TECHNISCHE UNIVERSITÄT DRESDEN

Faculty Computer Science Institute of Software and Multimedia Engineering, Chair of Software Technology

A SEMI-AUTOMATIC TRANSFORMATION FROM OBJECT-ORIENTED SYSTEMS TO ROLE-BASED SYSTEMS

Kay Bierzynski
Born on: 17th December 1990 in Wriezen
Matriculation number: 3668264
Matriculation year: 2013

MASTER THESIS

to achieve the academic degree

MASTER OF SCIENCE (M.SC.)

Supervisor
Dipl.-Inf. Thomas Kühn
Dipl.-Wirt.-Inf. Tobias Jäkel
Supervising professor
Prof. Dr. rer. nat. habil. U. Aßmann

Submitted on: 22nd September 2015
Defended on: 24th September 2015

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *A Semi-Automatic Transformation from Object-Oriented Systems to Role-Based Systems* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 22nd September 2015

Kay Bierzynski

Abstract

Current software systems have to cope with many different problems like increased complexity or changes in their application environment. To deal with these a demand for new concepts beside object-oriented programming has arisen. One of the new concepts are roles. Many groups and people contributed to the investigation of the concept of roles and one of them is the research-training group RoSI.
This group has as a focus the investigation of role-based information systems. In particular, the extension of common data models with new features, like Roles, Relationships and Compartments, that are used to capture dynamic and complex application domains. The first results in this regard could already be presented through the developments of a new conceptual modelling language that is called Compartment Role Object Model (CROM) and a query language with the name Role Structured Query Language (RSQL). Furthermore, there exist a prototypical Database Management System (DBMS) supporting RSQL.
The problem is that no benchmark exists to evaluate the prototypical DBMS or role-based information systems in general. Besides this, the incorporation of the research results like CROM and RSQL into existing software is a problem, as well. To solve these problems is is the goal of this work. Hence, to realize this goal a transformation tool is introduced that changes an object-oriented information system like a benchmark to an information system that is based on RSQL or CROM.
The introduced tool is composed of a model transformation process that can take an arbitrary input and a model-driven benchmark transformation process. Both processes are based on an investigation of five benchmark schemata. Moreover, the investigated benchmarks are SmallBank, TPC-C, TPC-H, AuctionMark and SEATS and these were transformed with the transformation tool to test its functionality.

Contents

Statement of authorship iii

Abstract iv

1 Introduction 1

2 Preliminaries 3
2.1 Paradigms . . . 3
2.1.1 Object-Oriented Paradigm . . . 3
2.1.2 Relational Paradigm . . . 3
2.1.3 Role-Oriented Paradigm . . . 4
2.2 CROM . . . 5
2.3 RSQL . . . 5
2.4 Model Transformation . . . 7
2.4.1 Overview . . . 7
2.4.2 Model-Driven Software Development . . . 8
2.4.3 GreGen.NET . . . 8
2.4.4 Kermeta . . . 9
2.4.5 QVT-R . . . 9
2.4.6 ATL . . . 9
2.4.7 UML-RSDS . . . 10
2.5 Benchmarking . . . 10
2.6 Metrics . . . 11

3 Investigation and Transformation of Benchmarks 13
3.1 Model Complexity Formula . . . 13
3.2 SmallBank . . . 15
3.3 SEATS . . . 17
3.4 TPC-C . . . 19
3.5 TPC-H . . . 21
3.6 AuctionMark . . . 22
3.7 Overview of the Needed Informations and Schema Complexities . . . 24

4 Theoretical System Transformation 27
4.1 Model Transformation . . . 27
4.1.1 Annotations . . . 27
4.1.2 Transformation Process . . . 29
4.1.3 Example SmallBank . . . 31
4.2 Benchmark Transformation . . . 33
4.2.1 Schema Transformation . . . 33
4.2.2 Data and Query Transformation . . . 34
4.2.3 Statement and Query Transformation Examples . . . 36

4.3 Limitations 38

5 Implementation of the System Transformation 39
5.1 Architecture 39
5.2 Parsing Component 40
5.3 Annotation Component 41
5.4 Transformation Component 41
5.5 Limitations 42

6 Related Work 43
6.1 Requirements 43
6.2 Overview of the Related Works 44
6.3 Comparison 45

7 Conclusion and Future Work 49

Index of Abbreviations 51

Appendix 57

List of Figures

2.1 Dynamic Data Type Person 6
2.2 Configurations of the DDT Person 6

3.1 SQL Schema of SmallBank 14
3.2 NTRT Schema of SmallBank 15
3.3 RSQL Schema of SmallBank 16
3.4 CROM Schema of SmallBank 17
3.5 CROM Basic Schema of SEATS 19
3.6 CROM Basic Schema of TPC-C 20
3.7 CROM Basic Schema of TPC-H 22
3.8 Model Complexity Diagram 25

4.1 Steps of the First Phase 30
4.2 Steps of the Second Phase 30
4.3 Annotated SQL Model of SmallBank 32
4.4 Annotated NTRT Model of SmallBank 33
4.5 Steps of the Schema Transformation 34
4.6 Steps of the First Data Transformation Variant 34
4.7 Steps of the Query Transformation 35

5.1 Architecture of the Transformation Tool 40

1 SQL Schema of SEATS 57
2 NTRT Schema of SEATS 58
3 RSQL Schema of SEATS 59
4 Content of the Compartment Type Flight(SEATS) 60
5 SQL Schema of TPC-C 61
6 NTRT Schema of TPC-C 62
7 RSQL Schema of TPC-C 63
8 Content of the Compartment Type Company(TPC-C) 64
9 Content of the Compartment Type Order(TPC-C) 64
10 SQL Schema of TPC-H 65
11 NTRT Schema of TPC-H 66
12 RSQL Schema of TPC-H 67
13 Content of the Compartment Type Business(TPC-H) 68
14 SQL Schema of AuctionMark 69
15 NTRT Schema of AuctionMark 70
16 RSQL Schema of AuctionMark 71
17 CROM Basic Schema of AuctionMark 72
18 Content of the Compartment Type System(AuctionMark) 73
19 Content of the Compartment Type Auction(AuctionMark) 74
20 Content of the Compartment Type Global Information(AuctionMark) 74

List of Tables

6.1 Comparison of Transformation Processes 46

1 Introduction

Current software systems have to cope with many different problems like increased complexity or changes in their context. To deal with these, a demand for new concepts beside object-oriented programming has arisen. One of the new concepts is role-based programming. Many groups and people contributed to the investigation of the concept of role-based programming and one of them is the research-training group RoSI [4].
The main focus of this group is the investigation of role-based information systems. In particular, the extension of common data models with new features, like Roles, Relationships and Compartments, that are used to capture dynamic and complex application domains. The first results in this regard could already be presented through the developments of a new conceptual modelling language that is called Compartment Role Object Model (CROM) [3] and a query language with the name Role Structured Query Language (RSQL) [5].
All in all, it is to assume that in some years a paradigm shift of software will happen depending on the result of the research on roles and the utility they can provide in practical areas. Hence, the goal of this work is to specify and implement a process to change the paradigm of a software application or a software system from object-oriented to role-based. To do so ways must be specified to shift the paradigm. A very simple way to change the paradigm of software is to develop and implement it from scratch. However, this would be very inefficient and time consuming, since the applications and devices, like cars and weather apps used in the everyday life, incorporate trillions of lines of code. Another problem is that many software applications and parts of software systems are based on the procedural paradigm instead of the object-oriented paradigm. So it is better to transform software systems to reuse as much of their code base as possible.
When a transformation is defined two decisions need to be made. The first deals with the question whether all parts of a software system should be transformed or only specific parts, like the architectural model or the Database Management System (DBMS). The second focuses on how the transformation is executed. For example, it is possible to make a one-to-one transformation where everything is directly converted to the new paradigm. Another approach is to do a model-driven transformation approach. In particular, such approaches first transform the model of an application or a system and than based on that model, transform or implement the rest.
To reach this goal, this work specifies and implements a semi-automatic model-driven transformation tool. The advantage of such a tool is that software can be transformed and evolved by the information the user can add in the scope of manual parts of the transformation process. Furthermore, the usage of a model-driven approach makes it easier to automate the transformation of all software system parts with the exception of models. Moreover, to reuse the research and to incorporate this work in the investigation of roles, the tool transforms an input model to a CROM model or a RSQL model. This makes it possible to transform DBMSs, as well as other parts of a software system with the transformation tool of this work.
The rest of the work is structured based on the main task as follows. In the next chapter the preliminaries of this work will be described. In the third chapter the benchmark schemata of SmallBank [19], SEATS [20], TPC-C [21], TPC-H [22] and AuctionMark [25] are investigated

to learn how a model changes when it is transformed to CROM or RSQL. Moreover, the informations necessary for the transformation are gathered and the possible degree of automatization is determined during the investigation, as well. The results of the investigation are used in the fourth chapter to specify a model transformation process and a model-driven benchmark transformation process as an example for the software system transformation. These processes are the base of the transformation tool and as such their implementation is explained in chapter five. Furthermore, in the scope of the 6 chapter related works are introduced and compared with this works. Last but not least, the 7 chapter deals with the conclusion and future works.

2 Preliminaries

For the development and evaluation of the transformation tool a wide variety of concepts is necessary. These concepts are introduced in this chapter. Hence, the structure of the rest of the chapter is as follows: The first section focus on the object-oriented paradigm, relational paradigm and the role-based paradigm. In the second section the Compartment Role Object Model (CROM) is introduced and Role Structured Query Language (RSQL) is the topic of the third section. A description of model transformation and several approaches are presented in the fourth section. The content of the next section is a description of the concept of benchmarks. Last but not least, the last section deals with the concept of metrics and some specific metrics that are used to evaluate parts of the work.

2.1 Paradigms

In this chapter the object-oriented paradigm, relational paradigm and the role-based paradigm are introduced. These paradigms are the most important basic concepts of the transformation tool and it is necessary to understand their differences to understand the relevance of the tool.

2.1.1 Object-Oriented Paradigm

The object-oriented paradigm is one of the approaches that are used in computer science for modelling and programming. The core elements of this paradigm are objects.
The term object describes in computer science a position in the memory having a value and possibly referenced by an identifier. An object can be a variable, a data structure or a function. In object-oriented (OO) programming theses different kinds of objects are combined and interactions between them are defined to design computer programs and systems. [47] Many different types of programming languages exists in the context of OO programming, but some of the most commonly used and popular programming languages like Java and C# are based on classes. In computer programming a class is an extensible program-code-template [48], that provides the initial values for variables and the implementations of functions and in computer modelling it is a design, which defines the structure and behaviour of all objects of a specific type. Hence, when a developer wants to create a program for example in Java he must define classes and relate them. Than when the program is executed instances of classes also objects are created.

