5	Y	Y	P: there is CREATE OPERATOR statement and possible to program it
10	*built-in* GROUP operator. The result of its invocation is a *set*	1, 6	P: COLLECT function (result is a *multiset)* + SET funct.	P: CAST function + SET function	N
11	*built-in* UNGROUP operator	4, 5	Y: UNNEST function	Y: TABLE function	N
12	*built-in* WRAP operator	1, 6	N	N	N
13	*built-in* UNWRAP operator	3	N	N	N
14	IS_EMPTY *built-in* scalar operator	1, 2, 3, 4, 5, 6	N	N	N
15	SEMIMINUS built-in relational operator and possibility to define new universal relational operators	1, 2, 3, 4, 5, 6	N	N	N
16	possibility to define new scalar *operators*	1, 2, 3, 4, 5, 6	P: Possible to create SQL-invoked routines	Y: There is CREATE OPERATOR statement It is possible to create SQL-invoked routines	
17	possibility to declare in a type creation statement that some special value represents the missing information	1, 2, 3, 4, 5	N	N	N
18	user-defined relational operators	1, 2, 3, 4, 5, 6	P: it is possible to create a table functions, the results of which are *multisets*. Duplicates have to be explicitly removed. In addition, in Oracle a table type and in PostgreSQL a row type has to be created.		
∑	**supports fully / supports partially / doesn't support**		**10 / 5 / 5**	**3 / 7 / 10**	**4 / 5 / 11**

3.6 Summary

Soutou (2001) writes: "Few efforts have been made to offer guidelines to design an OR-database." The general guidelines that we can give based on this chapter are:

- **D**o not use the "Universal Database Design" approach.
- **T**he use of the database design according to which a real relvar has an attribute that has complex type (see section Objectives) does not simplify database design, but repositions the complexity in a database. Prefer the designs that do not use attributes with complex types in real relvars.
- **U**se virtual relvars that have attributes that have complex types in order to implement whole-part relationships.
- **I**f you decide to use real relvars where attributes have complex types, then create constraints that prevent data redundancy.
- **F**ollow the extended Principle of Orthogonal Database Design in order to avoid data redundancy across different real relvars.
- **R**equire good support of virtual relvars (including updatable views) and declarative integrity constraints from a DBMS, which you plan to purchase.

These guidelines are not new. However, we have revised these guidelines in terms of $\mathrm{OR}_{\mathrm{TTM}}$. We offer additional examples that support these guidelines. For example, Date (2003) presents *one* example when to use a relation-valued attribute in a real relvar. We showed that if we extend a database, then the proposed solution leads to data redundancy and requires complex constraints.

The main results of this chapter are:

- Description of different approaches to repository database design. In particular we discussed the schema design and checking of the well-formedness rules. We also proposed an approach for the versioning of artifacts. We presented three approaches for designing a repository database schema – encapsulated artifact types, non-encapsulated artifact element types and encapsulated artifact element types. Loosely speaking, the first one proposes simple real relvars and complex data types. The second one proposes bigger amount and more complex real relvars that use simpler data types. The last one is a combination of previous two. These approaches are not new. Date and Darwen (2000, Appendix C) call the selection between them the "design dilemma". We have to agree with Date and Darwen (2000) that the use of complex data types in a database does not remove the complexity from the system because we still have to create complex virtual relvars and constraints.
- Evaluation of the "Universal Data Model" design approach. This chapter contains more thorough overview of problems of this kind of design than any other paper that we have found.

- Evaluation of the proposed designs that help to preserve semantics of whole-part relationships in a database. We performed the evaluation in terms of the secondary characteristics of whole-part relationships. Such evaluation approach is also one novel result of our work.
- The extended principle of orthogonal database design. It takes into account the use of complex data types in real relvars. This principle helps to avoid data redundancy across different real relvars. We showed that some existing database design guidelines (Soutou, 2001; Connolly and Begg, 2002) do not consider this principle. It leads to data redundancy in a database.
- Two heuristic rules that help to avoid data redundancy within the value of a real relvar that has an attribute with the declared type being a tuple type or a relation type.
- Overview of the shortcomings of OR_{SQL} and $\text{ORDBMS}_{\text{SQL}}$s. These shortcomings make it more difficult to implement the designs that were presented in this chapter.

One of the main advantages of ORDBMSs is the possibility to define new data types. *Main conclusion* of this chapter is that the so-called "traditional" database designs (that do not have attributes with complex data types in real relvars) together with the *special values*, *integrity constraints* and *views* that use complex data types offer actually more freedom and flexibility to designers than the designs that use complex data types in *real relvar*s. This finding supports the guideline that is presented by Date (2003, p. 374). According to this guideline, *real relvar*s *without* relation-valued attributes should be preferred. We also add that *real relvar*s *without* tuple-valued attributes should be preferred.

Some necessary features like views and constraints are not object-oriented. They were required by the SQL standard long before the incorporation of object-oriented features to SQL. Current $\text{ORDBMS}_{\text{SQL}}$s do not implement views and constraints correctly or do not implement them at all. These shortcomings cause criticism towards SQL and relational model. They cause addition of new features to the SQL standard and dialects that would be unnecessary if SQL fully conforms to the relational model.

4 REPOSITORY SYSTEM WITH A FIXED DATABASE SCHEMA

Successful development of an information system takes a lot of effort. Important part of this work is modeling of the system. Common approach is to create different types of models that describe different aspects (dimensions) of a system with a different level of abstraction. An example is the Zachman framework for the information systems architecture (Zachman, 1987). Each model type is used in order to describe one aspect of a system. For example, a system is described in terms of different views in case of the visual modeling language UML. Each view has one or more corresponding diagram types. UML version 1.5 specifies nine types of diagrams (OMG formal/03-03-01) and UML 2.0 specifies thirteen types of diagrams (OMG formal/05-07-04). In different projects, only subsets of these diagram types are used, depending on the goals. However, more than one type of diagrams is needed in order to describe static structure as well as behaviour of a system. For example, Larman (2002) presents a possible structure of a system analysis specification. The specification must contain UML diagrams (visual models) as well as textual models. These models are:

- Use case model (diagrams and textual specification of use-cases).
- Domain model (diagrams and textual specification of conceptual classes and attributes).
- System sequence diagram.
- Contracts of the system operations.

These models together also constitute a model. Final versions of the models that describe a system have to be syntactically and semantically correct, complete and consistent within itself and with each other in order to be most useful. If a model is the combination of diagrams and textual description, then there can be inconsistencies between these components.

Examples of the inconsistencies within one model:

1. Use cases/actors have different names in a diagram and in a text.
2. There are different amount of use cases/actors in a diagram and in a text.
3. A use case is associated with the different actors in a diagram and in a text.

Examples of the inconsistencies across different models:

- Names of the actors are different in a use case model and in the sequence diagrams that describe system operations.
- Names of the elements of a domain model differ from the names that are used in the pre- or post conditions of the contracts of the system operations.

Examples of the completeness problems:

- A domain model is missing.
- A use case diagram is not accompanied with a textual specification.

- An operation contract does not have the post-conditions.

Checking of the models in order to find such problems can be at least partially automated by a software system. Models could be inconsistent and incomplete during the development process. It should be possible at any time to get information about these problems. It helps to gradually improve the quality of the models. Pedagogical pattern *Built in Failure* (Eckstein et al., 2006) suggests that teacher should remove the fear of failure as a barrier to learning by making failure a part of the learning process.

The *hypothesis* that must be controlled in the future is that the checking functionality would also change a modeling tool to a valuable learning tool. It is because continuous feedback from the system instead of a teacher reduces fear to make mistakes. This fear hampers the learning process. It also eases the work of a teacher who does not have to check consistency and completeness (CC) problems manually.

However, CASE systems today do not provide enough support for checking consistency between different types of models (Richters and Gogolla, 2000), (Delen et al., 2005) and between evolving versions of models (Straeten et al., 2003). Nentwich et al. (2003) note that in many CASE systems the constraints are hard-coded into the tool and it is not possible to choose to ignore a constraint or delay its checking. Due to the limitations of CASE tools, the constraint checking is often manual work and takes quite a long time and a lot of effort. It is a "daunting tasks beyond anyone's cognitive ability" (Dori, 2002), even if CASE tools provide some support because of big amount of different types of models.

The author of this dissertation teaches database design in a university. A part of the course work is a term project. Students have to create a strategic- and detailed analysis of an information system and a prototype of its software. Current structure of the project documentation has been used during the last four years. For example, 75 projects were presented in the spring of 2005. Teacher reviewed projects together with their authors and pointed to the mistakes. Average length of a review of one project was about 20 minutes. Average interval between first checking of a project by the teacher and acceptance of the project was 4.2 days. Students improved their projects during this period and sometimes they did it repeatedly. Students used word processor in order to write textual models and CASE-tools or diagram editors in order to draw visual models. These systems do not support automatic consistency and completeness checks. The presented projects reflected this situation and contained many such deficiencies. Another problem is that students have difficulties to understand how all these different models are connected with each other. Therefore, a system is needed that gives fast and precise feedback to the students.

Delen et al. (2005) write: "Second, there is a need for a completely Web-based integrated modeling environment for distributed collaborative users." This chapter describes the system analysis environment that allows us to perform strategic- and detailed analysis of a database-centric information system without using complicated visual notations. This system should ease creation of an independent work by the students and correction of it by the teachers. It can also be used in the real-world information system development projects. The system should support methodological framework for the Enterprise Information System (EIS) strategic analysis (Roost et al., 2004). A person with the modeler role will record system specification in a database as an integrated model using one tool. A modeler does not have to create a collection of weakly connected models by using different tools any more. A modeler will use the modeling language that is *simplified* synthesis of the different system specification languages (UML (OMG formal/03-03-01), SMX (Jäderlund, 1981) and OPM (Dori, 2002) among others). The structure of the database will be derived from the fixed metamodel of this language. This system will be more similar to a CASE than to a Meta-CASE tool because of the fixed metamodel. Data about the various aspects of a system will be recorded in a database using a form-based and web-based user interface. We plan to use a DBMS that allows us to use SQL language. We propose queries that find consistency or completeness (CC) problems that are present in a specification. Our approach eliminates problems with the inconsistencies between the diagrammatic and textual representations. Diagrams and textual models are kind of views to the information in the database that could be generated by the system at any time. We do not have yet completely implemented the system but have started to create its prototype (see "Appendix D: The location of prototype system"). We think that such system will help students/teachers in the learning/teaching process. It will also help to improve quality of the result of the real development projects.

4.1 Related Works

UML is nowadays de facto standard for describing the information systems. Problems of consistency between UML diagrams have, for example, been acknowledged by Engels and Groewegen (2000) who describe open issues in the object-oriented development.

A modeler has to learn a lot of different notations, rules and guidelines that are used in the different types of models. UML 1.1 contains 233 discrete concepts (McLeod, 2000). Still he proposes rich visual notation for process models that could replace UML dynamic diagrams. If we study one UML model or other similar model, then we have to constantly look the other models in order to understand it fully. Switching between different pages/files/packages is inconvenient as well as mentally challenging and wearying. "Multiplicity of representational styles impedes communication between modeling professionals and their clients." (Geoffrion, 1989) Visual models are usually created with different CASE or model drawing tools (Rational Rose, ERWin, ArgoUML, Visio, Dia etc.) and are accompanied with textual specifications that are created using a text editor. We have to have this software in our computer if we want to thoroughly study these models or modify them. Modification of one model requires modifications of dependent models as well. Multiplicity of modeling software and files that contain models often causes creation of a model from scratch instead of reusing existing models. Next, we list some approaches that are used in order to achieve correct and consistent models:

1. Formulation of guiding rules that a modeler should follow. For example, Glinz (2000) describes rules that help to minimize inconsistencies between a class model and a use case model.
2. The use of cross-references between different types of models. For example, Glinz (2000) proposes to use references to a class model in scenarios of use cases.
3. The use of specific models that contain cross-references between other models. An example of such model is a CRUD matrix. It presents associations between object types and processes (Brandon, 2002) and helps to check their consistency.
4. The use of systems that evaluate models that are created by using CASE tools.
5. The use of systems that actively assist users of CASE tools and provide intelligent help.
6. The use of modeling notations and systems that use one type of model in order to specify multiple aspects of a system.

A drawback of the approaches 2 and 3 is that without a tool support, references in a model or new kinds of models may themselves have CC problems. CASE systems can have supporting tools that check models or provide active assistance to its users (approach 4 and 5). They may transform diagrams into some other form of representation in order to analyze them. For example, generated description logics statements (Straeten et al., 2003) are analysed by using description logic query tool. Richters and Gogolla (2000) describe a system that translates UML models to statements of UML-based Specification Environment language. The models are then analyzed by simulating that the model elements have instances. Framework xlinkit (Nentwich et al., 2003) that allows us to check consistency of distributed, heterogeneous documents is yet another example of the approach 4. The system allows us to express constraints between documents by using constraint language. This language is based on first order logic. The system also contains a constraint engine, which task is to check these constraints.

An example of the approach 5 is agents based system WayPointer (Racko, 2004). It monitors use case models that are created by using some CASE tool, for completeness, consistency and correctness. It can point to problems and offer recommendations. Another example is system ISEA (Intelligent Software Engineering Advisor) (Virvou and Tourtoglou, 2006), "which is a software tool for constructing UML diagrams and at the same time support the manager and software engineers adaptively." It evaluates user actions and offers feedback according to performance type and personality of users.

Agarwal and Sinha (2003) conclude that developers do not rate any of the UML diagrams as very high in terms of usability. It could be caused by the use of several model types that leads to the inconsistencies between various parts of system specification (Dori, 2002), (Dori et al., 2003). In addition, UML is a *complex* language. Siau and Cao (2003) evaluate the complexity of UML using complexity metrics and write: "Our findings suggest that each diagram in UML is not distinctly more complex than techniques in other modeling methods. But as a whole, UML is very complex-2-11 times more complex than other modeling methods."

People have been aware of "model multiplicity problem" for a long time. A single model-based approach is superior to multiple model approaches for late requirements engineering through implementation according to Paige and Ostroff (2001). Already Jäderlund (1981) describes a methodology for holistic system development that uses so-called system matrices in order to describe a system. A system matrix (SMX) incorporates multiple views of a system. An accompanying software tool provides methods for checking correctness, completeness, and consistency (CCC check) of a system.

More recently, the model multiplicity problem has been addressed by introducing Object-Process Methodology (OPM) (Dori, 2002), (Dori et al., 2003) that is a holistic system modeling, development and evolution approach. OPM uses Object-Process Diagrams (OP diagrams) for graphic specification and Object-Process Language (OPL) for textual specification of a system. OPM uses one integrated type of model in order to describe structural, functional and behavioural aspects of a system (Dori, 2002). CASE tool Object-Process Case Tool (OPCAT) that supports OPM has been developed (Dori et al., 2003).

Another example of a modeling languages that corresponds to the single-model principle is Eiffel (Paige and Ostroff, 2001). Delen et al. (2005) propose system Modelmosaic that allows us to create different types of models. It records models and relationships between elements of different types of models in a single integrated information base. These relationships, that are recorded as business rues, allow us to generate new models from the existing ones.

4.2 Description of the Modeling Language

A model is created by using some language. Specification of a semi-formal language should contain descriptions of the abstract syntax, well-formedness rules and semantics (Greenfield et al., 2004). For example, a metamodel that describes the abstract syntax of UML is presented as a set of class diagrams (OMG formal/03-03-01; OMG formal/05-07-04). Well-formedness rules of UML are expressed using OCL constraints and its semantics are described using free-form text. In this section, we present the metamodel of the language that will be used for specifying information systems in our proposed system. Diagrams that present fragments of the metamodel are accompanied with the free-form textual descriptions that explain some of the underlying concepts. The structure of the database for recording specifications of information systems will be derived from this metamodel. In section 4.3**,** we present queries that help to check well-formedness of the recorded specifications.

Interested parties can participate in the development project of information system in the different roles (see Figure 67).

An information system (IS) is described using three types of subsystems according to the methodological framework for the Enterprise Information System (EIS) strategic analysis (Roost et al., 2004). These types are: areas of competence, functional subsystems and data centric subsystems that are also called registers (see Figure 68). A functional subsystem corresponds to one or more business processes (Roost et al., 2004). "A register is a logical data-centric view of a business object that holds the state and transactions data of the object and provides related recording and query services." (Roost et al., 2004) Administrative subsystems help to perform administrative tasks of the organizations. The examples are subsystems for the management of data about the workers and documents. These kinds of subsystems are part of many different information systems. Business subsystems help to perform specific business tasks of the organizations. These tasks are the reason why this organization is founded. For example, university IS has subsystems for the management of data about students, curriculums and study results.

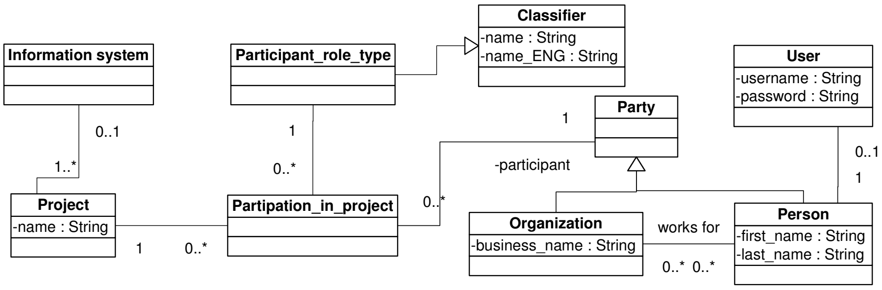


Figure 67 Metamodel of projects and participants

Functional subsystems use the services of one or more registers by reading and modifying data in them. Subjects who have some role in an IS use the services of one or more functional subsystems that belong to the area of competence of their role (Roost et al., 2004).

We provide possibility to specify non-functional requirements of a system (see Figure 69) by using the form that is described in Volere Requirements Model (Robertson and Robertson, 1999).

Functional requirements of an information system can be specified as use cases. Corresponding fragment of the metamodel (see Figure 69) is created based on the guidelines of Larman (2002) and Cockburn (1998). Each use case belongs to some functional subsystem (see Figure 68).