2.1.2 Relational Paradigm

The relational paradigm is based on the concept of *relation* and primarily used for databases. Relation is a well-defined term from mathematics. In the context of databases a relation is the mathematically description of a table. Hence, operations on the table of a relational

database are specified by relational Algebra. Consequently, relational Algebra is the theoretical basis of SQL and must be considered by the transformation tool of this work when a relational database is transformed.
Relational databases don't have the concept of objects, but today in general programmers and designers treat the data in a database as objects [49]. Furthermore, since a relational databases can't directly handle objects a bridge must be created to use it in an object-oriented system. When the developer creates such a bridge he must consider the *object-relational impedance mismatch*. This term describes a set of conceptual and technical difficulties that are often encountered when an object-oriented program is used in combination with a relational database. Hence, the *object-relational impedance mismatch* is for the transformation tool a problem when an object-oriented system is completely transformed.

2.1.3 Role-Oriented Paradigm

For the role-oriented paradigm a standardized definition in computer science doesn't exist to this date. In this thesis the definition of Friedrich Steimann [1] for roles is used, because he constructed it on the basis of many preceding definitions. Moreover, Compartment Role Object Model (CROM) and Role Structured Query Language (RSQL) are based on this definition. As they are important parts of the transformation the definition is for this thesis fundamental. Before the definition of Steinmann can presented the terms *rigidity* and *foundedness* must explained since they are used in it. The first describes that instances of a rigid type belong to that type until they cease to exist and the second denotes that instances of a founded type must always exist in collaboration with another object.
Steimann ontologically classifies roles with the properties *anti-rigid* and *founded*. The first of the two is the reverse of rigidity. Hence, a instance of an anti-rigid type can dynamically start and stop belonging to the type without ceasing to exist. An example for such a type would be a reader, because an instance can start and stop being a reader without losing its identity or existence. The second property of a role would mean in the case of the reader that an instance of a reader can only exist when it is associated with an object person and an object, which can be read.
Steimann defines the collaboration objects or players of a role as naturals and he denotes them in contrast to roles with the properties *rigid* and *non-founded*. Thus, naturals can exist on their own without a context or other entities and are never played by another object. Furthermore, a natural can play arbitrary many roles and more than one role at a time, when the context and constraints allow it. Despite these facts, a role instance can only be played by exactly one natural instance, but a role type can be fulfilled by many natural types, when their types fit in the context. An example for that would be that the natural types person and computer can fulfil the role reader, but an instance of a reader can only be played by an instance of the type person or computer.
Moreover, it is to note that only a few programming languages directly support roles, but the concept of roles where realized in many software system by means of object-oriented languages. This has led to the best practice in form of the Role Object Pattern, that was identified by Bäumer et.al. [2].

2.2 CROM

The Compartment Role Object Model (CROM) [3] is a role-based modelling language that was developed by Thomas Kühn et al and an important part of the transformation tool of this work. An instance of a CROM is called Compartment Role Object Instance (CROI). Furthermore, they were developed due the fact that none of the prior role-based modelling languages sufficiently incorporates the relational and context-dependent nature of roles together with the constraints, that where proposed within in the scope of the role research.
To the natural type and the role type defined by Friedrich Steimann [1] two new types are added in CROM. The two new types are called compartment type and relationship type. These four types are not only defined by the properties *rigidity* and *foundedness*, but also by the property *identity*, as well. We already explained rigidity and foundedness in the previous section 2.1. Hence, only identity needs to be defined. Identity describes whether an instance of a certain type has an unique, derived or composed identity. An example for this can be found in the context of a reader reading a book insofar that the person playing the role reader has an unique identity the entire life time and in contrast to that reader derives its identity from its player, the person. Furthermore, the read relationship between the reader and the read get its identity by combining the reader identity with the read identity.
Based on this the four types are defined in CROM as follows: A natural type is rigid, not founded and its instance has a unique identity, whereas role types are anti-rigid, founded and the identity of instances of them is derived from their respective player. The compartment types are defined as rigid, founded and regarding instances they have a unique identity. Last but not least, a relationship type is rigid, founded and an instance of this type has a composed identity. Moreover, it is to note that the compartment type can fill role types, represents the context of roles and as such can contain multiple role types and the relationship type is used to incorporate the relational nature of roles in CROM and connects two role types with one another.
Besides the two new types various constraints are added to the role model in CROM. The constraints will not be explain here, because they have no relevance to this work. Nevertheless, it is important to mention them since in future works the transformation tool needs to be adapted so it can handle the constraints. This is necessary to make use of all information provided by the input.
Overall CROM is used by the transformation tool of this work to transform object-oriented software programs to the role-based paradigm, because it sufficiently incorporates the relational and context-dependent nature of roles together with several constraints and therefore CROM is the most expressive role language to this date.

2.3 RSQL

The Role Structured Query Language (RSQL) [5] is an extension to SQL, that incorporates the principle of roles into the query language, and was developed by Tobias Jäkel et al. RSQL was created as a way for database management systems (DBMS) to deal with challenges and problems of modern DBMSs like complexity, change, object schizophrenia and distributed state. As such it is used by the transformation tool of this work to transform an arbitrary DBMS to a role-based DBMS. Furthermore, to enable the presentation of roles in DBMS Tobias Jäkel et al. developed a formal model for the logical representation of roles called Dynamic Data Types (DDT).

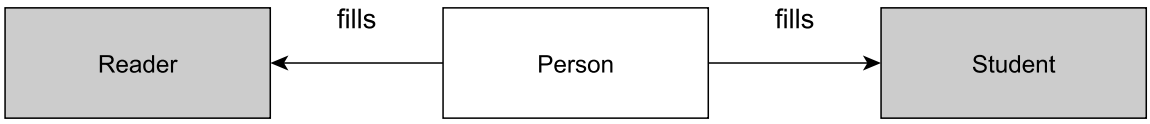

Figure 2.1: Dynamic Data Type Person

A DDT consists of one natural type and multiple role types that are filled by the natural type. An example of such a DDT is depicted in Figure 2.1, where person is a natural type and reader and student are role types filled by person. For both types the definition of Friedrich Steimann [1] is used. Moreover, the relationship type was added to the formal model of the DDTs as a third type. A relationship type is defined in the context of RSQL as a binary relation between two role types. The role types represent the two ends of the relationship.
An instance of a DDT is connected to the fills-relations and as already explained these relations connect natural types and role types. Furthermore, they indicate that a natural type *can play* a role type. The *can play* semantic leads to various settings for DDTs and these settings are called configurations in RSQL. All possible configurations for the example DDT person are displayed in Figure 2.2. The configurations are important to understand the instance representations of a DDT and an instance is called dynamic tuple. A dynamic tuple is always related to one configuration of the corresponding DDT. Hence, different dynamic tuples can be instances of the same DDT, but differ in their configuration. Besides this, an instance of natural type is called natural, the instance of a role type is called role, and the instance of a relationship type is called relationship. Last but not least, an instance of a fills-relation is called plays-relation and it is to note that a DBMS must always incorporate the concepts of DDTs to support RSQL.
As pointed out in the beginning, RSQL is an extension to SQL. Hence, parts of the RSQL syntax, where directly taken from SQL and RSQL is composed of a data query language (DQL), a data manipulation language (DML) and a data definition language (DDL) like SQL. Based on this fact, it is only necessary to discuss the differences between SQL and RSQL here.
In the area of the DQL the biggest difference between SQL and RSQL is the *config-expression* and the *relation-clause*. The config-expression handles the complexity of DDTs and is used in the *from-clause* of RSQL statements and queries instead of *table-expressions*. Fur-

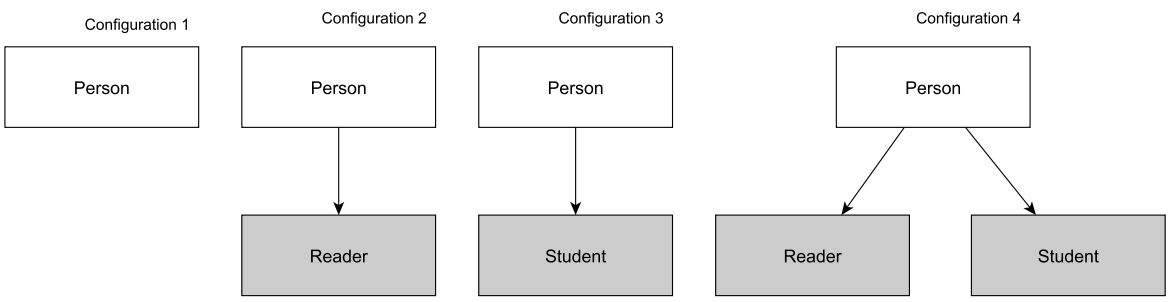

Figure 2.2: Configurations of the DDT Person

thermore, config-expressions consists of two parts. In the first part the natural type of the DDT is given and in the second part, called *playing-clause*, the configuration is defined by giving the role types that are played by the natural type in this configuration. To get a better understanding of this, a possible config-expression for the DDT Person is the following:

```
(NATURALTYPE Person p PLAYING( ROLETYPE Reader r) AND ( ROLETYPE Student s))
```

The *relation-clause* serves the purpose of handling relationship types in RSQL. In a select statement it can be placed as an optional clause with the prefix RELATING between the *from-clause* and the *where-clause*. The prefix can be followed by more than one relation-clause. In such a case the relation-clause are connected by ANDs. One relation-clause consists of two role type abbreviations and a relationship type reference. In RSQL the relation-clause is only used in the DQL, whereas the config-expression is used in the DML as well.
In the DML the config-expression is used in the insert statement for role types and relationship types, the update statement for role types and the delete statement for natural types and role types. The role type *insert* differs from the SQL *insert* in the point that the prefix INSERT is directly followed by the role type name, the attribute names, and the value expressions. Furthermore, an role type insert statement has a second part that starts with the prefix INTO and the INTO is followed by a config-expression and an optional where-clause. In the case of a relationship type insert statement, the first part of the statement consist of the prefix INSERT, the relationship type name, the attribute names, the prefix VALUES and the values expressions and the second part is composed of the prefix INSTANCE, two role type abbreviations, a from-clause with config-expressions and an optional where-clause.
The update statement for role types contains the config-expression between the table name and the set-clause. Beside this, the update statement doesn't differ much from the SQL update statement. In the delete statements for natural types and role types, config-expression are used in the the from-clause. Furthermore, another point that is different between RSQL and SQL is that the delete statement for natural type must start with DELETE NATURALTYPE and respectively for role types must the statement start with DELETE ROLETYPE followed by the role type name and a optional abbreviation. It is possible to use NT and RT in the place of NATURALTYPE and ROLETYPE.
The RSQL DDL differs from the SQL DDL in the following points: First, in RSQL the create statement must start with CREATE NATURALTYPE/NT for natural types, CREATE ROLETYPE/RT for role types and CREATE RELATIONSHIP/RST for relationship types. Second, the create statement for a role type has a *played by clause* at the end. This clause specifies the construction of the fills-relation. Last, the create statement for relationship types ends with the prefix CONSISTING OF followed by two *relation-participation-clauses*. A relation-participation-clause is composed of a role type name and the corresponding cardinalities of the role types in the context of the relationship.