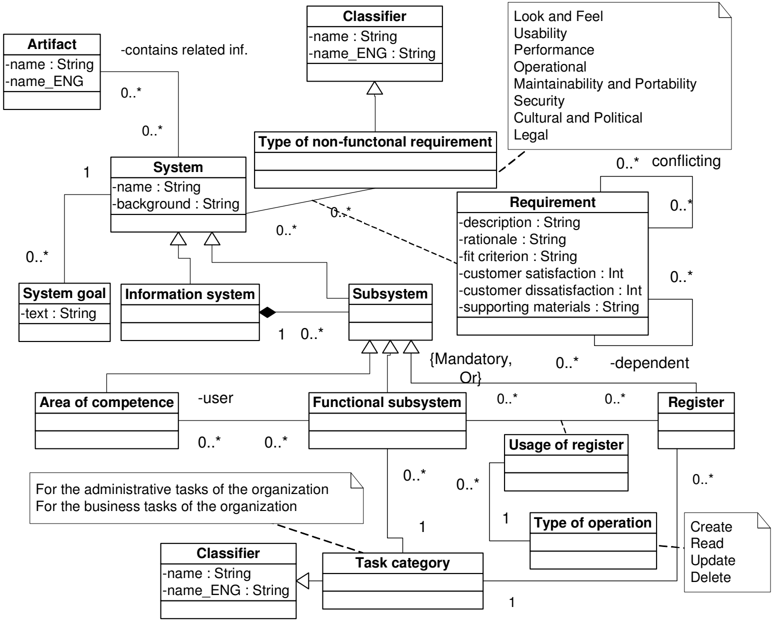


Figure 68 Metamodel of subsystems

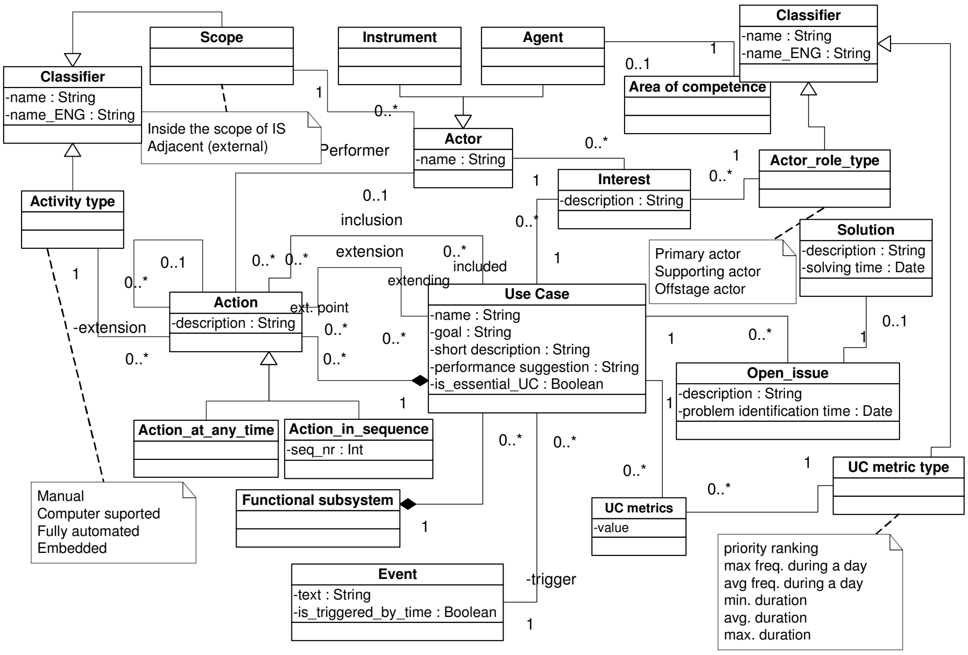


Figure 69 Metamodel of use cases

Each use case describes scenarios that consist of actions. Most actions are performed sequentially. Some actions can be performed at any point of the scenario. Use cases can be related by using either extension or inclusion relationships.

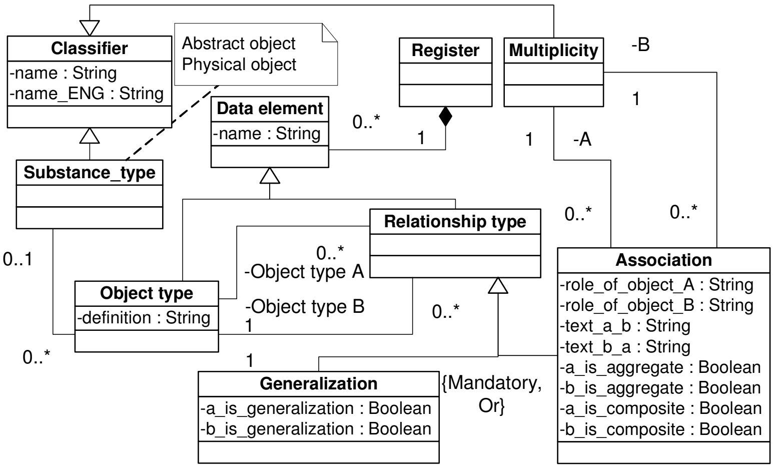


Figure 70 Metamodel of data elements

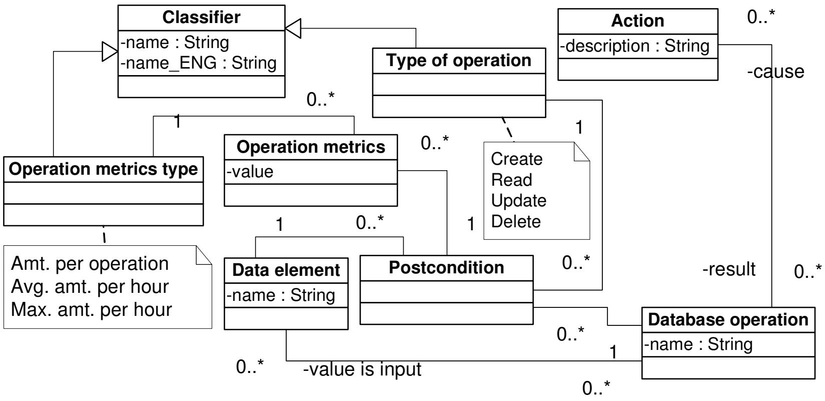
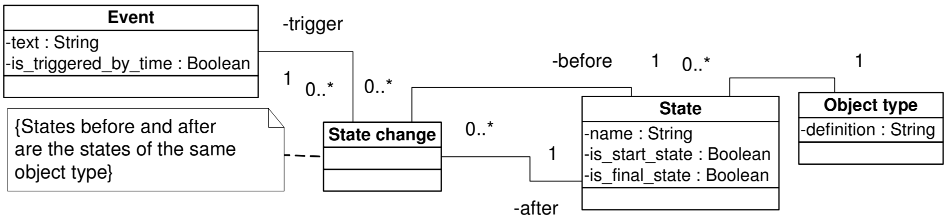


Figure 71 Metamodel of database operations



Event
-text : String
-is_triggered_by_time : Boolean
-trigger
1
0..*
0..*
-before
1
0..*
1
{States before and after are the states of the same object type}
State change
0..*
1
State
-name : String
-is_start_state : Boolean
-is_final_state : Boolean
Object type
-definition : String
-after

Figure 72 Metamodel of state changes

An actor who is either an agent or an instrument may perform the actions. An agent corresponds to an area of competence. Registers are specified in terms of the data elements (object types and relationship types)(see Figure 70) and contracts of the database operations (see Figure 71). An action that is part of a use case can cause execution of a database operation. Result of the operation is described in terms of the post-conditions. A use case is triggered by an event (see Figure 69). An object type may have associated state changes. Each state change is caused by an event (see Figure 72).

4.3 Queries

The system analysis environment must allow users to specify and execute database queries (see Figure 73).

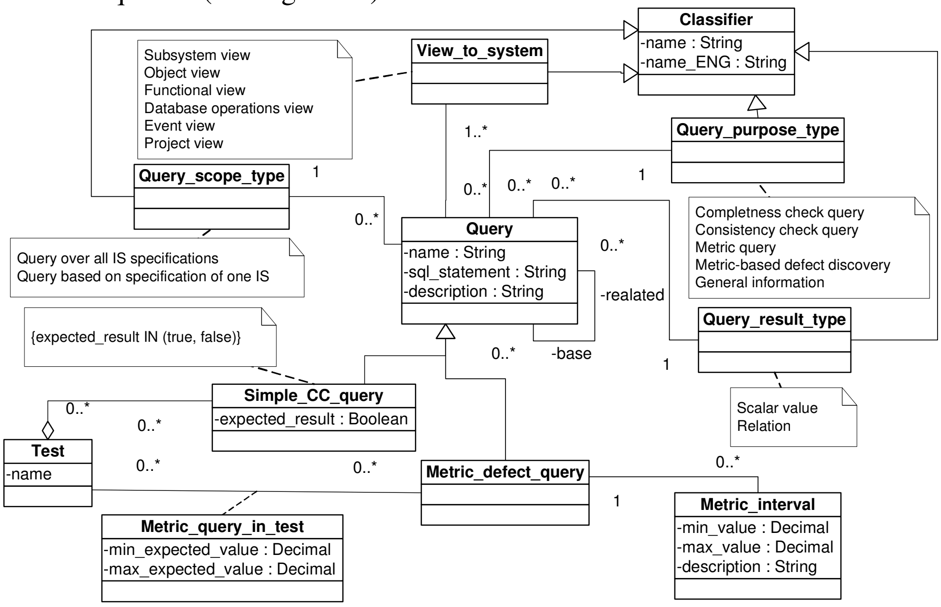


Figure 73 Metamodel of queries

The purpose of a query depends on the information that it helps to find:

- *Completeness or consistency (CC) problem of a model* (see section 4.3.1).
- *Value of metrics*. For example, Kim and Boldyreff (2002) present software metrics that are applicable to UML models. Choizon and Ueda (2006) summarize the existing work about object-oriented design metrics. *Examples* of metrics that are usable in this system are amount of areas of competence, amount of functional subsystems, amount of registers and average amount of use cases in functional subsystems.

- *Value of metrics that helps to find defects in a model.* A metrics can have associated threshold. If a metrics value is not between some predefined values, then it indicates existence of a problem in a model. Choinzon and Ueda (2006) present thresholds on metrics of object-oriented design that determine whether metrics values indicate critical situations or not. For example, we can count amount of use cases in different functional subsystems (FS). If some FS has a big amount of use cases compared to others, then it shows that we have to decompose this big FS.
- *Some other information that can be found from a model.* Queries can be used in order to classify elements of a model. For example, it is possible to determine whether a use case is concrete, abstract, base or addition use case (Larman, 2002, p. 388) by making query about its relationships with other use cases. Another example is that we do not have to separately record the events that influence a register but we can find them by using a query. A use-case case is triggered by an event. A step of a use-case can refer to a database operation, which creates/reads/modifies/deletes a data element that belongs to a register.

A query can give information about different aspects of a modeled system (see class *View_to_system* in Figure 73). For example, a query that finds average amount of use cases in functional subsystems is about subsystems view as well as about functional view. Query scope is either a specific model or all the models that are managed by our system. Result of a query can be either a scalar value or a relation. For example, the result of a simple CC check query is a (scalar) Boolean value.

A query that helps to find defects based on some metrics can have multiple intervals of associated values. These intervals correspond to the different severity-levels of a defect. For example, Choinzon and Ueda (2006) present the metrics "number of methods in a class" that has threshold of undesirable values "20-30 little bad, 31-50 bad, 50< very bad". A query, the result of which is a scalar value can belong to one or more tests. If one or more queries that belong to a test do not give an expected result, then it points to a defect in a model. The expected results of the queries are:

- All simple CC queries return a value that is the same as their expected value.
- All metrics queries that help to find defects return a value that is between minimum and maximum expected value.

4.3.1 Consistency and Completeness Checks

Structure of the database of the system analysis environment determines the elements that can be associated. It also enforces the relationship constraints according to which participation and/or cardinality is one.

However, the general principle of our system is that consistency and completeness of the models will not be ensured by the database constraints. This "tolerant" approach gives more freedom to a modeler. Occurrences of each consistency or completeness rule violation will be found by using a query or a set of queries. It must be possible to use these queries at any moment.

Examples of the completeness rules are:

1. An IS contains at least one area of competence (AC).
2. An IS contains at least one functional subsystem (FS).
3. An IS contains at least one registry subsystem (RS).
4. An AC has exactly one corresponding actor.
5. An actor (and therefore an AC) uses services of at least one FS.
6. Services of a FS are used by at least one actor.
7. An IS (through some of its FS) is used by at least one non-adjacent actor.
8. A FS uses services of at least one RS.
9. A RS has at least one FS that reads its data.
10. A RS has at least one FS that adds new data to it.
11. An IS has at least one administrative FS.
12. An IS has at least one administrative RS.
13. An IS has at least one business FS.
14. An IS has at least one business RS.
15. An IS (as a whole or through some of its FS) has at least one non-functional requirement from each of the different requirements types.
16. Functional requirements to a FS are described by using at least one use case.
17. A use case is associated with the description of the interest of a primary actor of this use case.
18. Ideally, all the open issues of a use case are solved.
19. A use case that is not an *essential use case* (Larman, 2002, p. 68) is associated with at least one database operation through some action.
20. A data element is created by at least one database operation (association through a post-condition).
21. A data element is read by at least one database operation (association through a post-condition).
22. A database operation has at least one post-condition.
23. A database operation is associated with at least one action that is part of a non-essential use case. Each database operation is used by at least one use case.
24. A register has at least one associated object type that has associated state-transition description.
25. An object type that has associated state-transition description has exactly one start state.
26. An object type that has associated state-transition description has one or more end-states.
27. It is possible to get from the start state of an object to any other state of an object by using state transitions.
28. A relationship has a mark about aggregation/composition/generalization at most at one end. In this case, the other end has no such marks.

29. A relationship end can either have mark about aggregation, composition or generalization but not more than one of them.

    Examples of the consistency rules are:

30. If there is a relationship between a FS and a RS according to which the FS creates/reads/updates/deletes data in the RS, then there must exist at least one use case that belongs to the FS so that this use case creates/reads/updates/ deletes a data element that belongs to the RS.

Completeness and consistency rules are well-formedness rules. Next, we present examples of the queries that could be constructed based on these rules:

1. A query where the result is a Boolean value. If the result of the query is the same as the expected result (see class *Simple_CC_query* in Figure 73), then it means that the specification of selected information system conforms to this rule. For example: "Find whether a model satisfies the rule that each area of competence has exactly one corresponding actor."
2. A query where the result is a relation:

- Queries that find parts of a model that *do not conform to* a CC rule. For example: "Find areas of competence that do not have exactly one corresponding actor."
- Queries that find parts of a model that *conform to* a CC rule. For example: "Find areas of competence that have exactly one corresponding actor."

Queries can also be used in order to find suspicious parts of a model that may or may not be the mistakes. For example, a query can search registers and data elements that are not subject of the update or delete operations.

4.4 Discussion and Comparisons

This system does not follow the popular approach according to which systems have to be specified by using a visual language. There exist researchers who think that a system specification does not have to be created by using a visual language. Brooks (1987) writes in his paper about the "silver-bullets" in software engineering: "In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable."

However, UML is nowadays widely used notation and therefore it is reasonable to teach it even after we start to use this system. For the teaching and presentation purposes, it is sometimes useful to see system specifications in the form of UML diagrams. A possible solution is a functionality of our system that allows us to generate XMI files based on the data in the database. These files, that contain UML models, can be opened in a CASE tool. Paige and Ostroff (2003) also support the idea that a modeling system must be able to produce multiple views from a single model.

The use of a central database and a web-based and form-based modeling interface are not unique features of our system. Commercial system EA WebModeler (EA Web Modeler, 2006) provides form-based and web-based user interface for creating models. Habela (2000) describes a system that allows us to extend a metamodel through form-based interface. Commercial systems Modelmosaic (Delen et al., 2005) and EA WebModeler also record models in a database. Ritter and Steiert (2000) present UML Repository. It allows us to record UML models in a centralized database that is created by using an ORDBMS$_{SQL}$. They see many advantages of such approach including more easy cooperation between developers, possibility to detect design errors by using database constraints and possibility to analyse the models by using the query facilities. Chapter 2 contains references to much more software engineering systems that use the help of a DBMS. "Appendix C: Comparison of some systems that record models in a database" compares our system analysis environment with UML repository (Ritter and Steiert, 2000) and UML Model Measurement Tool (UMMT) (Lavazza and Agostini, 2005).

The database of our system uses the schema design "Non-encapsulated Artifact Element Types" (see section 3.1.2). For the checking of the well-formedness rules we use the approach according to which a model can be checked by using queries (see approach 1 in section 3.2.4). Such solution is partially forced by the fact that we use ORDBMS$_{SQL}$ PostgreSQL. It provides limited means for creating declarative database constraints (see section 3.5.2).

For example, the purpose of the following query (58) is CC (consistency and completeness) check, its result is a scalar (Boolean) value and its scope is one information system. The query checks whether an information system consists of at least one functional subsystem.

```
SELECT IS_EMPTY('SELECT 1 FROM functional_subsystem          (58)
WHERE information_system_id=#IS#') AS result;
```

As you can see, we use IS_EMPTY function (see section 3.5) that is implemented as an SQL-invoked function. Its argument is a SQL SELECT statement. This statement contains placeholder "#IS#" that is replaced with the identifier of an information system if this query is executed.

The purpose of the following query (59) is CC check, its result is a relation and its scope is one information system. The query finds the names of functional subsystems that do not have associated use cases.

```
SELECT name AS result FROM functional_subsystem AS FS          (59)
WHERE information_system_id= #IS# AND NOT EXISTS
(SELECT 1 FROM use_case AS UC WHERE
FS.functional_subsystem_id=UC.functional_subsystem_id);
```

Metamodel of the language that is used in our system, contains whole-part relationships. We do not implement these relationships by using complex types (constructed multiset or row types or user-defined structured types) as declared types of columns in base tables. Firstly, our research shows that in case of using complex types we have to create constraints or queries that are more complex (see section 3.3.2.3) than in case of not using these types. In addition, PostgreSQL currently does not follow the SQL standard completely and therefore it is not possible to implement all the designs (see section 3.3.2.2) that use complex types as attribute types.

We have created a partial prototype of the system. This prototype allows us to manage subsystems (see Figure 68). It also makes possible the management and execution of the queries (see section 4.3). "Appendix D: The location of prototype system" describes the location and extent of the prototype in more detail.

A modelling language that follows the principle of the single model must satisfy the following three criteria: conceptual integrity, consistency of views, wide spectrum applicability (Paige and Ostroff, 2001). Our proposed solution satisfies the "conceptual integrity" criterion because models are recorded in one database. Each model element is recorded only once. Our proposed solution satisfies the "consistency of views" criterion because checking of the consistency of different views of a model is automated. Our proposed solution partly satisfies the "wide-spectrum applicability" criterion. This system is used in order to perform strategic- and detailed analysis of the system but not design or implementation. The system could be extended so that it could generate stored procedures based on the database operations and table specifications based on the data-elements.

Why cannot we use existing free software in order to model systems by using a single model type? Examples of such systems are SystemSpecifier (Systematik holistik metodik, 2006) that allows us to create system matrices and OPCAT (Dori et al., 2003) that allows us to create OPM models. The first reason is that they do not fully support the methodological framework for the Enterprise Information System (EIS) strategic analysis. They do not allow us to specify different types of subsystems and their interconnections. In addition, the proposed system uses the help of a DBMS but OPCAT and SystemSpecifier are file-based systems. The use of a DBMS helps to avoid well-known problems of the file-based systems like separation-, isolation- and duplication of data. More than one modeler can work with the same model at a time. A problem of an older version of OPCAT (for example ver. 2.55) is that it does not provide explicitly CCC checks functionality.

The use of the integrated model prevents repeating recording of the same information and thus helps to avoid inconsistencies. Our proposed system can also be used in order to collect information about the work amount and performance of modelers. It would also be a useful e-learning tool because it is planned to be a web-based system.

Barghouti et al. (1996) specify requirements to the information management component of Process-Centered Software Engineering Environment (PSEE). Next, we compare our proposed system with some of these requirements.

The data types that are used in a PSEE database might not be only "simple" predefined data types. In the database of our system, we use only predefined data types. Therefore, we do not use methods in order to access encapsulated data. Barghouti et al. (1996), on the other hand, propose the use of SQL-invoked methods as one well-established mean of preserving data integrity. Firstly, we use the schema design "Non-encapsulated Artifact Element Types" (see section 3.1.2). It causes creation of many base tables with many columns that use either predefined- or user-defined distinct types. In addition, the DBMS that we use provides limited support to user-defined data types (see Table 19). Finally, results of section 3.3 show that the use of complex data types in base tables has some disadvantages.

Barghouti et al. (1996) have the position that data of a PSEE does not have to be in a single database. Instead, a PSEE may use heterogeneous means for data management (see section 2.2.2) by incorporating different storage systems (files, DBMS). The system that is proposed in this chapter records data in a single database that is created and managed by using an $\text{ORDBMS}_{\text{SQL}}$.

Barghouti et al. (1996) think that the information management component of a PSEE must provide "a facility that supports abstract views of the data". Our system is built on top of the $\text{ORDBMS}_{\text{SQL}}$ that allows us to create views. Unfortunately, these views are not updateable without further programming (see Table 19). In addition, our system allows creating, recording and executing named queries that find information from the database.

Users of a PSEE start long-lasting sessions where actions cannot be determined a priori and where information flow between a user and the system is bidirectional (Barghouti et al., 1996). The latter property indicates that the PSEE is an interactive system.

Users perform actions in our system in order to fulfil their tasks. For example, a user can create a functional subsystem, modify name of a use case or delete a non-functional requirement. Conceptually a session consists of sequence of actions that are performed by a user during some period. A user action causes execution of one or more data manipulation statements that belong to one transaction. The database of our system contains only the minimal set of database constraints and therefore permits inconsistent and incomplete models. Each table has a primary key constraint. Values of a primary key are generated by the system. In addition, we have created foreign key constraints. These constraints enforce certain order of actions because a modeler must register data in a parent table (table without foreign key) before registering data in a child table (table with the foreign key). For example, it is not possible to create a use case before registration of the functional subsystem that contains this use case.

A DBMS ensures that data in a database satisfies the database constraints after each action. However, the database of our system can contain a model that does not satisfy some well-formedness rules. "A PSEE repository reaches global consistency incrementally as the sessions corresponding to the task's subtasks complete." (Barghouti et al., 1996)

In our system, a user can execute at any moment queries in order to find violations of the well-formedness rules (see section 4.3). Results of these queries give information that helps to improve the model and to move gradually towards the complete and consistent model. Barghouti et al. (1996), on the other hand, propose to use integrity constraints that are implemented by using triggers in order to check the well-formedness of software engineering artifacts.

The execution of a query in our system does not block concurrent data modification because our system is built on top of a DBMS (PostgreSQL 8.0) that uses Multi Version Concurrency Control mechanism (Weikum and Vossen, 2002). The modification of a data item (a row) does not block concurrent reading of this data item and vice versa.

The sessions in PSEE must share data collaboratively (Barghouti et al., 1996). It is also possible in our system because the results of performing an action become visible after the corresponding transaction is committed.

4.5 Summary

In this chapter, we described the principles of the system that help to perform strategic- and detailed analysis of information systems. We used the findings of previous chapters in order to design it.

This system allows us to record one integrated model of a system into a database by using the form-based and web-based user interface. The system provides predefined queries for finding consistency and completeness (CC) problems of a model. These queries can be used at any moment during the modeling process. The system allows us to specify additional queries, if needed. Some of these queries might not be for CC checks but, for example, could find metrics values.

We have also created a prototype of part of the system by using $\mathrm{ORDBMS}_{\mathrm{SQL}}$ PostgreSQL and PHP language.

CONCLUSIONS

This chapter summarizes the results of the dissertation and outlines future work.

Summary of Contributions

We investigated two data models – the underlying data model of the SQL:2003 standard (OR_{SQL}) and the data model that is explained in The Third Manifesto (OR_{TTM}). The dissertation had eight objectives. We will give a summary of these objectives and explain what we have done in order to achieve these aims.

Siau and Rossi (1998) describe methods for evaluating information modeling methods (see section 1.2). One of the comparison methods is a metamodel-based comparison. If we have metamodels of data models, then we can use the same method in order to compare the data models.

The first objective was to present a comparison of OR_{SQL} and OR_{TTM} based on their metamodels. Firstly, we proposed the method for such comparison (see section 1.3.1). The comparison is presented in sections 1.3.2 - 1.3.7. One precondition of this kind of work is the existence of metamodels of data models (in this case metamodels of OR_{SQL} and OR_{TTM}). Melton (2003b) writes: "The structure of the Definition Schema is a representation of the data model of SQL." The specification of the Definition Schema contains a *logical design* data model (meaning 2) of a database catalog in the form of DDL statements (Melton, 2003c). On the other hand, a metamodel should present a *conceptual* model with all the important relationships (including generalization and whole-part) in order to help to understand the meaning of constructs in a data model and interconnections of these constructs. It is not possible to understand OR_{SQL} only based on the Definition Schema description. Therefore, we had to study *Part 2: SQL/Foundation* (Melton, 2003) as well, in order to create the metamodel. Obviously, SQL:2003 does not provide a clear and compact specification of OR_{SQL}. In this regard, our work is somewhat similar to the work of Codd and Date (1975). They had to create definitions of the concepts of the network data model based on CODASYL DBTG language proposals in order to be able to compare the network and the relational data model.

There are works (OMG ad/01-02-01), (Calero et al., 2006), (Pedro et al., 2006), (DMTF CIM, 2006) that present some *parts of a* metamodel or an ontology of *SQL* (see section 1.4) This dissertation extends these works and presents the OR_{SQL} metamodel that covers data types, data structures, data operators, and data integrity rules. Section 1.4.5 explains which parts of SQL are not covered by the OR_{SQL} metamodel. We are not aware of any existence of the OR_{TTM} metamodel and therefore we present it as well.

The metamodel-based comparison of OR_{SQL} and OR_{TTM} consists of the following parts:

- Mapping between the metaclasses of the OR_{SQL} and OR_{TTM} metamodels. A pair of metaclasses presents the constructs that have exactly the same semantics (semantic equivalence) or quite similar semantics.
- Report of discrepancies between the data models (OR_{SQL} and OR_{TTM}).
- Metrics values that are calculated based on the metamodels. These values show the relative complexity of the data models. We calculated the metrics values for OR_{TTM} and OR_{SQL} and for the underlying data model of SQL:1992.
- Examples of violations of the orthogonality principle by OR_{SQL}. We discovered these violations by observing the OR_{SQL} metamodel.

The use of metamodel-based comparisons is not a new idea. However, the use of this kind of a method in order to compare *data models* is a novel result of our work. There are discrepancies between OR_{SQL} and OR_{TTM}.

Some metaclasses of the OR_{TTM} metamodel do not have a corresponding metaclass in the OR_{SQL} metamodel. This may be caused by different reasons:

- OR_{TTM} does not specify a construct but its authors do not prohibit it.
- Authors of OR_{TTM} deliberately do not specify a construct because they think that existing constructs are sufficient and the additional construct increases complexity without increasing the expressive power.

In the first case, creators of OR_{TTM} think that features that correspond to the missing metaclasses, are orthogonal to the data model. For example, OR_{SQL} specifies a large amount of predefined data types but OR_{TTM} requires only the data type Boolean. However, it does not prohibit other predefined data types but DBMS vendors can choose which types to implement.

An example of the second case is that the authors of OR_{TTM} think that the use of constructed reference types is a mistake because it leads back to the complexities with the pointers. For the same reasons they do not support the use of typed tables. They also think that a generalization relationship between the two tables can be implemented by using virtual relvars (viewed tables) (see section 3.3.1) and the use of supertable-subtable feature of OR_{SQL} is unnecessary.

Some metaclassess in OR_{TTM} do not have a corresponding metaclass in OR_{SQL}. For example, it is not possible to create *declarative* transition constraints and use RELATION Gen type generator in OR_{SQL}.

Some metaclasses in OR_{SQL} have more than one corresponding metaclass in OR_{TTM}. For example, OR_{TTM} distinguishes between the concepts *relvar value* (loosely speaking the table value that consists of rows that are in the table) and *relvar* (loosely speaking the specification of a table structure). On the other hand, OR_{SQL} uses the concept *table* in both cases. If we use the OR_{SQL} concept "table", then we have to explain what the exact meaning of this concept is in a particular context (for example, whether we want to update table structure or data in the table).

Some metaclasses in OR_{TTM} have more than one corresponding metaclass in OR_{SQL}. For example, the OR_{SQL} metamodel contains metaclasses *Assertion*, *CHECK constraint*, *Table check constraint* and *Table constraint*, but their corresponding metaclass in the OR_{TTM} metamodel is *Database constraint*. In this case, the metamodel of OR_{SQL} is overly complicated. For example, why do we have to distinguish between table constraints and assertions? The possible result is that we currently cannot use assertions in any DBMS (Türker and Gertz, 2001).

We inspected visual structures in the OR_{SQL} metamodel and discovered some violations of the *orthogonality principle* (see section 1.3.8). These violations are an addition to the examples that have already been presented in the literature (see the work of Date and Darwen (2000)). We did not create the metamodel of entire SQL language (see section 1.4.5). Nevertheless, the amount of metaclasses in the OR_{SQL} metamodel is larger than in the OR_{TTM} metamodel. In addition, the metaclasses of the OR_{SQL} metamodel have many more attributes. Metrics values show that OR_{SQL} is more complex than OR_{TTM}. A programming language that displays orthogonality must provide a comparatively *small* set of primitive constructs. We conclude that the designers of OR_{TTM} have paid more attention to the principle of orthogonality as compared to the designers of OR_{SQL}.

The metrics values (see section 1.3.7) show relative complexity of the data models but they do not show their "goodness". Metrics values of underlying data model of SQL:1992 are smaller as compared to OR_{SQL} and OR_{TTM}. This data model has many shortcomings (see section 2.3). Metrics values of OR_{TTM} are smaller as compared to OR_{SQL}. However, difficulties in creating the OR_{SQL} metamodel (see section 1.3.6), violations of the orthogonality principle in OR_{SQL} (see section 1.3.8) and discrepancies between OR_{SQL} and OR_{TTM} (see section 1.3) are small proofs that the OR_{SQL} data model has shortcomings that can cause difficulties in using $ORDBMS_{SQL}$s in software engineering systems (and in any other system as well). Demonstration of these shortcomings was *the fourth objective* of this dissertation.

The second objective was to find out what the problems of using RDBMSs and ORDBMSs in software engineering systems according to the existing research literature are. Chapter 2 presents a literature-based study of the software engineering systems that use the help of a DBMS. We found (as expected) that many researchers think that the relational data model and RDBMSs are not a suitable platform for software engineering systems. However, these opinions are based on the interpretation of the relational model by the SQL standard and the implementation of the standard by the $RDBMS_{SQL}$s. In addition, existing *overviews* about software engineering systems refer to few systems that use a $RDBMS_{SQL}$. It confirms that the relational model is not suitable for the engineering systems. However, we found many software engineering systems that use a $RDBMS_{SQL}$. We found more systems that use a $RDBMS_{SQL}$ than systems that use an $ORDBMS_{SQL}$. One reason is that $\text{ORDBMS}_{\text{SQL}}$s have been available for shorter time as compared to $\text{RDBMS}_{\text{SQL}}$s. On the other hand, some papers about the systems that use a $\text{RDBMS}_{\text{SQL}}$ have been published after the $\text{ORDBMS}_{\text{SQL}}$s came into existence. The main results of this chapter are:

- We presented a more thorough list of software engineering systems that use a $\text{RDBMS}_{\text{SQL}}$ or an $\text{ORDBMS}_{\text{SQL}}$ (see sections 2.2.4.1 and 2.2.4.3) than the existing overview papers. We found 16 systems that use only a $\text{RDBMS}_{\text{SQL}}$ and 6 systems that use only an $\text{ORDBMS}_{\text{SQL}}$.
- Based on the literature study we compiled the lists of problems of the relational data model and $\text{RDBMS}_{\text{SQL}}$s. These problems make the use of the relational model and $\text{RDBMS}_{\text{SQL}}$s in the software engineering systems more difficult (see section 2.3). We found that many of these problems are orthogonal to the relational data model (as defined by The Third Manifesto) or are caused by the shortcomings of implementation of the relational model in the current standards and systems.

In addition, we found that existing research papers about the software engineering systems that use an $\text{ORDBMS}_{\text{SQL}}$ pay little attention to the discussion of problems of OR_{SQL} and $\text{ORDBMS}_{\text{SQL}}$s.

The third objective was to describe the design alternatives of databases of software engineering systems that will be implemented by using an ORDBMS. Firstly, in section 2.2.4.3 we described some of the software engineering systems that use an $\text{ORDBMS}_{\text{SQL}}$. In Chapter 3, we presented different approaches to repository schema design (see section 3.1.1). We also presented different approaches how to check well-formedness of artifacts (see section 3.2) and one possible approach how to perform versioning (see section 3.2.4.1). In sections 3.1 and 3.2 we described approaches that are usable in an $\text{ORDBMS}_{\text{TTM}}$ database as well as in an $\text{ORDBMS}_{\text{SQL}}$ database. We analyzed the "Universal Database Design" approach (see section 3.1.4) and concluded that it has many more disadvantages than advantages. Despite that, this kind of design is sometimes used in the $\text{ORDBMS}_{\text{SQL}}$ databases. Existing literature usually refers to only some problems of this design (query complexity, performance). We found thirteen different types of problems.

We worked out principles of the system that helps to perform strategic and detailed analysis of information systems (see Chapter 4). It allowed us to put some of the ideas from the third chapter into action. This system uses the approach according to which artifact element types are not encapsulated and have corresponding tables (see section 3.1.2). In addition, system allows us to check well-formedness of the artifacts by using the queries (see section 3.2.4). We created a partial prototype of this system in order to prove the concept (see "Appendix D: The location of prototype system").

The fourth objective was to demonstrate that the OR_{SQL} data model has shortcomings that cause difficulties in using $\text{ORDBMS}_{\text{SQL}}$s in the software engineering systems. The papers that describe the use of the OR_{SQL} data model and $\text{ORDBMS}_{\text{SQL}}$s, concentrate mostly on the positive aspects of this model and the authors apply their new features as much as possible (see section 2.2.4.3 and 3.3.2.1). Therefore, we thought that a more balanced treatment of OR_{SQL} and $ORDBMS_{SQL}s$ is needed. Section 3.5.2 describes the problems of the OR_{SQL} data model that occur if we try to implement a software engineering system. In conclusion, we can say that there are many problems.

In addition, we referred to the specific problems that occur if we use two $ORDBMS_{SQL}s$ – PostgreSQL8.0 and Oracle 10g. As an example, we constructed a table (Table 19) that demonstrates the problems that make it more difficult to implement whole-part relationships in a database.

Therefore, this section also helped to demonstrate that there is a gap between the principles of OR_{SQL} and the actual implementation (practice) in current $ORDBMS_{SQL}s$. This was our *fifth objective*. This gap causes additional problems to the designers of software engineering systems. These problems occur during the development of any system, not only a software engineering system.

The *sixth objective* was to demonstrate that the OR_{TTM} data model is a suitable basis for a DBMS so that this DBMS can be used in a software engineering system (SES). We did not implement a SES on top of an $ORDBMS_{TTM}$. Instead, we presented examples of relvars and constraints that were created by using $ORDBMS_{TTM}$ Rel (Voorish, 2005). Unfortunately, this system was not mature enough and therefore we were not able to test all the examples (see beginning of Chapter 3). Despite that, we think that we have achieved this objective. Firstly, in sections 3.1 - 3.4 we described possible designs in terms of constructs of the OR_{TTM} data model. Secondly, in Chapter 2 we found that many SESs have successfully used the help of an $ORDBMS_{SQL}$ or even a $RDBMS_{SQL}$. If we can use an $ORDBMS_{SQL}$, then why cannot we use an $ORDBMS_{TTM}$? The reason could be that OR_{SQL} provides constructs that are necessary in order to build up a SES but they are missing in OR_{TTM}. We found a mapping between the metaclasses of the OR_{SQL} and OR_{TTM} metamodels and discrepancies of these data models. One type of discrepancy is that a metaclass in the OR_{SQL} metamodel does not have a corresponding metaclass in the OR_{TTM} metamodel. OR_{SQL} constructs (*typed table*, *REF Con.*, *Reference type*) that are sometimes used in the SESs (see section 2.2.4.3) do not have a counterpart in OR_{TTM}. However, existing research already shows that it is not actually a limitation because these features increase the complexity of a data model without providing clear advantages (Date and Darwen, 2000, Appendix J; Date, 2003 chapters 25 and 26). We have no reason to doubt in that based on sections 3.1 - 3.4. We do not claim that the OR_{TTM} data model is "silver-bullet" (Brooks, 1987) but we think that it makes the use of DBMSs in the software engineering systems easier and more comfortable.

Date and Darwen (2000, p. xiv) write: "Thus, we regard our Manifesto as being very much in spirit of Codd's original work and continuing along the path he originally laid down." Our research demonstrates that the **relational model**

is not outdated and based on that it is possible to create systems that manage complex data.

The seventh objective was to propose a set of designs for preserving the semantics of whole-part relationships in a database that is created by an ORDBMS$_{TTM}$ and guidelines explaining when to use these designs. Firstly, we investigated existing research about this topic (see section 3.3.2.1). It is a widespread opinion that it is advantageous to preserve whole-part relationship at the logical database level by having containment hierarchy within a *base* table/*real* relation. We presented six possible designs for implementing whole-part relationships (see section 3.3.2.2). Four of them use relvar attributes that have complex data types. We evaluated the alternatives in terms of some of the values of the whole-part relationship secondary characteristics (see section 3.3.2.3). We gave marks to the designs based on the amount of work that is needed in order to enforce all necessary integrity constraints that are imposed by the values of the secondary characteristics. After that, we constructed the comparison table and reorganized it by using the "minus technique" algorithm (see Table 18). This evaluation method is also a novel result of our work. We found that the designs that use complex data types in *real relvar*s, have stricter usage restrictions and a greater need for the accompanying constraints and special values than the designs that do not use these types in real relvars. If it is necessary to present to a user an artifact so that it is part of one relation, then it can be achieved by creating a *virtual* relvar.

The eighth objective was to extend the Principle of Orthogonal Database Design (Date and McGoveran, 1994) so that it would take into account the use of real relvars that have attributes with complex types. Application of this principle helps to avoid data redundancy across different relvars. We presented a motivating example (see section 3.4.1), the extended principle and examples of its usage (see section 3.4.2). We discovered that some database design guidelines that are presented in the literature do not follow this principle. In addition, we presented two *heuristic* rules that help to prevent data redundancy within the value of a real relvar, if this relvar has an attribute with a relation- or tuple type (see section 3.4.3).