2.4 Model Transformation

2.4.1 Overview

Model transformation or *model to model transformation* is an important tool to realize the model-driven engineering approaches in software development. The general functionality of model transformation is to translate the content of a source model to a target model.
Furthermore, model transformations are divided into two groups. Every transformation, that has a distinct source and target model belongs to the first group, *exogenous transformations*

[14]. Such transformations are typically used for model refinement and model migration. In contrast to transformations of the second group, called *endogenous transformations*, that are used to restructure or refactore models to improve the quality of the model or enforce a standard. A endogenous transformation has a source and a target model with the same meta model.
For the implementation of model transformation many different approaches exist. These approaches are distinguished by their nature. Some of them have an imperative nature, others are based on declarative specifications and even hybrid approaches, that combine aspects of declarative nature and imperative nature, exist. Moreover, in the recent years model transformation approaches became more and more popular, because of the need to deal with problems like legacy systems and insufficient documentary. Hence, to find the best approach people experimented with every part of model transformation and this lead to the different natures of the approaches.
Last but not least, the term model transformation can be used for *model to text transformations*, as well [11]. The general functionality of a model to text transformation is to generate text artefacts from the model.

2.4.2 Model-Driven Software Development

Model-driven software development (MDSD) is the development of software-systems based on models. The aim of MDSD is to deal with the increasing complexity of modern software-systems by abstraction, modelling and automatization. Hence, in general MDSD consist of the following steps. First, the requirements of the software are abstracted and put into a model. Then, depending on whether the software is refactored, migrated, or developed from scratch, the content of the old model or text artefacts will be automatically or semi-automatically transformed based on the model from the first step. Afterwards, the result of the second step, when it is a model, is used to generate code, queries and other text artefacts. This procedure is repeated every time new requirements are added to the software-system. The transformation tool of this work is an implementation of MDSD. Furthermore, the approaches GreGen.NET [7], Kermeta [8], QVT-R [9], ATL [12] and UML-RSDS [13] are introduced in the next subsections, because they were evaluated by S. Kolahdouz-Rahimi et al [14] and so they are a good base to choose a transformation approach for the tool.

2.4.3 GreGen.NET

GreGen.NET [7] is a model transformation approach based on graph theory. Edgar Jakumeit et al. developed this graph rewrite system to transform graphs into highly efficient .NET code. They realized the transformation through the combination of declarative graph transformation rules and imperative features.
A GreGen.NET transformation has transformation rules as basic units, that operate on one or more model elements. These rules use graph patterns to identify the elements to which they are applied to. When one of these elements is found the rules are able to create a new element based on the found element, delete it, or modify the features of the found element [14].

2.4.4 Kermeta

Kermeta [8] is an imperative model transformation approach, that is based on the the type system of MOF. In contrast to GreGen.NET, transformations in Kermeta are not using rules but are defined by sets of classes and operations that access and manipulate the model elements. The user of Kermeta can manipulate the model via the following options: creation of elements, deletion of elements and setting element features. Besides, the user can program logical data structures like sets to store a selected group of elements. To iterate over such data structures OCL-style operators like *select* and *each* are at the disposal of the user. Last but not least, program control structures such as conditionals, sequencing, and operation invocation can be used, as well. To simplify the usage of all these tools Kermeta provides the user with a meta-programming environment and an action language [14].

2.4.5 QVT-R

The third model transformation approach is QVT-R [9, 10]. QVT-R has a declarative nature and was defined in an OMG standard [10]. The transformation from the source model to the target is realized in QVT-R through relations that are established by creating or modifying elements in the target model. Furthermore, a relation always relates elements of a specific entity type with elements of another specific entity type and in most cases the first entity type is from source model, whereas the second entity type is from the target model. To relate the elements a relation is provided with the following means: rely on certain existing properties, use relations that are already established, invoke subordinate relations and actions. Last but not least, the execution of a QVT-R program is always started by a top relation. These deal with the matching elements of the input model in a non-deterministic order [14].

2.4.6 ATL

ATL [12] belongs to the category of hybrid model transformations, because it has declarative constructs, as well as imperative constructs. In contrast to the approaches introduced previously ATL transformations are constrained since they are unidirectional. This implies that the source model can be navigated but not changed or it can only be read during the transformation. Furthermore, the resulting target model can be changed but not navigated or only write operations can be applied to it by the transformation.
For the implementation of *model to model transformation* ATL provides modules. A module is composed of a header section, an import section, helper functions, and transformation rules. The header section provides the meta informations of the transformation. These are the name of the transformation, the declaration of the source model, and the target model. Moreover, an ATL transformation can have multiple input and output models and all of them need to be declared in the header. Additionally, it is to note that ATL supports two different execution modes for transformations: the refining mode for *endogenous transformations* and the default mode for *exogenous transformations*.
The helper functions and the rules are used to specify the transformation. Rules in ATL are divided into two groups: *matched rules* and *lazy rules*. The only difference between these two kinds of rules is that rules of the second group are only applied when they are called by another rule. Furthermore, it is possible to declare a *lazy rule* unique in ATL. Such a rule differs from normal *lazy rules* in the point that it always returns the same target element for

a given source element. This is done through navigating the internal traceability links of an ATL transformation module.
ATL rules are similar to QVT-R transformations since they relate elements of a specific entity type to elements of another specific type. In contrast to QVT-R the entity type of the first element group must always be from the source model and the entity type of the second element group must be from the target model. Besides, ATL provides the user with the possibility to add an optional guard to the source model element pattern. This guard is given as a Boolean expression in OCL. Last but not least, ATL rules create bindings. A binding refers to a feature of a target model entity type and specifies an expression whose value is used to initialise the refereed feature.
The structure of ATL rules is unique, because it can have declarative and imperative components. The declarative components are the using block, the source pattern and the target pattern. The elements are related and the bindings are specified within the source and target pattern. Furthermore, in the using block variables can be defined that can be used in the bindings or in the action block. The action block is the only possible imperative component of an ATL rule and it is nothing more than a sequence of imperative statements. It can be used as a replacement for the target pattern or in combination with the target pattern to assign attributes and create a control flow. Besides, the action block ATL provides another imperative construct, the *called rules*. Like the name suggest a *called rule* is a rule, that can be called by other rules as a procedure. Called rules differ in their structure from the other rules since they only can be composed of a using block, the target pattern with some bindings and the action block [14].
ATL can execute model to text transformation with ATL queries. These queries can only consist of helper functions and query section. The query section contains the query on the input model.

2.4.7 UML-RSDS

UML-RSDS [13] is, like ATL, a hybrid model transformation. In UML-RSDS, system data and meta models are defined by UML class diagrams together with OCL constraints. The behaviour in UML-RSDS is defined through operations of classes and use cases that operate on the class diagram. To specify a use case, preconditions and postconditions are used. Hence, on the meta data side, a transformation in UML-RSDS is composed of its source and target meta model or class diagrams and the functionality of the transformation is specified by operations and use cases. Furthermore, UML-RSDS provides a generator to generate a executable Java implementation from the class diagrams, operations and uses cases [13, 14].

2.5 Benchmarking

In general the term benchmarking describes a evaluation that compares a result or process with reference values, best practices, or a reference process. Benchmarking is a methodology used in many different science and economical areas. This leads to the fact that many different kinds of benchmarks with different goals exist [15].
The only kind of benchmarks that is relevant to this work are computer benchmarks. Computer benchmarks analyses the technology of IT infrastructures. They can be divided into three groups. The first group are the software benchmarks. These measure and compare the performance of programming languages and DBMSs. Hence, a software benchmark con-

$$\text{Design Complexity} = 1 - \frac{\text{Number of Design Entities}}{\text{Number of Design Relationships}} \tag{2.1}$$

Equation 2.1: Formula for Measuring Complexity of a UML Model by Harry Sneed et al.

sists of implementations of the same algorithm in different programming languages. The implementations are executed in succession on the same hardware during the conduction of the benchmark. Furthermore, to identify the best programming language the response time of the compiler, the run system, and the interpreter of the languages are measured. After these measurements, the measured values are then compared. In case of data base management system only the response time is evaluated.
The second group are hardware benchmarks and the third group are application benchmarks. These don't have any relevance for this works since the benchmark transformation process of the transformation tool only deals with database benchmarks. In general a database benchmark consist of DDL statements for the creation of the schema, a data generator and procedures. The procedures apply the workload of the benchmark. Furthermore, in the context of databases two types of workload exist.
The first type is called online transaction processing (OLTP). Workloads of this type contain queries that write data to the database and read data from it. In contrast workloads of the second type, called online analytical processing (OLAP), are composed of queries that retrieve, analyse and consolidate massive amounts of data. Hence, these queries have a large delay. A benchmark can have a workload of one of these type or a workload that is a combination of both.

2.6 Metrics

In science metrics are standards of measurement and used for evaluations, comparisons and analysis. Hence, to evaluate the transformation tool and the investigation results they are necessary for this work.
The group of metrics used in this work is called software metrics. A software metric is a function that projects a property of a software, e.g. size, quality or complexity, in a numeric value [16]. The numeric value serves as a base for comparisons, optimizations, and evaluations. Software metric are usually used to determine errors or estimate cost.
They range from simple to complex. Some of the simpler ones are the lines of code (LOC) and the program run time. Lines of code is a metric where the lines in the text of a programs source code are counted to determine the size of the respective computer program. Typically, the result of the LOC metric is used to predict the amount of effort required for the development of the program but for the estimation of software maintainability it can be used, as well. In contrast the program run time is just used to get the performance of a computer program by measuring the time the program needs to solve a problem or perform a task.
Computer programs are not the only part of software for which metrics exists. For example, the test coverage of a program, the extent of the programs test suite, can be measured by the metric code coverage. Furthermore, the complexity of an UML model can be calculated with the UML complexity formula of Harry Sneed et al [17, 18]. Code coverage can be implemented with many different metrics.
The UML complexity formula is presented in the Equation 2.1. Complexity in regard to

this formula is defined as the relation of design entities to design relationships, whereas the number of entities represents the size of the UML model. The number of relationships determines whether the complexity is high or low. As seen in the formula the more connections or dependencies exist between the model entities the higher is the model complexity. Note, that the terms design entities and design relationships in this context only stand for UML elements and relations. Furthermore, the *1* - in the formula is a correction for the case when the number of entities is greater than the number of relations.
The formula of Harry Sneed et al is used in the next chapter to analyse the investigation results regarding the complexity of the benchmark schemata at different transformation stages. It is possible to measure the complexity of a RSQL model and a CROM model with this formula since its definition is generic and can be easily extended for other model languages besides UML.