Date and Darwen (2000, Appendix C) write: "in general, types should correspond to properties and relvars to entities." Existing research mainly focuses on the positive aspects of having complex types as declared types of columns in base tables. The so-called "traditional" designs do not use columns that have complex types. Our research shows that the "traditional" designs together with the *special values*, *integrity constraints* and *views*, that use attributes with complex types, offer actually more freedom and flexibility to the designers. Some of the reasons for preferring the "traditional" designs are: the need of virtual relvars (see section 3.1.1), integrity constraints (see sections 3.3.2.3 and 3.4.3) and difficulties to discover violations of the principle of orthogonal database design (see section 3.4.2).

Directions for Further Research

We have to complete the specification of the $\mathrm{OR}_{\mathrm{TTM}}$ and $\mathrm{OR}_{\mathrm{SQL}}$ data models. This includes the formal specification of constraints that correspond to the well-formedness rules by using OCL or other languages. For example, one possibility would be to use the relational language Tutorial D. The comparison method that was used in this dissertation is not ideal. For example, one well-known problem of using UML class diagrams is that it is sometimes difficult to decide whether to model some real-world construct as a class or as an attribute. Therefore, different modelers could model the same data model construct as a metaclass or an attribute of metaclass. It is better to use metaclasses *and* attributes in order to create a mapping between different metamodels. In addition, mapping between the attributes of two metamodels is itself a subject of interest. In this dissertation, we do not use attributes in the mapping due to space restrictions. We think that it is not a major problem because both metamodels are created by the same modeler by following the same modeling conventions. An alternative is to use some modeling notation that does not distinguish classes/entity types and attributes. For example, we could use Object-Role Modeling (ORM) notation (Halpin, 2001). An additional advantage is that ORM allows us to specify more constraints visually in the metamodel compared to UML (Halpin, 2001, p. 401). It might be useful if we use the metamodel as a teaching tool in order to explain data models.

A possible future study could cover the creation of short and clear specification of the $\mathrm{OR}_{\mathrm{SQL}}$ data model that uses a structure similar to that of The Third Manifesto (by describing prescriptions, proscriptions and suggestions). This kind of specification would be, for example, useful for pedagogical purposes.

One direction of research is to compare $\mathrm{OR}_{\mathrm{SQL}}$ and $\mathrm{OR}_{\mathrm{TTM}}$ with other data models by using the metamodel-based comparison. Firstly, we have to find or create metamodels of these data models. Probably it will give ideas how to improve the comparison method of data models. Therefore, in this case we will also perform *method engineering* by using an action research.

In the evaluation of the whole-part relationships, we did not consider some secondary characteristics of them: transitivity, configurationality, mutability. Future studies must take these characteristics into account. We must also study how transition constraints help to implement operational properties of relationships. We have to investigate how to implement model management operations (match, difference, merge, composition, apply, copy ModelGen (Bernstein, 2003)) if a repository is implemented as an $\mathrm{ORDBMS}_{\mathrm{TTM}}$ database.

Implementation of a software engineering system based on an $\mathrm{ORDBMS}_{\mathrm{TTM}}$ is needed in order to prove the validity of the design guidelines that were presented in this dissertation. We need suitable $\mathrm{ORDBMS}_{\mathrm{TTM}}$s for this task.

One direction of work is to create a small expert system that is able to assist database designers. For example, a database designer has to: (a) choose the type of a DBMS ($\text{ORDBMS}_{\text{SQL}}$ or $\text{ORDBMS}_{\text{TTM}}$), (b) choose the type of a relationship (whole-part or generalization), (c) determine the properties of this relationship (for example, in case of whole-part relationship – one optional locally exclusive part with mandatory wholes), (d) specify the participants in this relationship (names of the tables/relvars, their columns/attributes and types of columns/attributes), (e) optionally specify whether or not he or she prefers to use columns/attributes with complex data types in base tables/real relvars.

The system generates DDL statements. These statements create database objects that implement this relationship. The system could generate a code according to different design alternatives.

It would be very useful if a DBMS or some separate tool would be able to analyze the relvars/tables in a database in order to find violations of the Principle of Orthogonal Design (POOD). We are currently not aware of any such tool. Clearly, it is more difficult to find the predicate of a table in $\text{ORDBMS}_{\text{SQL}}$ databases because many constraints are implemented by using triggers or SQL-invoked routines and database developers tend to use predefined types. We think that it is worth to investigate whether it is possible to use POOD in case of virtual relvars. Celko (2005) notes that a large amount of views (virtual relvars) leads to schema management problems. Current DBMSs allow us to create two or more distinct views that have the same subquery. They allow us to create two or more distinct views that have different subqueries, but the results of these subqueries are exactly the same. In other words, they have the same predicate but different names. On the other hand, it does not seem right to completely prohibit the views with the overlapping meanings because the representatives of different roles may use them.

We have to complete the implementation of the system that was presented in Chapter 4. After the system will be ready, we can perform the usability study in order to evaluate which way users prefer to describe the system – using visual diagrams with little support to CC check or using textual descriptions with the extensive CC checks. We can also investigate how a modeling system, which allows at any time to check a model, changes the learning experience and habits of the students who use it.

One possible extension of this system is to integrate with it a subsystem that helps to record and retrieve patterns. Its user interface should be web-based and it should record its data in a central database. Then it is possible to create a reference between a pattern and an information system specification (that is created by using our proposed system). This reference could refer to the fact that the specification applies this pattern. The specification could be an example that stresses the need of using this pattern. The pattern management system can take advantage of a query facility (see section 4.3). Eessaar (2004b) presents some possible queries from the database of patterns.

In Chapter 4, we mentioned that the system should support metrics queries that have associated thresholds. Development of these queries is also one direction of our future work. We have to find metrics values that allow us to find defects in a model. After that, we have to find their thresholds, test them in real development projects and change the thresholds if necessary. We could use existing system specification documents in order to find average values of the selected metrics. It helps to determine the thresholds on these metrics. For example, the pattern "Seven Plus or Minus Two" that is part of UML pattern language (Evitts, 2000) proposes a solution: "Limit the number of elements in any given diagram to the magic number of seven, give or take two elements." *Maybe* it is also usable in case of use cases? If the amount of steps in a use-case *main success scenario* is fewer than five, then it could indicate that a use case is defined at a too low level "that is, as a single step, subfunction, or subtask within an Elementary Business Process." (Larman, 2002, p. 60) If the amount of steps is more than nine, then it could indicate that the use-case has become too large. In this case it maybe useful to split it.

In this dissertation, we presented a system where a SES user cannot change a repository schema. However, we have also proposed a metadata driven repository system (Pattern Management Software System) that allows users to dynamically extend the database schema (Eessaar, 2004a; Eessaar, 2005b). It has built-in support for evolution. It makes it possible to dynamically add support for the management of new types of software engineering artifacts. We do not want to design its repository according to the "Universal Data Model" design (see section 3.1.4) because of its numerous problems. Instead, we use the design "Non-encapsulated Artifact Element Types" (see section 3.1.2). This means that each metamodel element has a corresponding base table. The general idea of the system is that the user can specify the abstract syntax of languages as metamodels. The system records a metamodel in the database tables. In addition, it immediately generates and executes DDL statements based on the changes in the metamodel. For example, the creation of a metaclass causes at least the generation of a CREATE TABLE statement and execution of it. The system also creates triggers if a metaclass is associated with other metaclass by generalization relationship (see the work of Pokrajac et al. (2004) about how to implement this kind of relationship in PostgreSQL). The generated tables make possible to record artifacts that have been created by using this particular language. The system records data about the created database objects as well as mapping between the database objects and metamodel elements. Therefore, the system can make changes in the database schema if we modify the metamodel. Further development (including implementation) of this system is yet another possible direction of our work. Among other things, this system needs integrated user-interface generator. If we make some changes in the metamodel and the system modifies the database structure, then it should modify the user-interface as well. We think that such a system could take advantage of the approach, according to which HTML pages are dynamically generated based on the specification that is recorded in a database. In this case, it is possible to change user-interface by changing its specification that is recorded in a database.

KOKKUVÕTE

Relatsioonilised- ja objekt-relatsioonilised andmebaasisüsteemid kui tarkvaraarenduse tulemite haldamise platvorm.

Mõiste "andmemudel" on ülekoormatud ja sellel on erinevates kontekstides erinev tähendus. Antud töös käsitletakse mõistet "andmemudel" kui spetsifikatsiooni, mis kirjeldab andmebaasi looja käsutuses olevaid universaalseid ehitusplokke, reegleid, mis tagavad plokkidest moodustatava struktuuri kvaliteedi ning operatsioone, mida saab teha nende struktuuridega. Andmemudelite näited on hierarhiline-, võrk-, relatsiooniline- ja objekt-relatsiooniline andmemudel. Tuleb öelda, et tegu on üldnimedega, sest eksisteerib erinevaid nägemusi, kuidas peaks üks või teine selline andmemudel olema üles ehitatud ning milliseid võimalusi oma kasutajatele pakkuma.

Relatsioonilise andmemudeli idee pakkus laiale avalikkusele välja E.F Codd oma 1970. aastal avaldatud artiklis. Tema ideid arvesse võttes töötati välja SQL keel millest sai ajapikku standard. Andmebaasisüsteemid, mis kasutavad seda keelt muutusid populaarseks ja laialdaselt kasutatavaks ning on seda tänaseni. Kuid paljud uurijad ning arendajad väidavad, et relatsiooniline mudel on ajale jalgu jäämas, sest sellel põhinevaid andmebaase on raske kui mitte võimatu kasutada keeruka struktuuriga andmete hoidmiseks ja töötlemiseks. Üheks rakenduse tüübiks, mis kasutavad ja loovad taolisi andmeid on tarkvaraarenduse süsteemid.

Tulenevalt relatsioonilise mudeli *väidetavast* sobimatusest mõningate rakenduste jaoks on pakutud välja uusi andmemudeleid. Antud töös keskendutakse objekt-relatsioonilisele andmemudelile, mis peaks endas ühendama relatsioonilise andmemudeli ja objekt-orienteeritud programmeerimisest tuntud võtted. Töö autor uurib kahte objekt-relatsioonilist andmemudelit – SQL:2003 standardi aluseks olev mudel (edaspidi kasutatakse selle tähistamiseks lühendit "OR_{SQL}") ja mudel mida kirjeldatakse Kolmandas Manifestis (edaspidi kasutatakse selle tähistamiseks lühendit "OR_{3MF}").

Alates SQL standardi versioonist SQL:1999 on SQL keelde lisatud objekt-orienteeritud programmeerimiskeeltest tuntud vahendeid (tüüpide deklareerimine, tabelite deklareerimine tüüpide põhjal, "tüübitud" tabelite vahelise pärimise võimaldamine jne.).

Kolmanda Manifesti loojad seevastu leiavad, et SQL sisaldab liiga palju puudusi ja kõrvalekaldeid relatsioonilise mudeli põhimõtetest ning et need ei ole mitte ainult teoreetilise arutelu teemaks *vaid tekitavad ka raskusi reaalsete süsteemide loomisel*. Kolmanda Manifesti autorite mõtteviisi kohaselt on kõiki häid omadusi, mida oodatakse objekt-relatsiooniliselt andmemudelilt, võimalik realiseerida *relatsioonilise mudeli raamistikus*.

Käesoleva töö alguses püstitati kaheksa eesmärki. Järgnevalt kirjeldatakse, mida on nende saavutamiseks tehtud.

Esimeseks eesmärgiks on esitada OR_{SQL} ja OR_{3MF} andmemudelite metamudelitel põhinev võrdlus. Taolise meetodi järgi võrdlemist on kasutatud näiteks ontoloogiate ja modelleerimismeetodite võrdlemiseks. Antud töö üks uudne tulemus on, et sellist meetodit kohaldatakse andmemudelite võrdlemiseks (peatükk 1.2). Selleks, et oleks võimalik võrdlust läbi viia tuleb kõigepealt luua andmemudelite metamudelid. UML'i klassidiagrammide abil esitatud metamudelid on peatükkides 1.3.2-1.3.5. Lisaks sisaldavad need peatükid andmemudelite võrdlust, mille käigus esitatakse erinevatesse metamudelitesse kuuluvate metaklasside paarid. Paarid moodustuvad sellistest metaklassidest, mis modelleerivad semantiliselt ekvivalentseid või väga sarnaseid andmemudelite konstruktsioone. Paljudele metaklassidele ei õnnestu teise andmemudeli metamudelist paarilist leida või siis vastab ühele metaklassile mitu metaklassi. See on viide andmemudelite lahknevusele. Peatükis 1.3.7 esitatakse metamudelite põhjal väljaarvutatud meetrikate väärtused, mis võimaldavad hinnata andmemudelite suhtelist keerukust. Vaadeldavateks meetrikateks on metaklasside arv, metaklasside atribuutide arv ning nende kahe arvu summa. Esitatud väärtuste kohaselt on OR_{SQL} suhteliselt keerukam võrreldes OR_{3MF}ga kuid see ei tähenda veel, et OR_{SQL} on parem. Peatükis 1.3.8 esitatakse OR_{SQL} metamudeli inspekteerimisel leitud probleemid, mis viitavad, et OR_{SQL} andmemudel ei pea piisavalt kinni hea programmeerimiskeele disaini põhimõtetest.

Teiseks eesmärgiks on teha olemasolevate teadustööde põhjal kindlaks, millised on relatsiooniliste ja objekt-relatsiooniliste andmebaasisüsteemide puudused, mis raskendavad nende kasutamist tarkvaraarenduse süsteemides. Peatükis 2 esitatakse ülevaade tarkvaararaarenduse süsteemidest, mis kasutavad andmebaasisüsteemide abi. Kirjeldatakse süsteeme, mis kasutavad relatsioonilist- (vt. peatükk 2.2.4.1), objekt-relatsioonilist- (vt. peatükk 2.2.4.3), objekt-orienteeritud- (vt. peatükk 2.2.4.2) või spetsiaalselt inseneritarkvara jaoks mõeldud andmebaasisüsteemi (vt. peatükk 2.2.3). Leidub ka heterogeenseid süsteeme, mille üks osa andmetest on andmebaasis ning teine osa on failides, mida andmebaasisüsteemi poolt ei hallata (vt. peatükk 2.2.2). Käesoleva töö puhul on uudne, et olemasolevad ülevaate artiklid viitavad suhteliselt väikesele arvule tarkvaraarenduse süsteemidele, mis kasutavad relatsioonilisi- või objekt-relatsioonilisi andmebaasisüsteeme. Antud töös on aga selliseid süsteeme leitud tunduvalt rohkem (16, mis kasutavad ainult relatsioonilist ja 6, mis kasutavad ainult objekt-relatsioonilist andmebaasisüsteemi). Kõik need süsteemid kasutavad andmebaasisüsteeme kus on tarvitusel SQL keel. Leitud teadustööd süsteemide kohta, mis kasutavad SQL keelt võimaldavad kokku panna nimekirja probleemidest, mida tuuakse välja relatsioonilise andmemudeli ja relatsiooniliste andmebaasisüsteemide kohta (vt. peatükk 2.3). Tegelikult on need kriitikaks SQL'i ning seda kasutavate andmebaasisüsteemide kohta, sest Kolmanda Manifesti põhimõtteid järgides taolisi probleeme ei tekiks.

Kolmandaks eesmärgiks on kirjeldada tarkvaraarenduse süsteemi andmebaasi alternatiivseid disainilahendusi juhul kui kasutusel on objekt-relatsiooniline andmebaasisüsteem. Kolmandas peatükis kirjeldatakse erinevaid andmebaasi skeemi disainilahendusi (vt. peatükk 3.1). Muuhulgas analüüsitakse peatükis 3.1.4 nn. "universaalse andmebaasi disaini" lahendust ja jõutakse järeldusele, et tema puudused kaaluvad üle võimalikud eelised. Samuti kirjeldatakse kolmandas peatükis kuidas saaks kontrollida andmete (sealhulgas tarkvaraarenduse tulemite) vastavust reeglitele (vt. peatükk 3.2) ning esitatakse disainilahendus versioonide haldamiseks (vt. peatükk 3.2.4.1). Mõningaid kolmandas peatükis välja pakutud kavandeid kasutatakse neljandas peatükis esitatud süsteemianalüüsi keskkonna loomisel. Veebikeskkonnas töötavad kasutajad registreerivad süsteemi kirjelduse serveris paikneva OR_{SQL} andmemudelil põhineva objekt-relatsioonilise andmebaasi tabelitesse. Loodud spetsifikatsioonist vigade otsimiseks on võimalik kasutada päringuid.

Neljandaks eesmärgiks on demonstreerida, et OR_{SQL} andmemudelil on puuduseid võrreldes OR_{3MF} andmemudeliga. Need puudused muudavad OR_{SQL} andmemudelil põhinevate andmebaaside loomise ja kasutamise raskemaks. Peatükis 3.5.2 kirjeldatakse OR_{SQL} andmemudeli probleeme, mis ilmnevad kui sellel mudelil põhinevaid andmebaasisüsteeme soovitakse kasutada tarkvaraarenduse süsteemide loomiseks. Kokkuvõttena võib öelda, et neid probleeme on palju.

Viiendaks eesmärgiks on demonstreerida, et erinevused OR_{SQL} andmemudeli ning seda mudelit järgivates andmebaasisüsteemides esineva praktika vahel muudavad selliste andmebaasisüsteemide kasutamise tarkvaraarenduse süsteemides veel raskemaks. Peatükis 3.5 vaadeldakse muuhulgas võimalusi, mida kaks objekt-relatsioonilist andmebaasisüsteemi (Oracle 10g ja PostgreSQL8.0) pakuvad andmebaasi programmeerijale, et realiseerida kolmanda peatüki eelmistes alapeatükkides kirjeldatud andmebaasi disainilahendusi. Näitena esitab tabel "Table 19" puudused, mis ei lase realiseerida osa-terviku seoseid peatükis 3.3.2 kirjeldatud viisil. Tuleb tõdeda, et need näited kinnitavad eesmärgis sõnastatud probleemi olemasolu.

Kuuendaks eesmärgiks on demonstreerida, et OR_{3MF} andmemudelil põhinev andmebaasisüsteem on tarkvaraarenduse süsteemides kasutamiseks sobiv. Käesoleva töö raames ei realiseerita tarkvaraarenduse süsteemi kasutades OR_{3MF} andmemudelil põhinevat andmebaasisüsteemi. Vaatamata sellele võib öelda, et eesmärk on saavutatud. Töös kirjeldatakse andmebaasi disaini põhimõtteid kasutades OR_{3MF} andmemudeli mõisteid (vt. peatükid 3.1-3.4) ning tuuakse koodinäiteid kasutades Tutorial D keelt. Teises peatükis viidatakse mitmetele tarkvaraarenduse süsteemidele, mis kasutavad OR_{SQL} andmemudelil põhineva andmebaasisüsteemi abi. Põhjus miks OR_{3MF} andmemudelil põhinev andmebaasisüsteem selliseks ülesandeks ei sobi võib olla, et OR_{SQL} andmemudel pakub hädavajalikke konstruktsioone, mis OR_{3MF} andmemudelis puuduvad. Metamudelitesse kuuluvate metaklasside vaheliste vastavuste analüüs näitab, et OR_{3MF} andmemudel ei toeta viite tüübi konstruktoreid ja "tüübitud tabeleid". Kuid olemasolevad uuringud näitavad, et see ei ole tegelikult puudus ja piirang, sest need võimalused suurendavad mudeli keerukust samas ilma selget eelist pakkumata (vaadake Date, 2003 ptk. 25 ja 26; Date and Darwen, 2000, Appendix J).

Seitsmendaks eesmärgiks on pakkuda välja disainisoovitused, mis juhendavad andmebaasi struktuuri, kitsenduste ja operaatorite kavandamist sellisel juhul, kui kontseptuaalses andmemudelis esineb osa-terviku seos. OR_{SQL} andmemudeli puhul soovitavad mitmed uuringud osa-terviku seoste realiseerimiseks kasutada *baastabelite* veerge, millel on kasutaja-defineeritud struktuurne andmetüüp või ROW või MULTISET tüübikonstruktori abil konstrueeritud andmetüüp (vt peatükk 3.3.2.1). Töös pakutakse välja kuus võimalikku kavandit, millest neljas kasutatakse nn. *keerukaid andmetüüpe* - korteeži tüüp, relatsiooni tüüp ja kasutaja-defineeritud skalaarne tüüp. Seejärel hinnatakse kavandeid vastavalt sellele kui palju tuleb näha vaeva, et jõustada kõik vajalikud andmete terviklikkuse reeglid (vt. peatükk 3.3.2.3). Need reeglid tulenevad osa-terviku seose nn. teiseste karakteristikute väärtustest. Hinnete põhjal koostatakse tabel "Table 18" ning reorganiseeritakse see kasutades miinustehnika algoritmi. Järelduseks on, et disainilahendused mille korral baasrelatsioonides kasutatakse keerukate tüüpidega atribuute nõuavad keerukamaid terviklikkuse kitsendusi ja rohkem spetsiaalväärtuseid kui disainilahendused, mis selliseid tüüpe baasrelatsioonides ei kasuta. Kui on vaja esitada kasutajale andmeid ühte relatsiooni kuuluvana, siis võib seda teha kasutades virtuaalseid relatsioone mille atribuudid on mõnda keerukat tüüpi.

Kaheksandaks eesmärgiks on esitada laiendatud *ortogonaalse andmebaasi disaini printsiip*. Selle printsiibi järgimine aitab vältida olukorda, kus andmebaasis on ühe ja sama olemi kohta käivad ühte tüüpi andmed mitmes erinevas relatsioonis. Printsiibi originaalversioon ei võta arvesse võimalust, et baasrelatsioonides kasutatakse keerukaid andmetüüpe kuid laiendatud printsiip (vt. peatükk 3.4.2) võtab selle arvesse. Lisaks esitatakse peatükis 3.4.3 kaks heuristilist reeglit, mida võib kasutada andmete liiasuse vältimiseks ühe baasrelatsiooni piires juhul kui selles relatsioonis on atribuut, millel on korteeži või relatsiooni tüüp.

Enamik siiani avaldatud uurimustest objekt-relatsiooniliste andmebaaside kohta rõhutab, et keerukate andmetüüpide kasutamine *baastabelites* on kasulik. Kuid käesolev uurimistöö demonstreerib, et "traditsiooniline" disain (mis ei kasuta keerukaid andmetüüpe baastabelites) koos kitsenduste ja vaadetega pakuvad kasutajatele palju rohkem paindlikust. Selliste lahenduste kasutamise eelduseks on andmebaasisüsteemi poolne ulatuslik toetus deklaratiivsete kitsenduste ja vaadete loomisele. Vaadete kaudu peab saama muuta andmeid baastabelites. Paraku tänapäeva SQL keelt kasutavad süsteemid on selles osas ebapiisavad ning sellest tuleneb ka vajadus relatsioonilist andmemudelit "laiendada" ja lisada sinna uuendusi, mida muidu ei oleks vaja.

REFERENCES

1. **Agarwal, R., Sinha, A. P.** 2003. Object-oriented modeling with UML: a study of developers' perceptions. Communications of the ACM, Vol. 46, No. 9, pp. 248-256.
2. **Agrawal, R., Jagadish, H. V.** 1987. Direct Algorithms for Computing the Transitive Closure of Database Relations. *In*: Proceedings of the 13th International Conference on Very Large Data Bases, 1-4 September 1987 Brighton, England. Morgan Kaufmann. pp. 255-266.
3. **Ahnøj, J.** 2003. Generic Design of Web-Based Clinical Databases. Journal of Medical Internet Research, Vol. 5, Issue 4. Retrieved 21 July, 2006, from http://www.jmir.org/2003/4/e27/
4. **Albrecht, M. Buchholz, E. Duesterhoeft, A. Thalheim, B.** 1998. An Informal and Efficient Approach for Obtaining Semantic Constraints Using Sample Data and Natural Language Processing. Semantics in Databases, LNCS Vol. 1358/1998. pp. 1-28.
5. **Allsop, D.J., Harrison, A., Sheppard, C.** 2002. A database architecture for reusable CommonKADS agent specification components. Knowledge-Based Systems Journal. Elsevier Science, Vol. 15, Issues 5-6, July 2002. pp. 275-283.
6. Alphora. Dataphor 2.0. Retrieved April 08, 2006, from http://alphora.com./tiern.asp?ID=DATAPHOR2
7. **Althoff, K.D., Birk, A., Hartkopf, S., Müller, W., Nick, M., Surmann, D., Tautz, C.** 1999. Systematic Population, Utilization, and Maintenance of a Repository for Comprehensive Reuse. *In*: Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, Learning Software Organizations Methodology and Applications, LNCS Issue 1756. Germany: Springer-Verlag, pp. 25 – 50.
8. **Ambriola, V., Conradi, R., Fugetta, A.** 1997. Assessing Process-centered Software Engineering Environments. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3. pp. 283 – 328.
9. **Astrova, I.** 2003. On Integration of Object-oriented Applications with Relational Databases. Doctoral Thesis on Informatics and System Engineering, Tallinn University of Technology, TTU Press.
10. **Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S.** 1989. The Object-Oriented Database System Manifesto. In: Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Vol. 57.
11. **Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A., Bruel, J.M.** 2003. Formalization of the Whole-Part Relationship in the Unified Modeling Language. IEEE Transactions on Software Engineering, Vol. 29, No. 5, pp. 459-470.

12. **Barghouti, N.S., Emmerich, W., Schaefer, W., Skarra, A.** 1996. Information Management in Process-Centered Software Engineering Environments. *In*: Software Process. Trends in Software. Wiley. pp. 53-87.
13. **Basili, V.R., Caldiera, G., Rombach, H.D.** 1994. The Experience Factory. *In*: Encyclopedia of Software Engineering, New York: John Wiley & Sons, 1994. pp. 469-476. Retrieved August 22, 2006, from ftp://ftp.cs.umd.edu/pub/sel/papers/fact.pdf
14. **Baskerville, R., Pries-Heje, J.** 2001. Racing the e-bomb: how the Internet is redefining information system development methodology. *In*: Realigning Research and Practice in Information System Development, Proceedings of the IFIP TC8/WG8.2 Working Conference, July 27-29, Boise, Idaho, USA, pp. 49-68.
15. **Batory, D., Thomas, J.** 1997. P2: A Lightweight DBMS Generator. Journal of Intelligent Information Systems, Vol. 9, No. 2, pp. 107-124.
16. **Bećarević, D., Roantree, M.** 2004. A Metadata Approach to Multimedia Database Federations. Information and Software Technology, Vol. 46, No. 3., pp. 195-207.
17. **Bednárek, D., Obdržálek, D., Yaghob, J., Zavoral, F.** 2005. Data Integration Using DataPile Structure. *In*: Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems, 12-15 September 2005 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 178-188.
18. **Behle, A.** 1998. An Internet-based information system for cooperative software reuse. *In*: Proceedings of the 5th International Conference on Software Reuse, 02 – 05 June 1998. IEEE Computer Society, pp. 236-245.
19. **Bernstein, P.A.** 1998. Repositories and ObjectOriented Databases. SIGMOD Record. Vol. 27, No. 1 (Mar. 1998), pp. 88-96.
20. **Bernstein, P.A.** 2003. Applying Model Management to Classical Meta Data Problems. *In*: Proceedings of the Conference on Innovative Data Systems Research. pp. 209-220.
21. **Bernstein, P.A., Dayal, U.** 1994. An Overview of Repository Technology. *In*: Proceedings of the 20th International Conference on Very Large Data Bases, 12-15 September 1994 Santiago de Chile, Chile. Morgan Kaufmann, pp. 705-713.
22. **Bernstein, P.A., Harry, B., Sanders, P., Shutt, D., Zander, J.** 1997. The Microsoft Repository. *In*: Proceedings of the 23rd International Conference on Very Large Data Bases, 25-29 August 1997 Athens, Greece. Morgan Kaufmann, pp. 3-12.
23. **Bernstein, P.A., Halevy, A.Y., Pottinger, R.** 2000. A Vision of Management of Complex Models. SIGMOD Record. Vol. 29, Part 4, pp. 55-63.
24. **Blaha, M., LaPlant, D., Marvak, E.** 1998. Requirements for Repository Software. *In*: Proceedings of the Working Conference on Reverse

Engineering, Honolulu, Hawaii, USA. IEEE Computer Society Press. IEEE Computer Society, pp. 164-173.

25. **Blanning, R.W.** 1982. Data management and model management: a relational synthesis. *In*: Proceedings of the 20th annual Southeast regional conference. New York: ACM Press, pp. 139-147.
26. **Boisvert, R.F.** 1994. Architecture of an intelligent virtual mathematical software repository system. Mathematics and Computers in Simulation, Vol. 36, No. 4, pp. 269-279.
27. **Boldyreff, C., Nutter, D., Rank, S.** 2002. Active Artefact Management for Distributed Software Engineering. *In*: Proceedings of the 26th Annual International Computer Software and Applications Conference 26-29 August 2002, IEEE Computer Press, pp. 1081-1086.
28. **Boyd, S.W., Keromytis, A.** 2004. SQLrand: Preventing SQL Injection Attacks. *In*: Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, LNCS Vol. 3089/2004. Germany: Springer Berlin, pp. 292–302.
29. **Brandon, D.** 2002. Crud matrices for detailed object oriented design. J The Journal of Computing in Small Colleges, Vol. 18, No. 2 (Dec. 2002), pp. 306-322.
30. **Brash, D., Stirna, J.** 1999. Describing Best Business Practices: A Pattern-based Approach for Knowledge Sharing. *In*: Proceedings of the 1999 ACM SIGCPR conference on Computer personnel research, Brisbane, Queensland, Australia. New York: ACM Press, pp. 57-60.
31. **Braz, L.M.** 1990. Visual syntax diagrams for programming language statements. *In*: Proceedings of the 8th Annual international Conference on Systems Documentation, Little Rock, Arkansas, United States. New York: ACM Press, pp. 23-27.
32. **Brooks, F.P.** 1987. No Silver Bullet: Essence and Accidents of SoftwareEngineering. IEEE Computer, Vol. 20, No. 4, pp. 10-19.
33. **Brown, J.W.** 2000. AntiPatterns in Project Management. John Wiley & Sons.
34. **Calero, C., Ruiz, F., Baroni, A.L., Brito e Abreu, F., Piattini, M.** 2006. An Ontological Approach to Describe the SQL:2003 Object-Relational Features. Journal of Computer Standards & Interfaces, Vol. 28, Issue 6, September 2006, pp. 695-713. Retrieved July 31, 2006, from http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/2005/baroniCSIinPress.pdf
35. **Celko, J.** 2005. Joe Celko's SQL Programming Style. Morgan Kaufmann.
36. **Ceri, S., Cochrane, R., Widom, J.** 2000. Practical Applications of Triggers and Constraints: Success and Lingering Issues. *In*: Proceedings of the 26th international Conference on Very Large Data Bases, 10 – 14 September 2000 Cairo, Egypt. Morgan Kaufmann, pp. 254-262.
37. **Chandrasekaran, B., Josephson, J.R., Benjamins, V.R.** 1999. What Are Ontologies, and Why Do We Need Them? IEEE Intelligent Systems and their Applications, Vol. 14, No. 1, January/February 1999, pp. 20-26.

38. **Chaudhuri, S., Weikum, G.** 2000. Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. *In*: Proceedings of the 26th international Conference on Very Large Data Bases. Morgan Kaufmann, pp. 1-10.
39. **Chen, R.S., Nadkarni, P., Marenco, L., Levin, F., Erdos, J., Miller, P.L.** 2000. Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute-Value Representation. Journal Of The American Medical Informatics Association, 2000 Sep-Oct; Vol. 7, Part 5, pp. 475-87.
40. **Chen, Y.F., Nishimoto, M.Z., Ramamoorthy, C.V.** 1990. The C Information Abstraction System. IEEE Transactions on Software Engineering, Vol.16, No. 3, pp. 325-334.
41. **Chisholm, M.** 2003. How to Build a Business Rules Engine: Extending Application Functionality through Metadata Engineering. Morgan Kaufmann Publishers, Elsevier.
42. **Choinzon, M., Ueda, Y.** 2006. Design Defects in Object Oriented Designs Using Design Metrics. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering, 28-31 August 2006 Tallinn, Estonia. IOS Press. pp. 61-72.
43. **Cochrane, R., Pirahesh, H., Mattos, N.M.** 1996. Integrating triggers and declarative constraints in SQL database systems. *In*: Proceedings of the 22th International Conference on Very Large Data Bases, 03 – 06 September 1996 Mumbai (Bombay), India. USA:IEEE, pp. 567–579.
44. **Cockburn, A.** 1998. Basic Use Case Template, Version 2, October 26, 1998, Retrieved March 11, 2006, from http://alistair.cockburn.us/usecases/uctempla.doc
45. **Codd, E.F.** 1970. A relational model of large shared data banks. Communications of the ACM, Vol. 13, No. 6, pp. 377-387.
46. **Codd, E.F.** 1981 Data models in database management. *In*: Proceedings of the workshop on Data abstraction, databases and conceptual modelling. ACM SIGART Bulletin, Issue 74 (Jan. 1981), pp. 112-114.
47. **Codd, E.F., Date, C.J.** 1975. Interactive support for non-programmers: The relational and network approaches. *In*: Proceedings of the 1975 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. New York: ACM Press, pp. 11-41.
48. **Connolly, T.M., Begg, C.E.** 2002. Database systems. A Practical Approach to Design, Implementation and Management. 3rd edn. Pearson/ Addison Wesley.
49. **Conradi, R., Westfechtel, B.** 1998. Version models for Software Configuration Management. ACM Computing Surveys, Vol. 30, No. 2, pp. 232-282.
50. **Consens, M., Mendelzon, A., Ryman, A.** 1992. Visualizing and querying software structures. *In*: Proceedings of the 14th international Conference on Software Engineering, 11 – 15 May 1992 Melbourne, Australia. New York: ACM Press, pp. 138-156.

51. **Constantopoulos, P., Jarke, M., Mylopoulos, J., Vassiliou, Y.** 1995. The Software Information Base: A Server for Reuse. VLDB Journal, Vol. 4, No. 1, pp. 1- 43.
52. **Conte, F.A., Hassine, I., Rieu, D., Tastet, L.** 2004. An Information System Development Tool Based on Pattern Reuse. *In*: Proceedings of the Sixth International Conference on Enterprise Information Systems, 14 – 17 April 2004 Porto, Portugal, Vol. 3. pp. 548 - 551.
53. **Cox, A., Clarke, C., Sim, S.** 1999. A model independent source code repository. *In*: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, 08 – 11 November 1999 Mississauga, Ontario, Canada. IBM Press.
54. **Darwen, H.** 2003. How To Handle Missing Information Without Using Nulls. Presentation in Warwick University. Retrieved September 7, 2006, from http://www.dcs.warwick.ac.uk/~hugh/TTM/Missing-info-without-nulls.pdf
55. **Date, C.J.** 1995. Relational database writings, 1991-1994. Addison Wesley.
56. **Date, C.J.** 1998. Relational database writings, 1994-1997 / by C. J. Date: with special contributions by Hugh Darwen and David McGoveran. Addison Wesley.
57. **Date, C.J.** 2001. The Database Relational Model: A Retrospective Review and Analysis: A historical account and assessment of E. F. Codd's contribution to the field of database technology. Addison-Wesley.
58. **Date, C.J.** 2003. An Introduction to Database Systems. 8th edn. Pearson/Addison Wesley.
59. **Date, C.J.** 2005 Database in Depth: Relational Theory for Practitioners. O'Reilly. Chapter 1 – Introduction. Retrieved August 13, 2006, from http://searchoracle.techtarget.com/searchOracle/downloads/ Database_in_Depth_Chapter_1.pdf
60. **Date, C.J.** 2006. The Relational Database Dictionary: A Comprehensive Glossary of Relational Terms and Concepts, with Illustrative Examples. O'Reilly Media.
61. **Date, C.J., Codd, E.F.** 1975. The relational and network approaches: Comparison of the application programming interfaces. *In*: Proceedings of the 1975 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. New York: ACM Press, pp. 83-113.
62. **Date, C.J., Darwen, H.** 1992. Relational Database Writings 1989-1991. Addison Wesley.
63. **Date, C.J., Darwen, H.** 2000. Foundation for Future Database Systems: The Third Manifesto, 2nd edn. Addison-Wesley.
64. **Date, C.J., Darwen, H.** 2006. Databases, Types and the Relational Model, 3rd edn. Addison Wesley. Chapter 4 – The Third Manifesto. Retrieved August 13, 2006, from http://www.dcs.warwick.ac.uk/~hugh/TTM/CHAP04.pdf