3 Investigation and Transformation of Benchmarks

The first subtask of this work and the topic of this chapter is the investigation of the benchmark schemata of SmallBank [19], SEATS [20], TPC-C [21], TPC-H [22] and AuctionMark [25]. The goals of the investigation are to learn how a model changes when its transformed to CROM or RSQL, to gather the informations that are necessary for the transformation and to determine the degree of automatization that can be reached. To reach the first goal the complexities of the schemata are calculated at every transformation stage and compared with each other. Furthermore, the results of the investigation are used as a base for the specification and implementation of the transformation tool in the next chapters.
The investigation procedure of a benchmark schema consists of the following steps. In the first step the schema is manually transformed to a schema with a NTRT meta model. The NTRT meta model has only natural types, role types, attributes and primary key constraints as entities. As relations it has only fills-relations and foreign key constraints. The purpose of this meta model is to serve as an interim stage in the transformation process between the input model and the CROM model or RSQL model. Such an interim stage is used here, because compartment types and relationship types can only be defined for already existing natural types and role types. While the transformation can be done without the NTRT meta model stage, its results in lesser process complexity per step.
After the schema is transformed to a NTRT schema, the next step is to manually transform the NTRT model to a RSQL model and a CROM model. The results of the first steps are used as a foundation for the next steps.
In the third step of the investigation the informations, essential for a transformation of an object-oriented model to a role-based model, are extracted from the results. Last but not least, the complexity of the schema in every transformation stage is calculated to analyse whether a role-based model has benefits in this regard.
Accordingly, the chapter is composed of the following sections. The first section presents how the model complexity of the schemata at every stage is calculated. The second section deals with the investigation results for the SmallBank schema. The next section contains the investigation of SEATS. Then the insights on the benchmark schemata of TPC-C, TPC-H and AuctionMark are stated in the fourth, fifth and sixth chapter. Last but not least, the last section summarizes the required information for transforming an object-oriented model to a role-based model and evaluates the calculated model complexities, as well.

3.1 Model Complexity Formula

For the calculation of the model complexity at the SQL, NTRT, CROM and RSQL stage the formula by Harry Sneed et al. [17, 18] (see Section 2.6) will be the base. This metric is only designed for UML models, but the formula is defined in such a generic way that it is easy to extend the definition for other models.

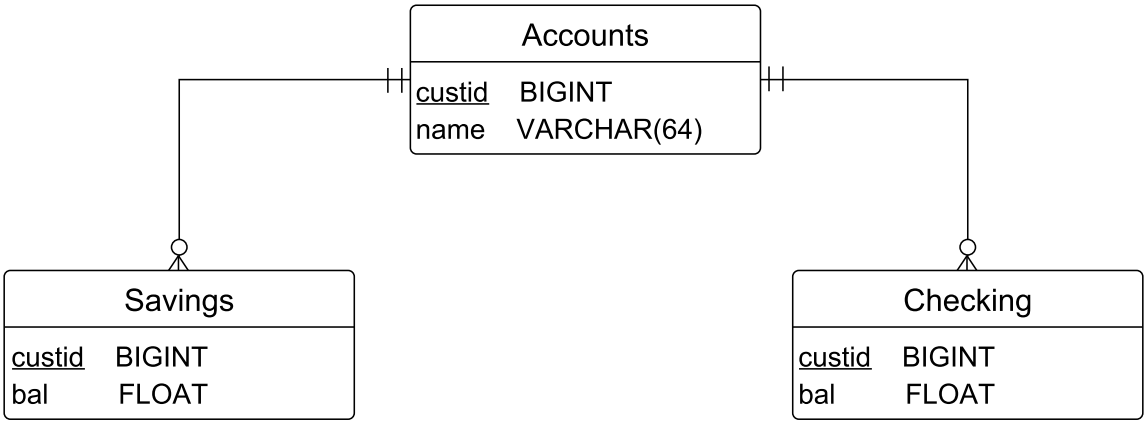

Figure 3.1: SQL Schema of SmallBank

To make the Sneed's formula usable for other kinds of model the specifications of terms design entities and design relationships just need to be expanded. Hence, to get the complexity of a SQL schema, tables will be counted as design entities and foreign key constraints will be counted as design relationships since they relate tables with one another. The columns and all constraint types except foreign key constraints are not counted, because they are parts of a table.
In case of a NTRT model, the natural types and role types can be seen as design entities and the fills-relations and foreign key constraints can be seen as design relationships. This specification applies for the CROM and RSQL models, as well. Except that the concept of foreign key constraints is not included by CROM and therefore they are not counted. Nevertheless, to use the formula for CROM and RSQL the specifications need to be expanded further. Hence, to measure the complexity of a RSQL model the relationship types should be counted as design relationships. Note, that in RSQL relationship types could be seen as design entities, because they have associated tables with columns. Since the relationship are defined as relations in the most cases it was decided to stick with the first specification. However, to emphasize the fact that relationships are more complex in RSQL than in CROM a relationship type is counted as two design relationships. They are not counted as one design entity and one design relationship, because the existence of the associated tables depends on the implementation of the RSQL DBMS. For example, it is possible to store the columns of the tables in the source and target role type of a relationship type and to connect the column through constraints.
Last but not least, the additional specifications for a CROM model are the following. Compartment types will be counted as design entities and relationship types and containment-relations as design relationships. A containment-relation describes the containment of a role type by a compartment type. The remaining elements of CROM will be ignored due the facts that they have no relevance to this work or they were neither considered nor found during the manual transformation.

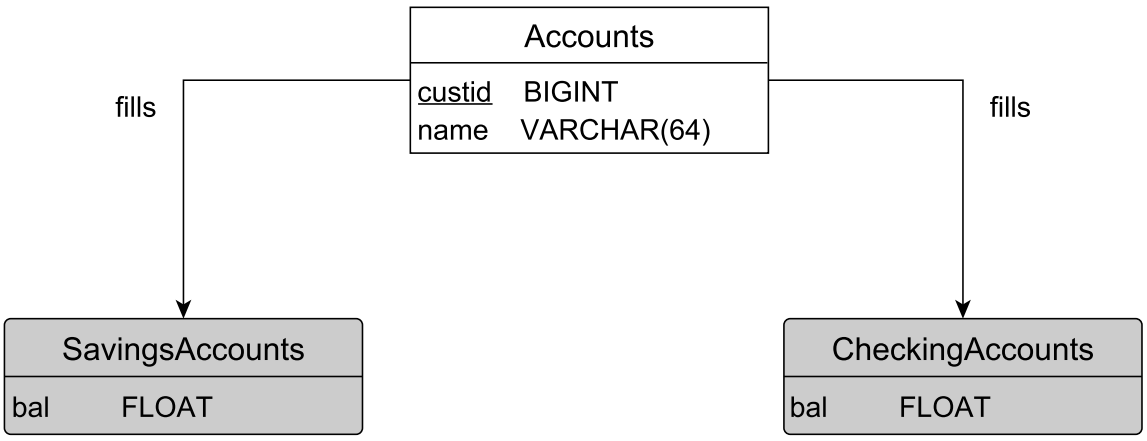

Figure 3.2: NTRT Schema of SmallBank

3.2 SmallBank

SmallBank [19] is the first benchmark, whose schema is investigated in this work, and was developed by Mohammad Alomari and Michael Cahill. It represents a banking application where transactions perform simple read and update operations on their checking and savings accounts [27].

At the SQL stage SmallBank consists of three tables: `Accounts`, `Savings` and `Checking`, as depicted in Figure 3.1. The table Accounts has two columns `custid` and `name`, whereas the former is a primary key. Similarly, `Savings` has two columns like `Accounts`, called `custid` and `bal`. The `custid` column is a primary key, as well as a foreign key that is related to the `custid` column of the table `Accounts`. Last but not least, the table with the name `Checking` also has the columns `custid` and `bal`. The first of the two is a primary key and a foreign key that relates to `custid` column of the table `Accounts`. Hence, at the SQL stage the SmallBank schema has 3 design entities and 2 design relationships. When these parameter values are used to calculate the model complexity the result value is 0.5 for the SmallBank SQL schema.

As described at the beginning the first step is the transformation to a NTRT schema. The result of the transformation is presented in Figure 3.2. As this figure indicates the table `Accounts` was transformed to a natural type, because an account is an entity able of existing on its own. Furthermore, an account losses its identity when closed or deleted. The columns and constraints of the table `Accounts` did not change during the transformation. The tables `Checking` and `Savings` were changed to role types since their foreign key constraints show that these two entities can not exist without the table `Accounts`. Accordingly, `Checking` was renamed to `CheckingAccounts` and `Savings` was renamed to `SavingsAccounts`. For the columns of them is to note that the `custid` columns were removed from both tables due the transformation of the foreign key constraints to fills-relations. The `custid` informations are now provided through these fills-relations. The foreign key constraints were changed to directly represent direct dependencies. Overall, the SmallBank NTRT schema has 3 design entities and 2 design relationships and consequently a complexity of 0.5.

The second stage of the investigation is the transformation of the NTRT schema to a CROM schema and a RSQL schema. A transformation to RSQL does not lead to any changes in the model. In this case changes would only occur when the schema is extended or

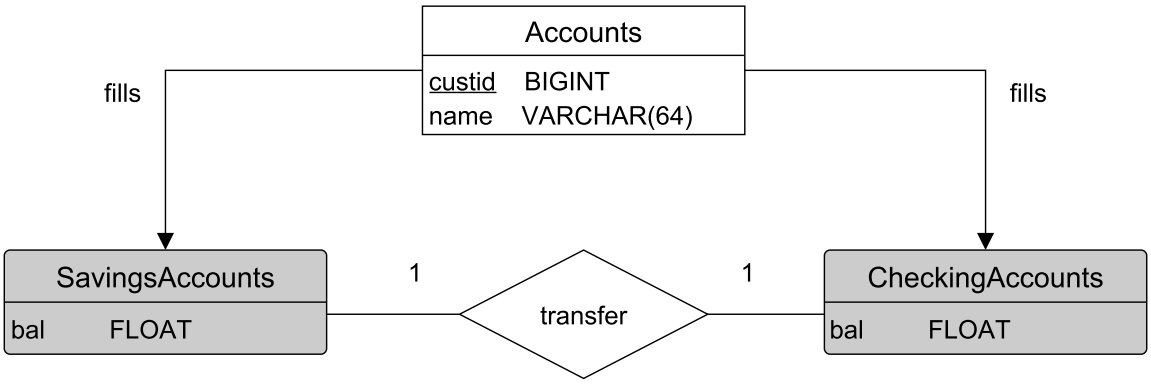

Figure 3.3: RSQL Schema of SmallBank

evolved. A possible extension were the adding of relationship types between the role types `CheckingAccount` and `SavingsAccount`. For example, a transfer relationship between the role types as seen in Figure 3.3 would be plausible, because in the most cases people own accounts of both types and need to transfer money between them. Note, that in practice the transfer would be defined in a more general way to represent all transactions. In case of a transformation to CROM extensions like these could be added, as well, but besides the compartment type `Bank` was added to the schema to fulfil the containment condition of CROM. The containment condition is a term used for the fact that in CROM a role type must be contained by a compartment type. One problem exists with the creation of the compartment. The input schema does not contain any specific informations or objects that could be transformed into a compartment type. Hence, the one who does the transformation needs to use his or her domain knowledge for the creation of `Bank`. The complete CROM schema is shown in Figure 3.4.
For the following complexity calculations the transfer relationship is counted since it has a high possibility to be created. Therefore, the RSQL schema has 3 design entities and 4 design relationships, also it has consequently a complexity of 0.75 . The CROM Schema has 4 design entities and 5 design relationships and thus a complexity of 0.8.
Last but not least, based on the SmallBank schema at the different transformation stages, it can be concluded that the information needed to transform an object-oriented model to a role-based model are the following:

- Hints about which objects should be changed to natural types.
- Hints about which objects should be transformed to role types.
- Hints to know, which objects should be renamed.
- Hints to know, which columns or attributes should be removed.
- Specifications for new relationship types.
- Specifications for new compartment types.
- Specifications for setting fills-relations and remove foreign key constraints or transforming foreign key constraints to fills-relations.

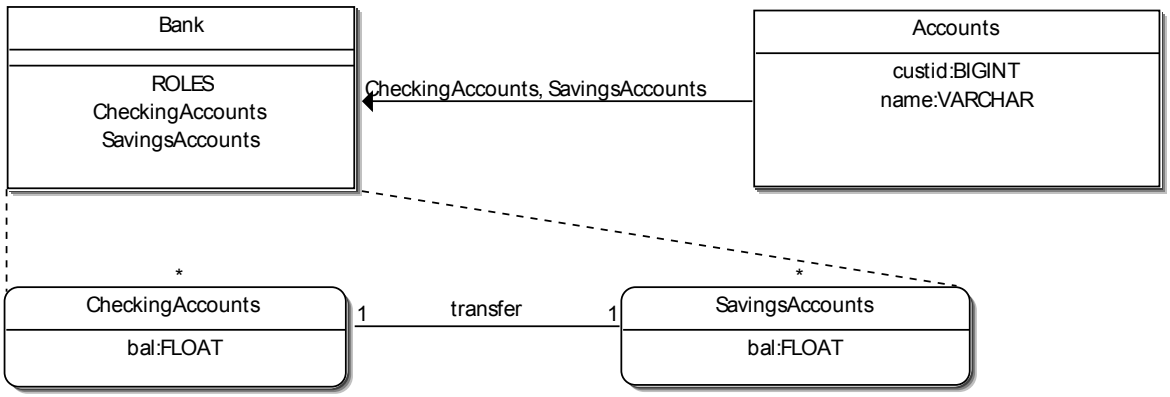

Figure 3.4: CROM Schema of SmallBank

3.3 SEATS

The schema of Stonebraker Electronic Airline Ticketing System (SEATS) [20] was investigated after the SmallBank schema. The application scenario of SEATS is a online ticketing service, where customers make reservations, search for flights, and update the frequent flyer information [27].

The SQL schema of SEATS is depicted in Figure 1. At the SQL stage the SEATS model is composed of the tables `Airport`, `Airport_Distance`, `Country`, `Airline`, `Customer`, `Flight`, `Frequent_Flyer` and `Reservation`. Hence, it has 8 design entities. On the relation side it has 12 foreign key constraints and therefore 12 design relationships. Consequently its complexity is 0.66. The foreign key constraints are related to the following columns: `R_C_ID` and `R_F_ID` of the table `Reservation`; `F_AL_ID`, `F_DEPART_AP_ID` and `F_ARRIVE_AP_ID` of the table `Flight`; `FF_C_ID` and `FF_AL_ID` of the table `Frequent_Flyer`; `C_BASE_AP_ID` of the table `Customer`; `AL_NAME` of the table `Country`; `D_AP_ID0` and `D_AP_ID1` of the table `Airport_Distance`; and `AP_CO_ID` of the table `Airport`. Before the investigation of the SEAT schema is explained, note that for this and the remaining schema investigations the reasons for transforming are not explained for every model element. Since in most cases similar transformations were explained in an other schema investigation or the transformations can be concluded from the definitions of natural types, role types, compartment types or relationship types.

The first step, the transformation of the SEATS schema to a NTRT schema. As the Figure 2 in the Appendix shows the tables `Country`, `Airport`, `Airport_Distance` and `Reservation` were transformed to natural types. The tables `Customer`, `Frequent_Flyer`, `Airline` and `Flight` were changed to role types and the table `Flight` was renamed to `PlaneInComission`. Accordingly, the columns of `Flight` were renamed, as well. The purpose of the renaming is to highlight the context of the fills-relation between the natural type `Plane` and the role type `Flight`. `Plane` was extracted from `Flight`, because of its meaning. `Flight` is a role type and as such needs to be played by a natural type. The problem in this case is that the input model does not have any other objects that could be transformed into a natural type that is able to fill `Flight` in the context of SEATS. To solve this problem the necessary information were extracted in form of columns from the table `Flight` and than they were transformed into the natural type `Plane`. The natural types `Company` and `Person` were created in the same way. `Company` was extracted from `Airline` and in the case of `Person` the information for the trans-

formation were extracted from the role types `Customer` and `Frequent_Flyer`. This shows that the extraction needs to work for multiple role types in one way or another. Furthermore, the fills-relations between the extracted natural types and their respective role types are specified or set during the extraction process. The last elements to be described are the role types `Arrival_Airport` and `Departure_Airport`. These elements were not created through extraction rather they were created based on the domain knowledge that `Airport_Distance` will be transformed into a relationship type. In terms of the domain this relationship will have a departure airport as source and as an arrival airport as target. Role types are needed to represent this. Hence, the creation of `Arrival_Airport` and `Departure_Airport` depends on domain knowledge. Overall, this leads to 13 design entities and 18 design relationships for the NTRT schema of SEATS, because the foreign key constraints were not transformed or deleted. The resulting complexity of these values is 0.72.
The second stage of the investigation is the transformation to a RSQL model and a CROM model. The important point of this stage is the change of natural types and role types to relationship types. As presented in the Appendix Figure 3 `Airport_Distance` was changed to the relationship `distance_between`, `Reservation` to the relationship `reserves` and `Frequent_Flyer` to the relationship `is_frequent_flyer`. All these transformations were done based on the meaning of the respective natural types and role types and the fills-relations associated with them were deleted. Furthermore, consider that from the input to NTRT the `Airport_Distance` and `Reservation` were only transformed to natural types, because of their meaning. They could not be role types and the NTRT meta model does not know any relationship types. Hence, both of them became natural types as a intermediate state. Moreover, it is possible to extend the schema for example by adding relationship types like the `transfer` that was added to the SmallBank schema. To consider this the relationship types `flying_to`, `depart_from` and `owns` were created and added to the RSQL schema. Hence, the SEATS RSQL model has 10 design entities and 30 design relationships including the relationships `flying_to`, `depart_from` and `owns`. These values lead to a complexity of 0.77 for the SEATS RSQL schema.
The CROM schema of SEATS differs from the RSQL, because the compartment type `Flight` was created and added to the model to fulfil the containment condition of the role types. Like the compartment `Bank` of the SmallBank schema the creation of `Flight` is based on the domain knowledge available during the transformation. The Figure 3.5 and the Appendix Figure 4 are the visual representations of this. All in all, the CROM schema has 11 design entities and 17 design relationships with the containment-relations. The complexity that follows from these values is 0.64.
Last but not least, the following information necessary for the transformation of an object-oriented model to a role-based model could be gathered from the investigation and transformation of the SEATS schema:

- Specifications for creating new role types and setting fills-relations between them and natural types.
- Hints to extract natural types from model elements.
- Specifications to define which children of the model elements will be moved to the new natural types.
- Hints about the natural types or role types that will be changed to relationship types.
- Specifications of the column names to be changed.

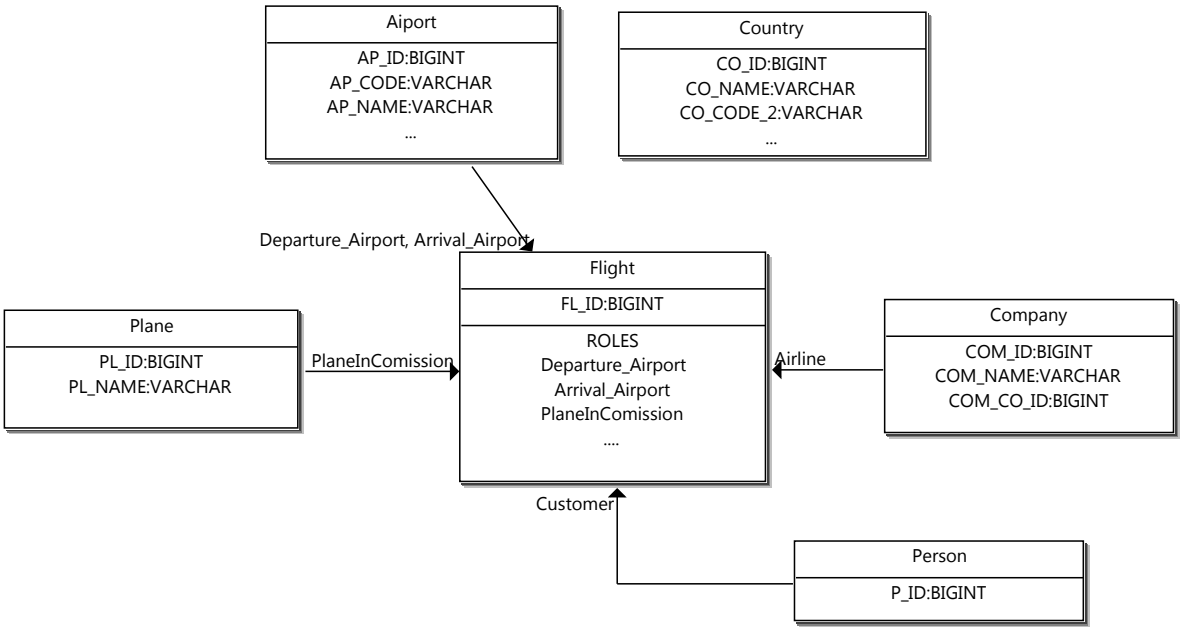

Figure 3.5: CROM Basic Schema of SEATS

3.4 TPC-C

TPC-C [21] is the third benchmark, whose schema was investigated in this work. Furthermore, it is the current industry standard for evaluating OLTP systems. To do the evaluation, the following application scenario is used. TPC-C simulates a company which is is a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. A district serves customers and these can order items that are stored in the districts warehouse or another warehouse. As the order is processed, the customer can look up the status of the order at anytime. When the company than completes an order it delivers the items and the respective customer makes a payment. Additional to these procedures the company's system and workers can look up the stock level of an item at anytime [23, 27].