65. **Date, C.J., Darwen, H., Lorentzos, N.A.** 2003 Temporal Data and the Relational Model: A Detailed Investigation into the Application of Interval and Relation Theory to the Problem of Temporal Database Management. Morgan Kaufmann.
66. **Date, C.J., McGoveran, D.** 1994. The Principle of Orthogonal Design. Database Programming & Design 7, No. 6 (June 1994). Retrieved December 10, 2005,from http://www.dbdebunk.com/page/page/622331.htm
67. **Davies, I., Green, P., Milton, S., Rosemann, M.** 2003. Using Meta Models for the Comparison of Ontologies. *In*: Proceedings Evaluation of Modeling Methods in Systems Analysis and Design Workshop - EMMSAD'03, Klagenfurt/Velden.
68. **de Freitas Sodré, V., Jugurta, L. F., Vilela, V. M., and Andrade, M. V.** 2005. Improving Productivity and Quality of GIS Databases Design using an Analysis Pattern Catalog. *In*: Proceedings of the Second Asia-Pacific Conference on Conceptual Modelling, ACM International Conference Proceeding Series, Vol. 107. Australia: Australian Computer Society, pp. 107-114.
69. **Delen, D., Dalal, N. P., Benjamin P. C.** 2005. Integrated modeling: the key to holistic understanding of the enterprise. Communications of ACM. Vol. 48, No. 4, pp. 107-112.
70. **Demuth, B., Hussmann, H.** 1999. Using UML/OCL Constraints for Relational Database Design. *In*: Proceedings of the 2nd International Conference on the Unified Modeling Language, October 28-30 1999 Fort Collins, Colorado, USA, LNCS Vol. 1723/1999. Springer, pp. 598-613.
71. **Dittrich, K., Tombros, D., Geppert, A.** 2000. Databases in Software Engineering: a roadmap. *In*: Proceedings of the Conference on the Future of Software Engineering, 04 – 11 June 2000 Limerick, Ireland. New York: ACM Press, pp. 293-302.
72. DMTF Common Information Model (CIM) Standards. CIM Schema Ver. 2.13. Database specification. Retrieved October 16, 2006, from http://www.dmtf.org/standards/cim/cim_schema_v213/CIM_Database.pdf
73. **Do, H.H., Rahm, E.** 2004. Flexible Integration of Molecular-Biological Annotation Data: The GenMapper Approach. *In*: Proceedings of the 9th International Conference on Extending Database Technology 14-18 March 2004 Heraklion, Greece, LNCS Vol. 2992/2004. Germany: Springer Berlin, pp. 811 – 822.
74. **Dori, D.** 2002. Why Significant Change in UML is Unlikely. Communications of the ACM, Nov.2002, pp. 82-85.
75. **Dori, D., Reinhartz-Berger, I., Sturm, A.** 2003. OPCAT - A Bimodal CASE Tool for Object- Process Based System Development. *In*: Proceedings of the Fifth International Conference on Enterprise Information Systems. pp. 286-291.
76. EA Web Modeler. Agilense Enterprise Architecture Frameworks. Retrieved March 12, 2006, from

http://www.agilense.com/documents/agilense_frameworks.doc

77. **Eckstein, J., Bergin, J., Marquardt, K., Manns, M. L., Sharp, H., Wallingford, E.** 2001. Patterns for Experimental Learning, Proceedings of EuroPLoP 2001, Retrieved March 16, 2006, from http://www.pedagogicalpatterns.org/current/experientiallearning.pdf
78. **Eessaar, E.** 2004a. Towards Pattern Management System. *In*: Proceedings of the Sixth International Conference on Enterprise Information Systems, 14 – 17 April 2004 Porto, Portugal. Vol. 3. pp. 655 – 658.
79. **Eessaar, E.** 2004b. Methods for Searching Patterns from the Database of Patterns. *In*: The 16th Conference on Advanced Information Systems Engineering Forum Proceedings, 7-11 June 2004 Riga, Latvia. pp. 103 – 111.
80. **Eessaar, E.** 2005a. Truly Relational Databases as a Platform for the Artifact Management. *In*: Proceedings of the Fourteenth International Conference on Information Systems Development: Pre-Conference 14-17 August 2005 Karlstad, Sweden. pp. 207-218.
81. **Eessaar, E.** 2005b. Architecture of Pattern Management Software System. *In*: Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems, 12-15 September 2005 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 189-207.
82. **Eessaar, E.** 2006a. Extended Principle of Orthogonal Database Design. *In*: Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, 15-17 February 2006 Madrid, Spain. pp. 360-365. CD-ROM.
83. **Eessaar, E.** 2006b. Guidelines about Usage of the Complex Data Types in a Database. WSEAS Transactions on Information Science and Applications, Vol. 3, Issue 4, April 2006, pp. 712-719.
84. **Eessaar, E.** 2006c. Using Relational Databases in the Engineering Repository Systems. *In*: Proceedings of the Eighth International Conference on Enterprise Information Systems, 23 –27 May 2006 Paphos, Cyprus. Vol. Databases and Information Systems Integration. pp. 30 – 37.
85. **Eessaar, E.** 2006d. Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 10th WSEAS International Conference on COMPUTERS, 13-15 July 2006 Vouliagmeni, Athens, Greece. pp. 1263-1268. CD-ROM.
86. **Eessaar, E.** 2006e. SQL or Third Manifesto Compliant Object-Relational Database Management Systems as the Platforms for Maintaining the Whole-Part Relationships in a Database. WSEAS Transactions on Computers, Vol. 5, Issue 10, October 2006, pp. 2440-2447.
87. **Eessaar, E.** 2006f. Integrated System Analysis Environment for the Continuous and Completeness Checking. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering 2006, 28-31 August 2006 Tallinn, Estonia. IOS Press. pp. 96-105.

88. **Eessaar, E.** 2006g. Preserving Semantics of the Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 15th International Conference on Information Systems Development, August 31 - September 2 2006 Budapest, Hungary. Springer. (forthcoming).
89. **Eessaar, E.** 2006h. Metamodel-based Comparison of Data Models. *In*: Proceedings of the International Conference on Systems, Computing Sciences and Software Engineering (SCS^2 06). Springer. (accepted paper).
90. **Eessaar, E.** 2007. Using Metamodeling in order to Evaluate Data Models. *In*: Proceedings of the 6th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, 16-19 February 2007 Corfu, Greece. (accepted paper).
91. **Emmerich, W.** 1995. Tool Construction for Process-Centred Software Development Environments based on Object Databases. PhD Thesis. University of Paderborn, Germany. Retrieved September 10, 2006, from http://www.cs.ucl.ac.uk/staff/W.Emmerich/publications/ PHDTHESIS/thesis.pdf
92. **Emmerich, W., Schäfer, W., Welsh, J.** 1992. Suitable Databases for Process-centred Environments Do not yet Exist. *In*: Proceedings of the Second European Workshop on Software Process Technology. UK: Springer-Verlag London, pp. 94-98.
93. **Engels, G., Groenewegen, L.** 2000. Object-oriented modeling: a roadmap. *In*: Proceedings of the Conference on the Future of Software Engineering, 04 – 11 June 2000 Limerick, Ireland. New York: ACM Press, pp.105–116.
94. **Engle, P.** 2003. Data Modeling – Left and Right. The Data Administration Newsletter. Retrieved October 07, 2005, from http://www.tdan.com/i024hy03.htm
95. **Englebert, V., Hainaut, J.L.** 1999. DB-MAIN: A next generation meta-CASE. Journal of Information Systems, Vol. 24, No. 2, April 1999, pp. 99-112.
96. **Evitts, P.** 2000. UML Pattern Language. Macmillian Technical Publishing.
97. **Feldmann, R.L.** 1999. Developing a Tailored Reuse Repository Structure - Experience and First Results. *In*: Proceedings of the Workshop on Learning Software Organizations, 16 June 1999 Kaiserslautern.
98. **Ferguson, E.** 2003. Object-oriented concept mapping using UML class diagrams. Journal of Computing Sciences in Colleges, Vol.18, Issue 4 (Apr. 2003), pp. 344-354.
99. **Fernström, C.** 1993. Process WEAVER: Adding Process Support to UNIX. *In*: Proceedings of the 2nd International Conference on the Software Process, 25-26 February 1993 Berlin, Germany. IEEE CS Press. pp. 12–26.
100. **Florijn, G., Meijers, M., Winsen, P.V.** 1997. Tool support for object-oriented patterns. *In*: Proceedings of 11th European Conference on Object-Oriented Programming, 9–13 June 1997 Jyväskylä, Finland, LNCS Vol. 1241/1997. Germany: Springer Berlin, pp. 472-495.

101. **Fowler, M.** 1997. Analysis Patterns: Reusable Object Models. Addison Wesley Professional.

102. **Gamma, E., Helm, R., Johnson, R., Vlissides, J.** 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional.

103. **Geoffrion, A. M.** 1989. Computer-based modeling environments. European Journal on Operational Research, Vol. 41, No. 1, pp. 33–43.

104. **Glinz, M.** 2000. A Lightweight Approach to Consistency of Scenarios and Class Models. *In*: Proceedings of the 4th International Conference on Requirements Engineering, 19-23 June 2000 Schaumburg, IL, USA. pp. 49-58.

105. **Gray, P.** 1997. CASE tool construction for a parallel software development methodology. Information and Software Technology, Vol. 39, No. 4, pp. 235-252.

106. **Gray, P., Welland, R.** 1999. Increasing the flexibility of modelling tools via constraint-based specification. *In*: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, 08 – 11 November 1999 Mississauga, Ontario, Canada. IBM Press. p. 3.

107. **Greenfield, J., Short, K., Cook, S., Kent, S.** 2004. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley Publishing, Inc.

108. **Gruber, T.R.** 1995. Towards principles for the design of ontologies used for knowledge sharing. International Journal of Human Computer Studies, Vol. 43, No. 5/6, pp. 907-928.

109. **Gruhn, V., Schneider, M.** 1998. Workflow Management Based on Process Model Repositories. *In*: Proceedings of the 20th international Conference on Software Engineering, 19 – 25 April 1998 Kyoto, Japan. USA: IEEE Computer Society, pp. 379-388.

110. **Guizzardi, G.** 2005. Ontological Foundations for Structural Conceptual Models. Telematica Instituut Fundamental Research Series No. 15. Ph.D. thesis, University of Twente. Retrieved April 5, 2006, from https://doc.freeband.nl/dscgi/ds.py/Get/File-56338

111. **Gulutzan, P., Pelzer, T.** 1999. SQL-99 Complete, Really. CMP Books.

112. **Guo, J., Luqi, A.** 2000. A Survey of Software Reuse Repositories. *In*: Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 3-7 April 2000. USA: IEEE Computer Society. pp 92 -100.

113. **Habela, P.** 2002. Metamodel for Object-Oriented Database Management Systems. PhD Thesis. Polish Academy of Sciences, Warsaw, Poland August 2002. Retrieved August 3 2006, from http://www.planetmde.org/phds/phds/MetamodelForObject OrientedDatabaseManagementSystems.pdf

114. **Hageman, D., Reeves, D.M.** 2001. net-Trials TM Clinical Trials Information System. In: Proceedings of 14th IEEE Symposium on

Computer-Based Medical Systems, 26-27 July 2001 Bethesda, MD, USA. pp. 141-145.

115. **Halpin, T.** 2001. Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. Morgan Kaufman Publishers.

116. **Halpin, T., Bloesch, A.** 2000. Modeling Collection in UML and ORM. *In*: Proceedings of the 5th IFlP WG8.1 International Workshop on Evaluation of Modeling Method in System Analysis and Design.

117. **Hammer, M., Mc Leod, D.** 1981. Database description with SDM: A Semantic Database Model. ACM Transactions on Database Systems. Vol. 6, No. 3 (Sep. 1981), pp. 351-386.

118. **Hardwick, M.** 1984. Extending the relational database data model for design applications. *In*: Proceedings of the 21st Conference on Design Automation, 25 – 27 June 1984 Albuquerque, New Mexico, US. USA, NJ: IEEE Press Piscataway, pp. 110-116.

119. **Hardwick, M., Samaras, G.** 1989. Using a relational database as an index to a distributed object database in engineering design systems. *In*: Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering, 16-18 October 1989 Gaithersburg, MD, USA. pp. 4-11.

120. **Hardwick, M., Spooner, D.L.** 1989. The ROSE data manager: Using object technology to support interactive engineering applications. IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, pp. 285-289.

121. **Harrison, W., Ossher, H., Tarr, P.** 2000. Software Engineering Tools and Environments: A Roadmap. *In*: Proceedings of the Conference on the Future of Software Engineering, 04 – 11 June 2000 Limerick, Ireland. New York: ACM Press, pp. 261-277.

122. **Haskin, R.L., Lorie, R.A.** 1982. On extending the functions of a relational database system. *In*: Proceedings of the ACM SIGMOD International Conference on Management of data. New York: ACM Press, pp. 207-212.

123. **Hay, D.C.** 1996. Data model patterns: conventions of thought, New York: Dorset House Pub.

124. **Haynie, M.N.** 1981. The relational/network Hybrid data model for Design Automation Databases. *In*: Proceedings of the 18th Conference on Design Automation June 29 - July 01 1981 Nashville, Tennessee, US. NJ: IEEE Press Piscataway, pp. 646-652.

125. **Henderson-Sellers, B., Barbier, F.** 1999. Black and White Diamonds. *In*: Proceedings of the Second International Conference "UML" '99 - The Unified Modeling Language: Beyond the Standard, October 1999 Fort Collins, CO, USA, LNCS Vol. 1723/1999. Germany: Springer Berlin, pp. 550-565.

126. **Henderson-Sellers, B., Atkinson, C., Kühne, T., Gonzalez-Perez, C.** 2003. Understanding Meta-modelling. Tutorial in 22nd International

Conference on Conceptual Modeling ER2003, 15 October 2003. Retrieved November 20, 2004, from http://www.er.byu.edu/er2003/slides/ER2003T1HendersonSellers.pdf

127. **Henninger, S.** 2001. Turning Development Standards into a Repository of Experiences. Software Process Improvement and Practice, Vol. 6, No. 3, pp. 141-155.

128. **Herbst, H.** 1996. Business rules in systems analysis: A meta-model and repository system. Information Systems, Vol. 21, Issue 2, April 1996, pp. 147-166.

129. **Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.** 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. The Computer Journal, Vol. 42, No. 2, pp. 100–111.

130. **Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.** 1987. PRIMA-a DBMS Prototype Supporting Engineering Applications. *In*: Proceedings of the International Conference on Very Large Data Bases 1-4 September 1987 Brighton, England. Morgan Kaufmann, pp. 433—442.

131. **Härder, T., Mahnke, W., Ritter, N., Steiert, H.P.** 2000. Generating Versioning Facilities for a Design-data Repository Supporting Cooperative Applications. *In*: International Journal of Cooperative Information Systems, Vol. 9, Part. 1/2, 2000, pp. 117–146.

132. **Jasper, H.** 1994. Active Databases for Active Repositories. *In*: Proceedings of the 10th International Conference on Data Engineering, 14 – 18 February 1994 Houston, TX, USA. IEEE Computer Society, pp. 375-384.

133. **Jäderlund, C.** 1981. Systematrix. Complete SMX handbook. Stockholm.

134. **Kaiser, G.E., Barghouti, N.S., Feiler, P.H., Schwanke, R.W.** 1988. Database Support for Knowledge Based Engineering Environment. IEEE Expert, Vol. 3, No. 2, pp. 18-32.

135. **Kalnins, A. Barzdins, J. Celms, E.** 2005. *In*: Model Driven Architecture. LNCS Vol. 3599/2005. pp. 62-76.

136. **Katz, R.H.** 1990. Toward a Unified Framework for Version Modeling in Engineering Databases. ACM Computing Surveys, Vol. 22, No. 4, pp. 375– 409.

137. **Keller, R.K., Bedard, J.F, Saint-Denis, G.** 2001. Design and Implementation of a UML-Based Design Repository. *In*: Proceedings of the 13th International Conference on Advanced Information Systems Engineering, June 4-8 June 2001 Interlaken, Switzerland, LNCS Vol. 2068/2001. Germany: Springer Berlin, pp. 448 – 464.

138. **Kemper, A., Lockemann, P.C., Wallrath, M.** 1987. An object-oriented system for engineering applications. *In*: Proceedings of the 1987 ACM SIGMOD international Conference on Management of Data, 27 – 29 May 1987 San Francisco, California, United States. New York: ACM Press, pp. 299-310.

139. **Keqin, L., Lifeng, G., Hong, M., Fuqing, Y.** 1997. An Overview of JB (Jade Bird) Component Library System JBCL. *In*: Proceedings of the Technology of Object-Oriented Languages and Systems-Tools-24, 01 – 01 September 1997. Washington, DC: IEEE Computer Society, pp. 206.

140. **Kiesel, N., Schürr, A., Westfechtel, B.** 1995. GRAS, a Graph-Oriented (Software) Engineering Database System. Information Systems, Vol. 20, No. 1, pp. 21-52.

141. **Kim, H., Boldyreff, C.** 2002. Developing software metrics applicable to UML Models. *In*: Proceedings of the 6th International Workshop on Quantitative Approaches in Object–Oriented Software Engineering, 10-14 June 2002 Málaga, Spain. Germany: Springer Berlin, pp. 147-153.

142. **Kolovos, D. S., Paige, R. F., and Polack, F. A.** 2006. Model comparison: a foundation for model composition and model transformation testing. *In*: Proceedings of the 2006 international Workshop on Global integrated Model Management, 22 May 2006 Shanghai, China. New York: ACM Press, pp. 13-20.

143. **Kovse, J., Härder, T., Ritter, N.** 2002. Supporting Mass Customization by Generating Adjusted Repositories for Product Configuration. *In*: Proceedings of the International Conference CAD 2002 - Corporate Engineering Research, 04-05 March 2002. pp. 17-26.

144. **Kyte, T.** 2001. Expert One-on-One Oracle, Wrox Press.

145. **Kyte, T.** 2003. Effective Oracle by Design. Oracle Press, McGraw-Hill/Osborne.

146. **Lange, C., Sneed, H.M., Winter, A.** 2001. Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. *In*: Proceedings of the 9th International Workshop on Program Comprehension 12-13 May 2001 Toronto, Canada. IEEE Computer Society. pp. 209-218.

147. **Larman, C.** 2002. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2nd edn. Prentice Hall, Upper Saddle River, USA.

148. **Lavazza, L., Agostini, A.** 2005. Automated Measurement of UML Models: an open toolset approach. Journal of Object Technology. Vol. 4, No. 4, May-June 2005.

149. **Leff, A., Rayfield, J. T.** 2006. IBM Research Report. Relational Blocks: Fully Declarative Visual Application Assembly. RC23908 (W0603-069) March 9, 2006 Computer Science. Retrieved October 6, 2006, from http://domino.research.ibm.com/library/cyberdig.nsf/papers/A3CB4C1C2499057A852571370059465D/$File/rc23908.pdf

150. **Lejter, M., Meyers, S., Reiss, S.P.** 1992. Support for Maintaining Object-Oriented Programs. IEEE Transactions on Software Engineering. Vol. 18, Issue 12 (Dec. 1992), pp. 1045-1052.

151. **Levy, A.Y., Rajaraman, A.,Ullman, J.D.** 1996. Answering Queries Using Limited External Query Processors. *In*: Proceedings of the Fifteenth

ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ACM Press, pp. 227-237.

152. **Li, J.L., Li, M.X., Deng, H.Y, Duffy, P.E., Deng, H.W.** 2005. PhD: a web database application for phenotype data management. Bioinformatics, Vol. 21, No. 16, pp. 3443-3444.

153. **Linton, M.A.** 1984. Implementing relational views of programs. *In*: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. New York: ACM Press, pp. 132-140.