Figure 5 in the Appendix shows the SQL schema of TPC-C. As depicted their the SQL schema consist of the tables `Warehouse`, `Stock`, `Item`, `District`, `Customer`, `Order`, `History`, `New_Order` and `Order_Line`. In terms of relations it has foreign key constraints on the following columns: `D_W_ID` of the table `District`; `C_W_ID` and `C_D_ID` of table `Customer`; `H_C_W_ID`, `H_C_D_ID`, `H_C_ID`, `H_W_ID` and `H_D_ID` of the table `History`; `NO_W_ID`, `NO_D_ID` and `NO_O_ID` of the table `New_Order`; `O_W_ID`, `O_D_ID` and `O_C_ID` of the table `Order`; `OL_W_ID`, `OL_D_ID`, `OL_SUPPLY_W_ID`, `OL_I_ID` and `OL_O_ID` of the table `Order_Line`; and `S_W_ID` and `S_I_ID` of the table `Stock`. Overall, the TPC-C schema has 9 design entities, 21 design relationships, and consequently a complexity of 0.57.
During the first step of the investigation the SQL schema is transformed to a NTRT schema. The transformed schema is shown in the Appendix Figure 6. As the figure depicts the tables `Stock`, `History`, `Item` and `Order_Line` were changed to natural types. `Order_Line` is a special case, since it is a natural type only temporary before the transformation to a relationship type. The tables `Warehouse`, `District`, `Order`, `New_Order` and `Customer` were converted to role types. `Order` was renamed to `OrderInProcess` due the fact that from `Order` and

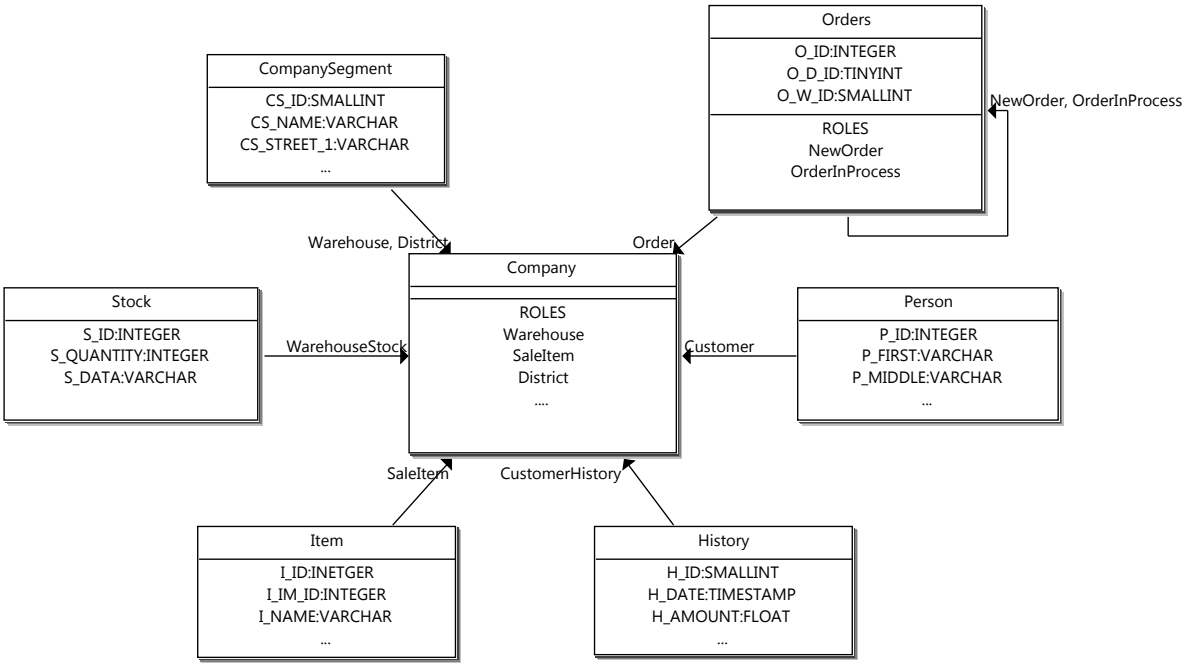

Figure 3.6: CROM Basic Schema of TPC-C

`New_Order` the natural type `Order` was extracted. The natural types `CompanySegment` and `Person` are extractions, as well. Hence, `CompanySegment` fills `Warehouse` and `District` and was obtained from them. Respectively `Person` fills `Customer` and was extracted from it. The last elements to discuss are the role types `SaleItem`, `WarehouseStock` and `CustomerHistory`. `SaleItem` was newly created and added to the model for the possible definition of relationship types between itself and `WarehouseStock`. Such relationship makes sense, because on a semantic level they are connected. Last but not least, `WarehouseStock` and `CustomerHistory` are role types extracted from `Stock` and `History`. In some cases the extraction of role types is necessary to be able to define relationships at the next stage of the transformation. For example, `WarehousStock` is the target of `Order_Line`, when it is changed to a relationship type. Overall, the TPC-C NTRT schema has 15 design entities, 29 design relationships and a model complexity of 0.51.

The next stage deals with the transformation of the NTRT schema to a RSQL schema and a CROM schema. The RSQL schema differs from the NTRT schema in the following issues. `Order_Line` was transformed to a relationship type and the relationship types `has`, `covers`, `contains`, `serves`, `has_a`, `places` and `request_status` were created and added to the model during the transformation. In the end, TPC-C at the RSQL stage has 14 design entities and 45 design relationships. These values lead to a model complexity of 0.68 for the TPC-C RSQL schema.

The CROM schema has the same relationship types as the RSQL schema. Besides this, the compartment type `Company` was created and added, the natural type `Order` was changed to a compartment type and the proxy role type `Order` was created and added to `Company`. Furthermore, sometimes tables and natural types describe a context or grouping and not a concrete type. For example, the term Order is the genus for all order kinds. Consequently, the natural type `Order` is better represented as a compartment type and should be transformed accord-

ingly. During such a transformation the fills-relations are preserved. However, a new problem arises from multiple compartment types in a model, because not all relationship types can be connected with their sources and targets. In the context of TPC-C, for example, the relationship `Order_Line` is no longer able to reach its source the role type `OrderInProcess`, since a relationship can not leave the containment of a compartment type. To solve this problem CROM provides proxy role types. Such a role type has the same name as the compartment type that contains the missing sources or targets. Moreover, it is filled by this compartment type and added to the compartment type that contains the associated relationship types. Therefore, it can serve as stand-in source or target. The role type `Order` is exactly such a role type for the relationship types `places`, `request_status` and `Order_Line`. Figure 3.6 and the Figures 8 and 9 in the Appendix are the visual representations of TPC-C at the CROM stage. Overall, the CROM schema of TPC-C is composed of 15 design entities and 26 design relationships. Furthermore, the resulting complexity from these values is 0.57.
During the investigation of the TPC-C schema at the different transformation stage it was learned that the following information are needed for the shifting the paradigm of a model from object-oriented to role-based :

- Hints for choosing the correct sources of role type extractions.
- Specifications for changing natural types to compartment types.
- Specifications for the creation of proxy role types.

3.5 TPC-H

The schema of TPC-H [22] is the subject of the fourth investigation section. In contrast to TPC-C [21] TPC-H is designed to evaluate OLAP systems. To evaluate such systems it considers a wholesale supplier that wants to analyse his or her business with a complex business analysis application. This application allows the supplier to analyse the following things: pricing, promotions, supply management, demand management, profit, revenue management, customer satisfaction study, market share study and shipping management [24].
The SQL schema of TPC-H is displayed in the Appendix Figure 10. At the SQL stage the TPC-H model is composed of the tables `Region`, `Nation`, `Supplier`, `Customer`, `Part`, `PartSupp`, `LineItem` and `Orders`. Furthermore, the foreign key constraints are defined on the following columns: `S_Nationkey` of the table `Supplier`; `PS_Suppkey` and `PS_Partkey` of the table `PartSupp`; `C_Nationkey` of the table `Customer`; `O_Custkey` of the table `Order`; `L_Orderkey`, `L_Partkey` and `L_Suppkey` of the table `LineItem`; and `N_Regionkey` of the table `Nation`. Hence, the TPC-H SQL schema has 8 design entities, 9 design relationships and consequently a complexity of 0.88 .
In the first part of the investigation, the SQL TPC-H was converted to a NTRT TPC-H. The transformed schema is presented in Figure 11 in the Appendix. As this figure shows the tables `Region`, `Order` and `PartSupp` were changed to natural types and `Part`, `LineItem`, `Customer`, `Supplier` and `Nation` were transformed to role types. The new natural type `Item` was extracted from `Part` and `LineItem`. `Item` is not the only extraction in the schema. The natural types `Company` that was obtained from `Supplier` and `Customer`, and `Person` that was extracted from Customer, are extractions, as well. With all of this, NTRT TPC-H has 11 design entities and 14 design relationships. When these value are put into the complexity formula the resulting complexity is 0.78. Furthermore, for the design relationships is to consider that

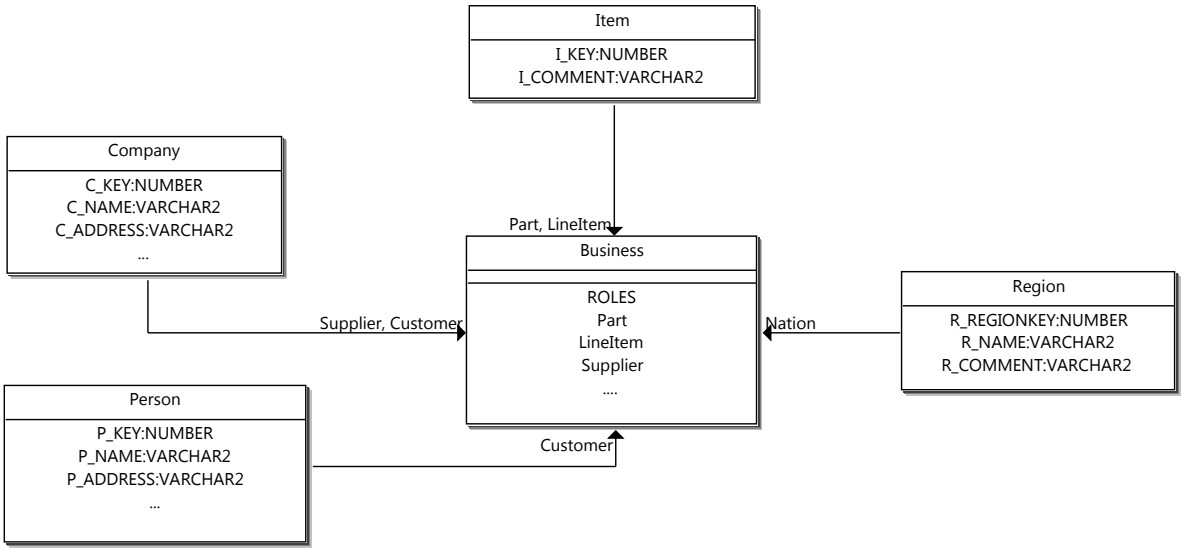

Figure 3.7: CROM Basic Schema of TPC-H

the foreign key constraint on the column `N_Regionkey` was transformed to a fills-relation between `Nation` and `Region`.
At the second stage of the investigation, the NTRT schema was transformed to a RSQL schema and a CROM schema. The RSQL schema differs from the NTRT schema, because the natural types `PartSupp` and `Order` were changed to the relationship types `supplys` and `orders`. Additionally, it is possible to create new relationship types for the schema based on the available domain knowledge. As the Appendix Figure 12 depicts this was considered by the creation of the relationship types `can_be_a` and `is_from`. In the end, TPC-H at the RSQL stage has 9 design entities and 24 design design relationships. These lead to a model complexity of 0.625 for RSQL TPC-H.
The CROM schema, displayed in Figure 3.7 and the Appendix Figure 13, has the same relationship types as the RSQL schema. Moreover, the compartment type `Business` was created and added to the model. As a result the TPC-H CROM schema is composed of 10 design entities and 16 design relationships. Hence, the complexity of CROM TPC-H is 0.625.

Last but not least, for the third step of the the investigation the information needed to transform an object-oriented model to a role-based model were sought from the TPC-H schema at the different transformation stages. However, no information were found that not already were gathered in prior investigations, but the necessity of the found informations were proven again.

3.6 AuctionMark

The AuctionMark schema [25, 26] is the last subject of investigation in this work. AuctionMark has an OLTP workload like TPC-C. This workload simulates an auction website. A user of this website can be a seller or a buyer. As a seller the user is able to add items for an auction, update the informations of existing items, and respond to the comments of the buyers. Is the user a buyer he can add items to his watch list, retrieve the details of an item, place a

bid on an open auction, purchase an item and leave comments or feedback for sellers. The system itself does the following transactions: periodically it checks whether auctions come to an end and it looks up the max bid for an item.
The Figure 14 in the Appendix presents the SQL schema of AuctionMark. As shown their the SQL schema consist of the tables `Region`, `User`, `Item`, `User_Attributes`, `Category`, `Global_Attribute_Group`, `Global_Attribute_Value`, `Item_Attribute`, `User_Watch`, `Item_Image`, `User_Feedback`, `Item_Max_Bid`, `Item_Bid`, `Item_Purchase`, `Item_Comment` and `User_Item`. Moreover, the schema has foreign key constraints on the following columns: `gag_c_id` of the table `Global_Attribute_Group`; `gav_gag_id` of the table `Global_Attribute_Value`; `c_parent_id` of the table `Category`; `ua_u_id` of the table `User_Attributes`; `i_u_id` and `i_c_id` of the table `Item`; `ii_i_id` and `ii_u_id` of the table `Item_Image`; `ic_i_id`, `ic_u_id` and `ic_buyer_id` of the table `Item_Comment`; `uf_i_id`, `uf_i_u_id`,`uf_u_id` and `uf_from_id` of the table `User_Feedback`; `ib_i_id`, `ib_u_id` and `ib_buyer_id` of the table `Item_Bid`; `imb_i_id`, `imb_u_id`, `imb_ib_id`, `imb_ib_i_id` and `imb_ib_u_id` of the table `Item_Max_Bid`; `ip_ib_id`, `ip_i_id` and `ip_u_id` of the table `Item_Purchase`; and `ui_u_id`, `ui_i_id` and `ui_seller_id` of the table `User_Item`. All in all, the AuctionMark schema has at the SQL stage 16 design entities, 30 design relationships and consequently a complexity of 0.53.
At the first stage of investigation, the AuctionMark SQL schema was transformed to a NTRT schema. The transformed schema is depicted in the Appendix Figure 15. Their is shown that the tables `Region`, `Item`, `User_Attributes`, `Category`, `Global_Attribute_Group`, `Global_Attribute_Value`, `User_Watch`, `Item_Image`, `Item_Max_Bid`, `Item_Comment` and `User_Item` were transformed to role types. In contrast `User`, `User_Feedback`, `Item_Attribute`, `Item_Purchase` and `Item_Bid` were changed to natural types. Furthermore, the following elements are extractions: the natural type `Resource` was extracted from `Item_Comment`, `Item_Image`, and `Item_Max_Bid`, the natural type `Object` from `Item`, and the natural type `Feature` from `User_ Item` and `User_Watch`. Other extractions are the natural type `Attribute` that was obtained from `User_Attributes`, the natural type `Global_Information` from `Region`, `Category`, `Glo- bal_Attribute_Group` and `Global_Attribute_Value`. Last but not least, the role types `buyer` and `seller` are extractions from `User`. In the end, NTRT AuctionMark has 23 design entities and 43 design relationships. These values lead to a complexity of 0.53.
During the second step of the investigation the NTRT schema were changed to a RSQL schema and a CROM schema. To get the RSQL schema the natural types `User_Feedback`, `Item_Attribute`, `Item_Purchase` and `Item_Bid` were transformed to the relationship types `gives_feedback`, `Item_Attribute`, `purchase` and `bids`. Moreover, to consider the possibility of extending the schema during the transformation the relationship types `has`, `creates`, `adds_for_auction`, `has_image`, `has_max_bid`, `is_value_of`, and `is_in_category` were created and added to the model. This is displayed in Figure 16. All in all, AuctionMark at the RSQL stage is composed of 19 design entities and 48 design relationships. Hence, it has a model complexity of 0.6.
The Figures 17, 18, 19 and 20 are the visual representation of CROM AuctionMark. The schema at this stage has the same relationship types as the RSQL schema. Besides this, the natural type `Global_Information` was converted to a compartment type and the compartment types `System` and `Auction` were created and added to the model. To deal with problem of multiple compartment types the proxy role types `Auction` and `Global_Information` were created and added to `System`. As a result the AuctionMark CROM schema has 19 design entities, 35 design relationships and consequently a complexity of 0.543 .
During the investigation of the AuctionMark schema no new information needed for the paradigm shift were gathered, but the necessity of the found informations were proven again.

3.7 Overview of the Needed Informations and Schema Complexities

The investigations of the benchmark schemata resulted in many informations that are needed to transform an object-oriented model to a role-based model. The problem with these is that most of them are derived from the context, domain or the meaning of objects. The knowledge of these three areas can easily be obtained by a human, but for a computer it is much harder to gather this knowledge. Hence, when a computer should do the transformation the needed informations must be provided or learned. To have all the needed information in one place they are listed here:

1. Hints about which objects should be changed to natural types.
2. Hints about which objects should be transformed to role types.
3. Hints to know, which objects should be renamed.
4. Hints to know, which columns or attributes should be removed.
5. Specifications for new relationship types.
6. Specifications for new compartment types.
7. Specifications for setting fills-relations and remove foreign key constraints or transforming foreign key constraints to fills-relations.
8. Specifications for creating new role types and setting fills-relations between them and natural types.
9. Hints to extract natural types from model elements.
10. Specifications to define which children of the model elements will be moved to the new natural types.
11. Hints about the natural types or role types that will be changed to relationship types.
12. Specifications of the column names to be changed.
13. Hints for choosing the correct sources of role type extractions.
14. Specifications for changing natural types to compartment types.
15. Specifications for the creation of proxy role types.

The last part of this section is devoted to the fourth steps of all investigations. The computation of the model complexities were done in the previous sections. The results are summarized in Figure 3.8. As this bar chart depicts the schema kind with highest complexity on average is RSQL. The reason for that are the complex relationship types that in contrast to CROM are composed of relations and tables. On average the CROM schemata have the second highest complexity due to the containment relations and the relationship types. Note, that with a very high probability CROM would be on place one when its constraints would be considered in this work. The lowest complexity on average has the NTRT schema kind, because of the following reasons. In many cases foreign key constraints were transformed to fills-relations so the increase of the design relationship counter was relatively low. The increase of design entities, on the contrary, was relatively high due to the extraction and creation of the natural types and role types.
Overall, it can be concluded that in most cases, the transformation of a model to RSQL or