154. **Liu, C., Li, H., Orlowska, M.E.** 1996. Object-Oriented Design of Repository for Enterprise Workflows. CRC for Distributed Systems Technology and Computer Science Department, The University of Queensland, 1996. Retrieved October 5, 2005, from http://www.dstc.uq.edu.au/Research /Distributed_Databases/papers/Liu-OOD-1996.ps

155. **Lloyd, J. W.** 1994. Practical Advantages of Declarative Programming. Invited Lecture, GULP-PRODE '94, Peñiscola, Spain. Retrieved November 08, 2006, from ftp://clip.dia.fi.upm.es/pub/papers/PARFORCE/second_review/D.WP3.1.M2.3.ps.Z

156. **Lopez, O., Laguna, M.A., Garcıa, F.J.** 2002. Reuse based analysis and clustering of requirements diagrams. *In*: the Pre-Proceedings of the Eighth International Workshop on Requirements Engineering: Foundation for Software Quality. pp. 71–82.

157. **Ma, H., Johansson, H., Orsborn, K.** 2005. Distribution and synchronisation of engineering information using active database technology. Advances in Engineering Software. Vol. 36, No. 11-12, November-December 2005, pp. 720-728.

158. **Mahnke, W., Ritter, N.** 2002. The ORDB-based SFB-501-Reuse-Repository. *In*: Proceedings of the 8th International Conference on Extending Database Technology, 25-27 March 2002 Prague, Czech Republic, LNCS Vol. 2287/2002. Germany: Springer Berlin, pp. 745-748.

159. **Marcos, E., Vela, B., Cavero, J.M.** 2001. Extending UML for Object-Relational Database Design. *In*: Proceedings of the 4th international Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, 1-5 October 2001 Toronto, Canada, LNCS Vol. 2185/2001. Germany: Springer Berlin, pp. 225-239.

160. **Marenco, L., Tosches, N., Crasto, C., Shepherd, G., Miller, P.L., Nadkarni, P.M.** 2003. Achieving Evolvable Web-Database Bioscience Applications Using the EAV/CR Framework: Recent Advances. Journal of American Medical Informatics Association, Vol. 10, Sep-Oct 2003, pp. 444–453.

161. **Matjás, L.** 2006. Catalogue of Design Patterns. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering 2006, 28-31 August 2006 Tallinn, Estonia. IOS Press. pp. 139-142.

162. **Mattos, N., DeMichiel, L.G.** 1994. Recent design trade-offs in SQL3. SIGMOD Record, Vol. 23, No. 4, Dec. 1994, pp. 84-90.

163. **McLeod, G.** 2000. Beyond Use Cases. *In*: Proceedings of 5th International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'00) at CAiSE, Stockholm, Sweden.

164. **Melton, J.** 2003. ISO/IEC 9075-2:2003 (E) Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation). August, 2003. Retrieved December 26, 2004, from http://www.wiscorp.com/SQLStandards.html

165. **Melton, J.** 2003b. ISO/IEC 9075-1:2003 (E) Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). July, 2003. Retrieved December 26, 2004, from http://www.wiscorp.com/SQLStandards.html

166. **Melton, J.** 2003c. ISO/IEC 9075-11:2003 (E) Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata). July, 2003. Retrieved December 26, 2004, from http://www.wiscorp.com/SQLStandards.html

167. **Miguel, L., Kim, M.H., Ramamoorthy, C.V.** 1990. A Knowledge and Data Base for Software Systems. *In*: Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence. 6-9 November 1990 Herndon, VA, USA. pp. 417-423.

168. **Mocko, G., Malak Jr, R., Paradis, C., Peak, R.** 2004. A Knowledge Repository for Behavioral Models in Engineering Design. *In*: Proceedings of 24th ASME Computers and Information in Engineering Conference 28 September – October 3 2004 Salt Lake City, Utah.

169. **Mok, W.Y., Ng, Y., Embley, D.W.** 1996. A Normal Form for Precisely Characterizing Redundancy in Nested Relations. ACM Transactions on Database Systems, Vol. 21, No. 1, pp. 77-106.

170. **Montero, M.G., Wright, P.K., Séquin, C.H.** 2002. Managing Complexity in the Design of Electromechanical Products. *In*: Proceedings of the 2002 NSF Design, Service and Mfg. Grantees and Research Conference, Jan. 2002. Retrieved November 10, 2005, from http://www.ifm.eng.cam.ac.uk/mcn/pdf_files/part8_1.pdf

171. **Mühlen, M.** 1999. Evaluation of Workflow Management Systems Using Meta Models. *In*: Proceedings of the 32nd Hawaii International Conference on System Sciences, 5-8 January 1999 Maui, HI, USA, Vol. Track5. pp. 1-11.

172. **Mylopoulos, J., Stanley, M., Wong, K., Bernstein, M., De Mori, R., Ewart, G., Kontogiannis, K., Merlo, E., Müller, H., Tilley, S., Tomic, M.** 1994. Towards an integrated toolset for program understanding. *In*: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 03 1994 Toronto, Ontario, Canada. IBM Press, pp. 48.

173. **Nentwich, C., Emmerich, W., Finkelstein, A., Elmer, E.** 2003. Flexible Consistency Checking. ACM Transactions on Software Engineering and Methodology, Vol. 12, No. 1, pp. 28-63.

174. **Ng, K.W., Ma, J., Nam, G.** 1993. A class library management system for object-oriented programming. *In*: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, 14 – 16 February 1993 Indianapolis, Indiana, US. New York: ACM Press, pp. 445-451.

175. ObjectVenture. Pattern and Component Markup Language. Draft 3. Retrieved June 19, 2003, from http://www.objectventure.net/files/docs/PCMLSpecification.pdf

176. **Oinas-Kukkonen, H., Rossi, G.** 1999. On Two Approaches to Software Repositories and Hypertext Functionality. Journal of Digital Information, Vol. 1, No. 4.

177. OMG Common Warehouse Metamodel Specification formal/03-03-02. March 2003. Version 1.1. Retrieved October 23, 2006, from http://www.omg.org/technology/documents/formal/cwm_mip.htm

178. OMG OCL 2.0 OMG Adopted Specification formal/2006-05-01. Retrieved October 23, 2006, from http://www.omg.org/technology/documents/formal/ocl.htm

179. OMG Reusable Asset Specification. OMG Adopted Specification ptc/04-06-06. Retrieved March 1, 2005, from http://www.omg.org/technology/documents/formal/ras.htm

180. OMG Unified Modeling Language Specification formal/03-03-01. March 2003. Version 1.5.

181. OMG UML 2.0 Superstructure Specification, formal/05-07-04. Retrieved September 25, 2006, from http://www.omg.org/technology/documents/formal/uml.htm

182. **Opdahl, A.L., Henderson-Sellers, B.** 2002. Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model. Software and Systems Modeling, Vol. 1, No. 1, Sep 2002, pp. 43 – 67.

183. Oracle® Database SQL Reference 10g Release 1 (10.1) Part Number B10759-01. Retrieved October 4, 2005, from http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10759/toc.htm

184. **Paige, R. F., Ostroff, J. S.** 2001. The Single Model Principle. *In*: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering. IEEE Computer Society, pp. 292-293.

185. **Pardede, E., Rahayu, J.W., Taniar, D.** 2003. Normalization of Single Level Nested Structure in Object-Relational Data Model. *In*: Proceedings of the International Conference on Informatics, Cybernetics and Systems, Kaoshiung, Taiwan, 2003. IEEE, pp.1884-1889,

186. **Pardede, E., Rahayu, J.W., Taniar, D.** 2004. Mapping Methods and Query for Aggregation and Association in Object-Relational Database using

Collection. *In*: Proceedings of the International Conference on Information Technology: Coding and Computing, 5-7 April 2004, Vol.1. IEEE Computer Society, pp. 539-543.

187. **Pardede, E., Rahayu, J.W., Taniar, D.** 2005. Composition in Object-Relational Database. Encyclopedia of Information Science and Technology, IDEA Publishing, pp. 488-494.

188. **Park, H.C., Lee, W.B., Kim, T.G.** 1994. A relational algebraic framework for models management. *In*: Proceedings of the 26th conference on Winter simulation, 11-14 Dec. 1994. pp. 649-656.

189. **Pascal, F.** 2000. Practical issues in Database Management. A Reference for the Thinking Practitioner. Addison-Wesley.

190. **Pascal, F., Darwen, H., McGoveran, D.** 2005. On POFN and POOD – two complementary database design principles. Retrieved October 01, 2006, from http://www.dbdebunk.com/page/page/3010532.htm

191. **Pedro, L., Lucio, L., Buchs, D.** 2006. Principles for System Prototype and Verification using metamodel based Transformations. *In*: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping, 14-16 June 2006. pp. 10- 17.

192. **Penedo, M.H.** 1987. Prototyping a project master database for software engineering environments. SIGPLAN Not. Vol. 22, No. 1 (Jan. 1987), pp. 1-11.

193. **Pokrajac, D., Patel, H., Rasamny, M.** 2004. Inheritance Constraints Implementation in PostgreSQL. Proc. 48 thETRAN Conference.

194. PostgreSQL 8.0.3 Documentation. Retrieved October 4, 2005, from http://www.postgresql.org/docs/8.0/interactive/index.html

195. **Purao, S.** 1998. APSARA: A Tool to Automate Systems design via Intelligent Pattern Retrieval and Synthesis. The Data Base for Advances in Information Systems – Fall, Vol. 29, No. 4.

196. **Racko, R.** 2004. A Cool Tool Tool. Software Development Magazine, May 2004, Vol. 12, Part 5, pp. 21-26.

197. **Rahayu, W., Chang, E., Dillon, T.S.** 1998. Implementation of Object-Oriented Association Relationships in Relational Databases. *In*: Proceedings of the International Database Engineering and Applications Symposium, 8-10 Jul 1998 Cardiff, UK. IEEE Computer Society, pp. 254-263.

198. **Rahayu, J.W., Taniar, D.** 2002. Preserving Aggregation in an Object-Relational DBMS. *In*: Proceedings of the Second International Conference on Advances in Information Systems, 23-25 October 2002 Izmir, Turkey, LNCS Vol. 2457/2002. Germany: Springer Berlin, pp. 1-10.

199. **Rashid, A., Loughran, N.** 2003. Relational Database Support for Aspect-Oriented Programming. *In*: Proceedings of the International Conference NetObjectDays, 7-10 October 2002 Erfurt, Germany, LNCS Vol. 2591/2003. Germany: Springer Berlin, pp. 233 – 247.

200. **Rasmussen, R.W.** 2005. A framework for the UML meta model. Retrieved March 26, 2005, from http://www.ii.uib.no/~rolfwr/thesisdoc/main1.html

201. **Richters, M., Gogolla, M.** 1999. A Metamodel for OCL. *In*: Proceedings of UML '99: the Unified Modeling Language: beyond the standard, 28-30 October 1999 Fort Collins CO, USA, LNCS Vol. 1723. Germany: Springer Berlin, pp. 156-171.

202. **Richters, M., Gogolla, M.** 2000. Validating UML Models and OCL Constraints. *In*: Proceedings of the Third International Conference UML 2000 - The Unified Modeling Language. Advancing the Standard, October 2000 York, UK, LNCS Vol. 1939/2000. Germany: Springer Berlin, pp. 265-277.

203. **Rising, L.** 2000. The Pattern Almanac 2000. Addison Wesley.

204. **Ritter, N., Steiert, H.P., Mahnke, W., Feldmann, R.,L.** 1999. An Object-Relational SE-Repository with Generated Services. *In* Proceedings of the 1999 Information Resources Management Association International Conference, May 16-19, 1999, Hershey, Pennsylvania, USA. IDEA Group Publications.

205. **Ritter, N., Steiert, H.P.** 2000. Enforcing modeling guidelines in an ORDBMS-based UML-repository. *In*: Proceedings of the 2000 information Resources Management Association International Conference on Challenges of Information Technology Management in the 21st Century, May 2000 Anchorage, Alaska, US. pp. 269-273.

206. **Rittgen, P.** 2006. Translating Metaphors into Design Patterns. Advances in Information Systems Development. Bridging the Gap between Academia and Industry, Vol. 1. Springer. pp. 425-436.

207. **Robertson, S., Robertson, J.** 1999. Mastering the requirements process. Addison-Wesley.

208. **Roost, M., Kuusik, R., Rava, K., Veskioja, T.** 2004 Enterprise Information System Strategic Analysis and Development: Forming Information System Development Space in an Enterprise. *In*: Proceedings of the International Conference on Computational Intelligence, pp. 215-219.

209. **Rossi, M., Brinkkemper, S.** 1996. Complexity Metrics for Systems Development Methods and Techniques. Information Systems, Vol. 21, No. 2, pp. 209-227.

210. **Ruggia, R., Ambrosio, A.P.** 1997. A Toolkit for Reuse in Conceptual Modelling. *In*: Proceedings of the 9th International Conference on Advanced Information Systems Engineering, 16 – 20 June 1997, LNCS Vol. 1250/1997. Germany: Springer Berlin, pp. 173-186.

211. **Sapia, C., Blaschka, M., Höfling, G.** 2000. GraMMi: Using a Standard Repository Management System to Build a Generic Graphical Modeling Tool. *In*: Proceedings of the 33rd Hawaii International Conference on System Sciences, 04 – 07 January 2000. IEEE Computer Society, pp. 10.

212. **Savnik, I., Mohorič, T., Dolenc, T., Novak, F.** 1993. Database model for design data. SIGPLAN OOPS Messenger, Vol. 4, No. 3, Jul. 1993, pp. 26-40.

213. **Seacord, R.C. Hissam, S.A. Wallnau, K.C.** 1998. AGORA: a search engine for software components. IEEE Internet Computing, Vol.2, No.6, Nov/Dec 1998, pp.62-.

214. **Seaman, C., Mendonça, M., Basili, V.R., Kim, Y.M.** 1999. An Experience Management System for a Software Consulting Organization. *In*: Proceedings of the 24th SEL Workshop, Greenbelt, MD, USA.

215. **Seidewitz, E.** 2003. What models mean. IEEE Software, Vol. 20, Issue 5, Sept.-Oct. 2003, pp. 26-31,

216. **Serrano, J.A.** 1999. Formal Specifications of Software Design Methods. *In*: Proceedings of the 3rd Irish Workshop on Formal Methods, 1-2 July 1999 Ireland, Galway.

217. **Seshadri, P.** 1998. Enhanced abstract data types in object-relational databases. The VLDB Journal, Vol. 7, No. 3, pp. 130-140.

218. **Sheth, A.P., Larson, A.J.** 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys, Vol. 22, No. 3.

219. **Siau, K., Cao, Q.** 2002. How complex is the unified modeling language? In Advanced Topics in Database Research Vol. 1, pp. 294-306.

220. **Siau, K., Rossi, M.** 1998. Evaluation of Information Modeling Methods - A Review. *In*: Proceedings of the 31st Annual Hawaii International Conference on System Sciences, Vol. 5. USA: IEEE Computer Society, pp. 314-322.

221. **Sidle, T.W.** 1980. Weaknesses of commercial data base management systems in engineering applications. *In*: Proceedings of the 17th Conference on Design Automation 23 – 25 June 1980 Minneapolis, Minnesota, US. New York: ACM Press, pp. 57-61.

222. **Silverston, L.** 2001. The Data Model Resource Book: A Library of Universal Data Models for All Enterprises. Vol. 1. Wiley Computer Publishing.

223. **Singh, H., Han, J.** 1996. Requirements for Object Management in Software Engineering Environments. Technical Report 96-02, Peninsula School of Computing, Monash University, Melbourne, Australia, February 1996. Retrieved July 24, 2005, from http://citeseer.ist.psu.edu/113414.html

224. **Skatulla, S., Dorendorf, S.** 2003. Optimization of Storage Structures of Complex Types in Object-Relational Database Systems. Advances in Databases and Information Systems, LNCS Vol. 2798/2003. Germany, Springer Berlin. pp. 220-235.

225. **Smith, J.M., Smith, D.C.** 1977. Database abstractions: aggregation, Communications of the ACM, Vol. 20, No. 6, June 1977, pp. 405-413.

226. **Sneed, H., Dombovari, T.** 1999. Comprehending a complex, distributed, object-oriented software System - a Report from the Field. *In*:

Proceedings of the Seventh International Workshop on Program Comprehension, 5-7 May 1999. Pittsburgh: IEEE Computer Society Press, pp. 218-225.

227. **Soutou, C.** 2001. Modeling relationships in object-relational databases. Data and Knowledge Engineering, Vol. 36, Issue 1, pp. 79-107.

228. **Stonebraker, M., Rowe, L. A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., Beech, D.** 1991. Third-generation database system manifesto. Comput. Stand. Interfaces. Vol. 13, No.1-3 (Oct. 1991), pp. 41-54.

229. **Straeten, R., Mens, T., Simmonds, J., Jonckers, V.** 2003. Using Description Logic to Maintain Consistency between UML Models. "UML" 2003 - The Unified Modeling Language, LNCS Vol. 2863/2003. Germany: Springer Berlin, pp. 326-340.

230. **Szykman, S. Sriram, R. D. Bochenek, C. Racz, J. W. Senfaute, J.** 2000. Design Repositories: Engineering Design's New Knowledge Base IEEE Intelligent Systems and their Applications, Vol. 15, No. 3., pp. 48-55.

231. Systematik holistik metodik. Retrieved March 5, 2006, from http://www.systematik.se/

232. **Taylor, R.N., Belz, F.C., Clarke, L.A., Osterweil, L., Selby, R.W., Wileden, J.C., Wolf, A.L., Young, M.** 1988. Foundations for the Arcadia environment architecture. *In*: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 28 – 30 November 1988, Boston, Massachusetts, US. New York: ACM Press, pp. 1-13.

233. **Tombros, D., Geppert, A.** 1995. A survey of database support for process centered software development environments, Technical report 95.28, Universität Zürich. Retrieved October 18, 2006, from http://historical.ncstrl.org/litesite-data/unizh_ifi/ifi-95.28.ps.gz

234. **Tsai, Y.C.** 2001. Comparative analysis of model management and relational database management. Omega, Vol. 29, No. 2, pp. 157–170.

235. **Turns, J. Atman, C. J. Adams, R.** 2000. Concept Maps for Engineering Education: A Cognitively Motivated Tool Supporting Varied Assessment Functions. IEEE Transactions on Education. Vol. 43; Part 2, pp 164-173.

236. **Türker, C., Gertz, M.** 2001. Semantic integrity support in SQL:1999 and commercial (object-) relational database management systems. The VLDB Journal, Vol. 10, No. 4, pp. 241–269.

237. **Vincent, M.W.** 1998. Redundancy Elimination and a New Normal Form for Relational Database Design. Semantics in Databases, LNCS Vol. 1358/1998. Germany: Springer-Verlag, pp. 247-264.

238. **Virvou, M., Tourtoglou, K.** 2006. Intelligent Help for Managing and Training UML Software Engineering Teams. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering, 28-31 August 2006 Tallinn, Estonia. IOS Press, pp. 11-20.

239. **Voorish, D.** 2005. An Implementation of Date and Darwen's "Tutorial D". Retrieved March 26, 2005, from http://dbappbuilder.sourceforge.net/Rel.html

240. **Võhandu, L., Kuusik, R., Torim, A., Aab, E., Lind, G.** 2006. Some Monotone Systems Algorithms for Data Mining. WSEAS Transactions on Information Science & Applications. Vol. 3, Issue 4, pp. 802-809.

241. **Wang, S.A., Yang, F., Huey, C., Pecjak, F., Upender, B., Frazin, A., Lingam, R., Chintala, S., Wang, G., Kellog, M., Martino, R.L., Johnson, C.A.** 2004. Performance of using Oracle XMLDB in the evaluation of CDISC ODM for a clinical study informatics system. *In*: Proceedings of the 17th IEEE Symposium on Computer-Based Medical Systems. pp. 594- 599.

242. **Weikum, G., Vossen, G.** 2002. Transactional information systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. USA: Morgan Kaufman Publishers, Academic Press.

243. **Wurden, F.L.** 1997. Content Is King (If You Can Find It): A New Model for Knowledge Storage and Retrieval. *In*: Proceedings of the 13th International IEEE Conference on Data Engineering 7-11 April 1997. pp. 149-157.

244. **Yen, I.L., Khan, L., Prabhakaran, B., Bastani, F.B., Linn, J.** 2001. An On-line Repository for Embedded Software. *In*: Proceedings of the 13th IEEE international Conference on Tools with Artificial Intelligence, 07 – 09 November 2001. pp. 314-321.

245. **Zachman, J.A.** 1987. A framework for information systems architecture. IBM Systems Journal, Vol. 26, No. 3, pp. 276 – 292.

246. **Zhang, N., Ritter, N., Härder, T.** 2001. Enriched Relationship Processing in Object-Relational Database Management Systems. *In*: Proceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications, 23-24 April 2001 Beijing, China. pp. 50-59.

247. **Zhang, Z., Lyytinen, K.** 2001. A Framework for Component Reuse in a Metamodelling-Based Software Development. Requirements Engineering, Vol. 6, Part 2, pp. 116 – 131.

248. **Zimbrão, G., Miranda, R., Souza, J.M., Estolano, M.H., Neto, F.P.** 2003. Enforcement of Business Rules in Relational Databases Using Constraints. XVIII Simpósio Brasileiro de Banco de Dados - 2003 - Manaus, AM, Brasil. pp. 129-141.

APPENDIX A: SOME PROPERTIES OF EXISTING SOFTWARE ENGINEERING SYSTEMS THAT USE THE HELP OF A DBMS

Reference	DBMS type	Development phase where the content is used			Type of content					Location of content	
		Analysis	Design	Implement	models	project data	experiences	code	inf. about code	in DB	outside DB
Linton (1984)	RDBMS			+				+		+	
Ng et al. (1993)	RDBMS			+					+	+	
Mylopoulos et al. (1994)	RDBMS			+					+	+	
Gray (1997)	RDBMS		+		+					+	
Keqin et al. (1997)	RDBMS			+					+	+	
Blaha et al. (1998)	RDBMS	+	+		+					+	
Purao (1998)	RDBMS		+		+		+			+	
Gruhn and Schneider (1998)	RDBMS	+	+		+					+	
Seaman et al. (1999)	RDBMS	+	+	+			+			+	
Sneed and Dombovari (1999)	RDBMS	+	+	+	+					+	
Henninger (2001)	RDBMS	+	+	+			+			+	
Lopez et al. (2002)	RDBMS	+			+					+	
Rashid and Loughran (2003)	RDBMS			+				+	+	+	
Chisholm (2005)	RDBMS	+			+					+	
Lavazza and Agostini (2005)	RDBMS	+	+		+					+	
Penedo (1987)	RDBMS	+	+	+		+				+	+
Chen et al. (1990)	RDBMS			+				+	+	+	+

Appendix A continued

Reference	DBMS type	Development phase where the content is used			Type of content					Location of content	
		Analysis	Design	Implement	model	project data	experiences	code	inf. about code	in DB	outside DB
Lejter et al. (1992)	RDBMS			+				+	+	+	+
Boisvert (1994)	RDBMS			+				+	+	+	+
Behle (1998)	RDBMS			+					+	+	+
Seacord et al. (1998)				+				+	+	+	+
Boldyreff et al. (2002)		+	+	+	+	+	+	+		+	+
Conte et al. (2004)			+		+			+		+	+
Ambriola et al. (1997)	EDBMS	+	+		+					+	
Cox et al. (1999)	EDBMS			+				+	+	+	
Ambriola et al. (1997)	OODBMS	+	+		+					+	
Liu et al. (1996)	OODBMS		+		+					+	
Keller et al. (2001)	OODBMS		+		+					+	
Miguel et al. (1990)	ORDBMS			+				+		+	
Althoff et al. (1999)	ORDBMS	+	+	+			+			+	+
Yen et al. (2001)	ORDBMS			+				+		+	+
Allsop et al. (2002)	ORDBMS	+			+					+	
Mahnke and Ritter (2002)	ORDBMS	+	+	+			+			+	
Kovse et al. (2002)	ORDBMS		+		+					+	
Ritter and Steiert (2000)	ORDBMS	+	+		+					+	

APPENDIX B: SOME SECONDARY CHARACTERISTICS OF WHOLE-PART RELATIONSHIPS

Type of secondary characteristic	Value of sec. characteristic	Description of the value of the secondary characteristic
Shareability: "the ability of the part to belong to two or more wholes at the same time" (Henderson-Sellers and Barbier, 1999)	Locally exclusive part	A part object is related through a type of whole-part relationship to at most one whole object. ([W]-<>-**..1**------------[P])
	Locally shareable part	A part object can be related through a type of whole-part relationship to more than one whole object. ([W]-<>-**..n**------------[P] **n>1**)
	Globally exclusive part	A part object type cannot be related through whole-part relationship types to more than one whole object type. ([W]-<>-------------[P])
	Globally shareable part	A part object type can be related through whole-part relationship types to more than one whole object type. ([W]-<>-----------[P]-----------<>-[W'])
Existential dependency: the existence of part/whole objects depends on the existence of whole/part objects.	Essential part	A whole object must have the specific associated part object(s) and is existentially dependent on it (them) (Guizzardi, 2005, p. 343). ([W]-<>-----**1..**-[P])
	Inseparable part	A part object cannot be disconnected (separated) from the whole object(s) and is existentially dependent on it (them) (Guizzardi, 2005, p. 343). ([W]-<>-**1..**------[P])
Separability: "piece(s) can be removed from the whole without destroying either" (Henderson-Sellers and Barbier, 1999)	Mandatory whole	A part object must be associated with a whole object (but not with any specific object). ([W]-<>-**m..**------[P] **m>=1**)
	Mandatory part	A whole object must be associated with a part object (but not with any specific object). ([W]-<>-------**n..**-[P] **n>=1**)
	Optional whole	A part object can be disconnected from a whole object and does not have to have associated whole object. ([W]-<>-**0..**------------[P])
	Optional part	A whole object can be disconnected from a part object and does not have to have associated part object. ([W]-<>-------------**0..**-[P])

APPENDIX C: COMPARISON OF SOME SYSTEMS THAT RECORD MODELS IN A DATABASE

	System analysis environment (proposed in Chapter 4)	UML Repository (Ritter and Steiert, 2000)	UML Model Measurement Tool (Lavazza and Agostini, 2005)
Purpose of the system	Allows us to create a system specification and to validate it.	Allows us to record UML models in order to later reuse them, analyse them, share them between developers and generate code based on them.	Allows us to record UML models in order to find metrics values.
Metamodel	Repository schema is based on the metamodel of a language that is specifically worked out in order to support methodological framework for the Enterprise Information System (EIS) strategic analysis (Roost et al., 2004).	Repository schema is based on UML metamodel.	Repository schema is based on *simplified* UML metamodel (supports subset of class and state models).
DBMS type	$\text{ORDBMS}_{\text{SQL}}$	$\text{ORDBMS}_{\text{SQL}}$	$\text{RDBMS}_{\text{SQL}}$
The use of OR_{SQL} specific solutions - typed tables and table inheritance	No	Yes	No

	System analysis environment (proposed in Chapter 4)	UML Repository (Ritter and Steiert, 2000)	UML Model Measurement Tool (Lavazza and Agostini, 2005)
The use of database constraints	Uses minimal amount of database constraints – only primary and foreign keys. Uses queries in order to find violations of consistency and completeness rules.	Uses CHECK constraints and triggers in order to preserve the consistency of UML models and enforce design rules and guidelines.	Does not mention the use of database constraints. Probably does not use them because the system must be able to calculate metrics values based on any model – correct or incorrect.
Constraint checking on demand	Yes	Yes	Does not mention the use of database constraints.
Source of constraints	Worked out by us	OCL invariants that accompany UML metamodel, global design guidelines and process-related design rules.	Does not mention the use of database constraints.
Built-in queries	Yes	Does not mention	Yes
User-defined queries	System users can define new queries by using SQL.	System users can specify the constraints by using OCL. System converts them to SQL.	System users can define new queries by using SQL.
User can specify queries which find metrics values.	Yes	Does not mention	Yes
User-interface for accessing models	Custom-built web-based and form based interface.	Database administration tool	Database administration tool

APPENDIX D: THE LOCATION OF PROTOTYPE SYSTEM

Chapter 4 presents the system that can be used in order to create a system analysis specification, check its consistency and completeness and calculate metrics values.

We have implemented a partial prototype of this system. This web-based system can be found from: http://viktor.ld.ttu.ee/modeler/index.php

This prototype is created by using PHP language and uses the help of ORDBMS$_{SQL}$ PostgreSQL 8.0.4. The prototype implements partially two functional subsystems of our system – *query management* and *subsystems management*. It is possible to register new users, which is functionality of the *user management* subsystem. It is also possible to specify names of the use cases, which is functionality of the *scenario management* subsystem.

The prototype was developed in collaboration with my student Erko Aaberg who created it as part of his bachelor thesis. The author of this dissertation has created the following parts of the *prototype*:

- Part of the query subsystem that allows us to manage and execute detailed CCC checks, metrics queries and general queries. The author of this dissertation has also worked out these queries.
- IS_EMPTY scalar operator that is used by many queries. We have implemented it by using PL/pgSQL language.

A *query manager* can manage (create/read/update/delete) queries, including their SQL statements. A *system viewer* can see specifications, execute queries and see the results. A *system describer* can see and modify the specifications, execute queries and see the results.

The database contains partial specification of an IS in order to make possible the testing of these queries.

APPENDIX E: CURRICULUM VITAE

1. Personal Data

Name:	Erki Eessaar
Date of birth and place:	09.03.1977, Tallinn, Estonia
Citizenship:	Estonian
Maritual status:	unmarried
Children:	-

2. Contact Data

Address:	Raja 15-409, Tallinn, 12618
Phone:	+372 6202306 (at work)
E-mail:	eessaar@staff.ttu.ee;Erki.Eessaar@mail.ee

3. Education

Educational Institution	*Graduation time*	*Speciality / grade*
Tallinn Technical University	1999	Informatics / Bachelor of technical science
Tallinn Technical University	2001	Informatics / Master of technical science

4. Language Skills (basic, intermediate or high level)

Language	*Level*
Estonian	High Level (mother tongue)
English	High Level
Russian	Intermediate Level

5. Special courses: -

6. Professional employment

Period	*Institution*	*Position*
9/1999-8/2002	Tallinn Technical University, Institute of Informatics	Assistant
9/2002-	Tallinn University of Technology (former Tallinn Technical University), Institute of Informatics	Lecturer

7. Scientific Work

Eessaar, E. 2002. Patterns as Reusable Fragments of Knowledge. *In*: Proceedings of the Fifth International Baltic Conference on Databases and Information Systems, 3-6 June 2002 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 243 - 248.

Eessaar, E. 2004a. Towards Pattern Management System. *In*: Proceedings of the Sixth International Conference on Enterprise Information Systems, 14 – 17 April 2004 Porto, Portugal. Vol. 3. pp. 655 – 658.

Eessaar, E. 2004b. Methods for Searching Patterns from the Database of Patterns. *In*: The 16th Conference on Advanced Information Systems Engineering Forum Proceedings, 7-11 June 2004 Riga, Latvia. pp. 103 – 111.

Eessaar, E. 2005a. Truly Relational Databases as a Platform for the Artifact Management. *In*: Proceedings of the Fourteenth International Conference on Information Systems Development: Pre-Conference 14-17 August 2005 Karlstad, Sweden. pp. 207-218.

Eessaar, E. 2005b. Architecture of Pattern Management Software System. *In*: Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems, 12-15 September 2005 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 189-207.

Eessaar, E. 2006a. Extended Principle of Orthogonal Database Design. *In*: Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, 15-17 February 2006 Madrid, Spain. pp. 360-365. CD-ROM.

Eessaar, E. 2006b. Guidelines about Usage of the Complex Data Types in a Database. WSEAS Transactions on Information Science and Applications, Vol. 3, Issue 4, April 2006, pp. 712-719.

Eessaar, E. 2006c. Using Relational Databases in the Engineering Repository Systems. *In*: Proceedings of the Eighth International Conference on Enterprise Information Systems, 23 –27 May 2006 Paphos, Cyprus. Vol. Databases and Information Systems Integration. pp. 30 – 37.

Eessaar, E. 2006d. Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 10th WSEAS International Conference on COMPUTERS, 13-15 July 2006 Vouliagmeni, Athens, Greece. pp. 1263-1268. CD-ROM.

Eessaar, E. 2006e. SQL or Third Manifesto Compliant Object-Relational Database Management Systems as the Platforms for Maintaining the Whole-Part Relationships in a Database. WSEAS Transactions on Computers, Vol. 5, Issue 10, October 2006, pp. 2440-2447.

Eessaar, E. 2006f. Integrated System Analysis Environment for the Continuous and Completeness Checking. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering 2006, 28-31 August 2006 Tallinn, Estonia. IOS Press. pp. 96-105.

Eessaar, E. 2006g. Preserving Semantics of the Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 15th International Conference on Information Systems Development, August 31 - September 2 2006 Budapest, Hungary. Springer. (forthcoming).

8. Theses Accomplished and Defended

   B. Sc. Thesis (1999): Strategic Analysis of Information System for Managing Data About the Estonian Repressed Persons.

   M. Sc. Thesis (2001): Pattern Based Development of the System that Assists Usage of Patterns.

9. Research Interests: Data models, database design, repositories, Meta-CASE, metamodeling, patterns.

10. Research projects: -

Signature: Date: 14.11.2006

APPENDIX F: ELULOOKIRJELDUS (CV IN ESTONIAN)

1. Isikuandmed

Ees- ja perekonnanimi:	Erki Eessaar
Sünniaeg ja –koht:	09.03.1977, Tallinn, Eesti
Kodakondsus:	Eesti
Perekonnaseis:	vallaline
Lapsed:	puuduvad

2. Kontaktandmed

Aadress:	Raja 15-409, Tallinn, 12618
Telefon:	+372 6202306 (tööl)
E-posti aadress:	eessaar@staff.ttu.ee;Erki.Eessaar@mail.ee

3. Hariduskäik

Õppeasutus (nimetus lõpetamise ajal)	*Lõpetamise aeg*	*Haridus (eriala/kraad)*
Tallinna Tehnikaülikool	1999	Informaatika / tehnikateaduste bakalaureus
Tallinna Tehnikaülikool	2001	Informaatika / tehnikateaduste magister

4. Keelteoskus (alg-, kesk- või kõrgtase)

Keel	*Tase*
Eesti	Kõrgtase (emakeel)
Inglise	Kõrgtase
Vene	Kesktase

5. Täiendõpe: -

6. Teenistuskäik

Töötamise aeg	*Ülikooli, teadusasutuse või muu organisatsiooni nimetus*	*Ametikoht*
9/1999-8/2002	Tallinna Tehnikaülikool, Informaatikainstituut	Assistent
9/2002-	Tallinna Tehnikaülikool, Informaatikainstituut	Lektor

7. Teadustegevus

Eessaar, E. 2002. Patterns as Reusable Fragments of Knowledge. *In*: Proceedings of the Fifth International Baltic Conference on Databases and Information Systems, 3-6 June 2002 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 243 - 248.

Eessaar, E. 2004a. Towards Pattern Management System. *In*: Proceedings of the Sixth International Conference on Enterprise Information Systems, 14 – 17 April 2004 Porto, Portugal. Vol. 3. pp. 655 – 658.

Eessaar, E. 2004b. Methods for Searching Patterns from the Database of Patterns. *In*: The 16th Conference on Advanced Information Systems Engineering Forum Proceedings, 7-11 June 2004 Riga, Latvia. pp. 103 – 111.

Eessaar, E. 2005a. Truly Relational Databases as a Platform for the Artifact Management. *In*: Proceedings of the Fourteenth International Conference on Information Systems Development: Pre-Conference 14-17 August 2005 Karlstad, Sweden. pp. 207-218.

Eessaar, E. 2005b. Architecture of Pattern Management Software System. *In*: Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems, 12-15 September 2005 Tallinn, Estonia. Tallinn: Institute of Cybernetics at Tallinn University of Technology, pp. 189-207.

Eessaar, E. 2006a. Extended Principle of Orthogonal Database Design. *In*: Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, 15-17 February 2006 Madrid, Spain. pp. 360-365. CD-ROM.

Eessaar, E. 2006b. Guidelines about Usage of the Complex Data Types in a Database. WSEAS Transactions on Information Science and Applications, Vol. 3, Issue 4, April 2006, pp. 712-719.

Eessaar, E. 2006c. Using Relational Databases in the Engineering Repository Systems. *In*: Proceedings of the Eighth International Conference on Enterprise Information Systems, 23 –27 May 2006 Paphos, Cyprus. Vol. Databases and Information Systems Integration. pp. 30 – 37.

Eessaar, E. 2006d. Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 10th WSEAS International Conference on COMPUTERS, 13-15 July 2006 Vouliagmeni, Athens, Greece. pp. 1263-1268. CD-ROM.

Eessaar, E. 2006e. SQL or Third Manifesto Compliant Object-Relational Database Management Systems as the Platforms for Maintaining the Whole-Part Relationships in a Database. WSEAS Transactions on Computers, Vol. 5, Issue 10, October 2006, pp. 2440-2447.

Eessaar, E. 2006f. Integrated System Analysis Environment for the Continuous and Completeness Checking. *In*: Proceedings of the Joint Conference on Knowledge-Based Software Engineering 2006, 28-31 August 2006 Tallinn, Estonia. IOS Press. pp. 96-105.

Eessaar, E. 2006g. Preserving Semantics of the Whole-Part Relationships in the Object-Relational Databases. *In*: Proceedings of the 15th International Conference on Information Systems Development, August 31 - September 2 2006 Budapest, Hungary. Springer. (forthcoming).

8. Kaitstud lõputööd

   Bakalaureusetöö (1999): Eesti Represseeritute Registri haldamise infosüsteemi strateegiline analüüs.

   Magistritöö (2001): Mustrite kasutamist abistava süsteemi mustritel põhinev projekteerimine.

9. Teadustöö põhisuunad: Andemudelid, andmebaasi disain, teadmusbaasid, Meta-CASE, metamodelleerimine, mustrid.

10. Teised uurimisprojektid: -

Allkiri: Kuupäev: 14.11.2006