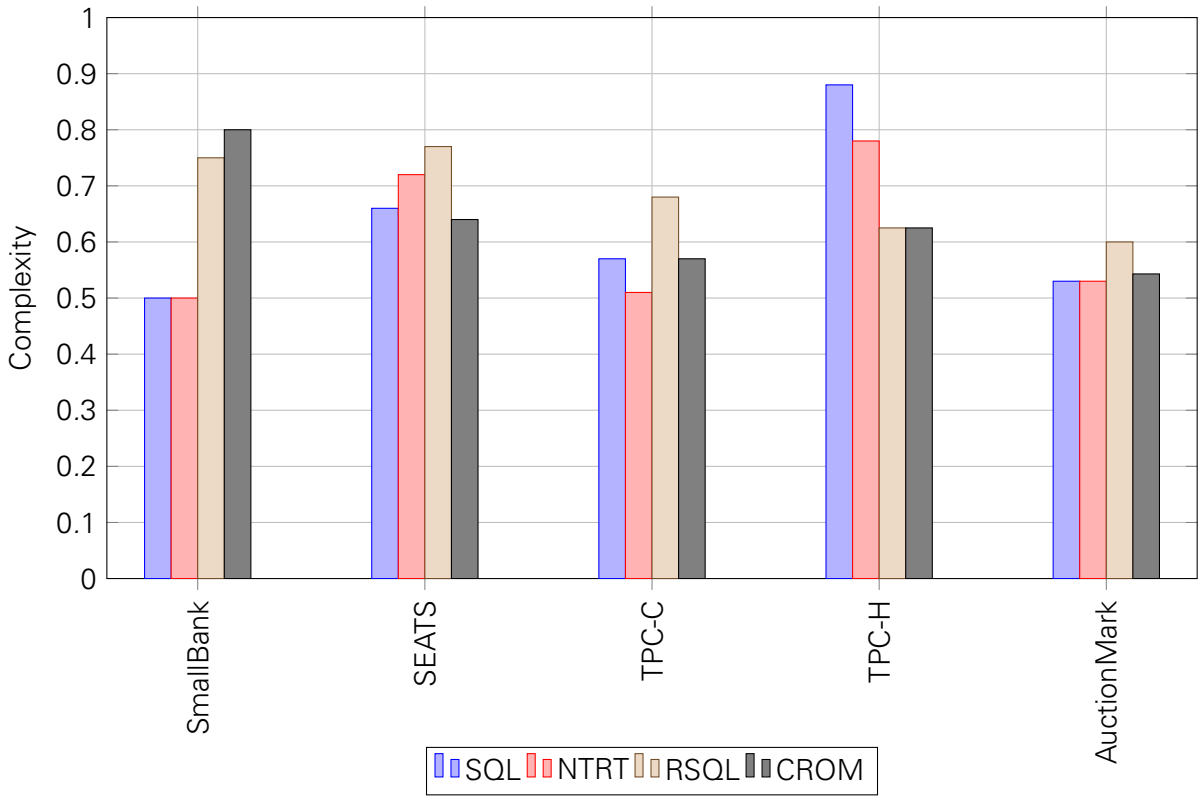

Figure 3.8: Model Complexity Diagram

CROM leads to an increase in the complexity of the model, but as the complexities of TPC-H suggest not in all cases. It can be assumed that the low complexity of TPC-H at the CROM and RSQL stage is the consequence of the transformation of two design entities to design relationships. Furthermore, only a few new entities were extracted or created. These two points influence the complexity so that when a model has much more relationships than entities multiple relationships have often the same target and the same source. Hence, except of the first relationship between a source and a target these relationships do not add much complexity to the model, because they can be encapsulate in one big relationship. In other words in the implemented system all of them would share the same channel so they do not cause the number of domain entities to increase. Nevertheless, the complexities increases, but in return the redundance of the model is reduced by the fills-relations, documentation is created or updated via the model and the expressiveness of the model is improved since roles and naturals allow a more concrete representation of the real world.

4 Theoretical System Transformation

The second subtask of this work is to specify and implement a tool for transforming an arbitrary model to a RSQL or a CROM model. Furthermore, it should be able to use a transformed model to change a benchmark to a RSQL benchmark. The first part of this task is the construction of the theoretical process behind the tool. The result of the construction is presented in this chapter. Hence, the chapter is structured as follows: The first section deal with the model transformation and the second section has as content the benchmark transformation.

4.1 Model Transformation

As learned in the previous chapter some informations for the model transformation need to be learned or provided to let the transformation be executed by the computer. In the case of learning, an automatic transformation process could be created, whereas for provisioning the transformation process can only be done manually or semi-automatically. For the transformation process of the tool, it was decided to use provision as a base since a learning algorithm for an arbitrary context or domain would be complex and time-intensive in its execution. In contrast the provision of information could be done by letting the users annotate elements of the model. This way makes direct use of the user knowledge, is simple and intuitive. Therefore, it was decided to use the provision approach. Hence, the structure of the rest of the section is the following. The first subsections will present the possible annotations and their parameters that the transformation tool will provide. Afterwards, the general steps of the model transformation are described. At the end, the model transformation process is applied on the SmallBank schema to prove its functionality.

4.1.1 Annotations

As described previously the annotations are a crucial part of the transformation tool since they provide the information needed for the transformation of the model. The information necessary were listed in Section 3.7. Before the description of which annotation provides which listed information is started the structure of an annotation is discussed.
To easily accesses the informations that an annotation contains an object is used as the annotation structure. This object has a string variable holding the name of the annotation and a list of parameters. A parameter is an object ,as well. Furthermore, it is composed of the parameter name as a string variable and a list of values.
Many different types of annotations are needed for the transformation of a model to a RSQL model and a CROM model. For sake of clarity the types that are needed for the transformation to the NTRT model are discussed first. Afterwards, the types for the transformation to CROM and RSQL are presented. The first group consist of the following annotation types:

- *NaturalType*
- *CreateNaturalType*
- *ExtractNaturalType*
- *RoleType*
- *CreateRoleType*
- *ExtractRoleType*
- *Remove*
- *Rename*
- *Extract*
- *SetRelation*

As the names of these types suggest, they deliver the informations that are needed to deal with natural types and role types during the transformation. More specifically, an object can be marked for the transformation to a natural type or role type with the annotations of the types *NaturalType* or *RoleType*. In contrast, the user can use *CreateNaturalType* and *CreatedRoleType* to instruct the tool to create completely new natural types and role types during the transformation. Note, that annotation types having a name with the prefix *Create* can be added to an arbitrary element of the model to be processed by the tool, since they have no direct relation to any model element.
Annotation types with the prefix *Extract* serve the purpose of highlighting an object. Afterwards, a role type or a natural type is extracted from it by the transformation tool. Which of the two is obtained depends on the rest of the annotation name. Hence, in case of *ExtractNaturalType* it would be a natural type and *ExtractRoleType* would result in the extraction of a role type. In direct relation to these annotation types are the *Extract* annotations. An *Extract* annotation is added to a child of an object to move it to another parent object. The new parent object is specified in the parameter *moveTo* of the annotation. *MoveTo* is necessary due to the fact that multiple natural types and role type can be extracted from an object. Next, the realization of the extraction of a natural type or a role type from multiple objects needs to be explained. It is intended that the user deals with this case by annotating only one object. The children of this object are than extracted and moved by the transformation tool. The children from the other objects should be removed via adding annotations of the type *Remove* to them. The advantage of this is that an annotation of type *Extract* has always the same result i.e., moving of a child from one object to another. In contrast to, sometimes the child is just removed, when the target parent already has a child with same name. Furthermore, a *Rename* annotation changes the name of an object during the transformation and an annotation of the type *Remove* not only leads to the removal of the children of a model element, but to the removal of the annotated object. Moreover, a *SetRelation* annotation allows the manipulation of the transformation trace of a model element. *Remove* annotations have the parameter *mapTo* that is also used to manipulate the trace. A transformation trace is needed for the benchmark transformation and will be explained in the next section. The second annotation type group contains the following types:

- *Compartment*
- *CreateCompartment*
- *Relationship*
- *CreateRelationship*
- *ProxyRoleType*
- *CreateRoleType*
- *Remove*
- *Rename*
- *MoveTo*
- *SetRelation*

The annotation types *Remove* and *Rename* of this group are identical to the *Remove* and *Rename* of the first group. Besides, annotations of type *Compartment* and *Relationship* are used to mark an object that is than transformed to a compartment type or a relationship type respectively. In contrast, the user can make use of *CreateCompartment* and *CreatedRelationship* annotations to specify completely new compartment types and relationships types.

Afterwards, these are created and added to the model by the tool. Another annotation type that is related to compartment types is *MoveTo*. *MoveTo* annotations provide the user of the tool the means to realize the containment requirement of CROM. They provide the informations about which role type should be moved into which compartment type. When the model is transformed to a RSQL model the *MoveTo* annotations do nothing and annotations of the type Compartment and CreateCompartment yield natural types, because RSQL does not include the concept of compartment types yet. The last three types to discuss are *SetRelation*, *ProxyRoleType* and *CreateRoleType*. A *SetRelation* annotation can do the same as a *SetRelation* from the fist group. In addition to that a *SetRelation* annotation of the second group allows the definition of partner attributes for relationship type attributes. This functionality is needed for the generation of relationship type insert statements. *ProxyRoleType* and *CreateRoleType* annotations differ from *RoleType* and *CreateRoleType annotations* of the first group insofar as they have the additional parameters *filledBy* and *moveTo*, to specify the creation of fills-relations and the affiliation to a compartment type. The following list gives an overview of all possible annotation parameters:

- *newName*
- *name*
- *fills*
- *filledBy*
- *newAttributes*
- *attributes*
- *newOperations*
- *operations*
- *mapTo*
- *moveTo*
- *source*
- *target*
- *sourceCardinality*
- *targetCardinality*
- *RelationPartner*
- *RelationshipPartner*

The parameter *newName*, *name*, *newAttributes*, *attributes*, *newOperations* and *operation* are self-explanatory in the context of their annotation. Note, that the *attribute* and *operation* parameter only take strings as values. These are than transformed to attributes or operations in an object during the execution of the transformation tool. The parameters *fills* and *filledBy* are similar, since they specify respectively the source or target of a fills-relation. Furthermore, the information they contain is than used to create the relation. The next parameters *mapTo* and *moveTo* were already explained. The four following parameters are self-explanatory and are only included in annotations of the type *Relationship* and *CreateRelationship*. Last but not least, the parameters *RelationPartner* and *RelationshipPartner* belong to the *SetRelation* annotations and specify the information needed for realizing the described *SetRelation* functionality.
The two last things to consider are the following. First, the annotations are not only based on the investigation result of the previous chapter, but also they are based on the SQL-based Schema Modification Operators (SMOs) from Carl Curino et al [28]. See chapter 6 for more details. Second, the group of annotation types is complete with respect to the results of the investigation of the five benchmarks. This can be proven by comparing the items of the list from Section 3.7 with the annotation types.

4.1.2 Transformation Process

The model transformation process is composed of two phases, because of the following reasons. First, the model is transformed to a NTRT model, whose elements were listed in Chapter 3, as a intermediate stage. As described in the beginning of Chapter 3 such a

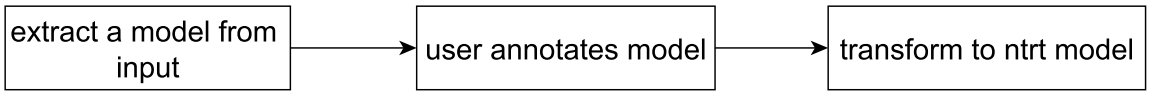

Figure 4.1: Steps of the First Phase

stage is needed, since the definition of compartment types and relationship types is based on the existence of natural types and role types. To realize this intermediate stage with a NTRT model has the advantage that the process complexity per phase is reduced. Another reason for two phases is that the user of the transformation tool does not need to specify all information for the transformation at once. Instead, he is able to concentrate first on natural types and role types and than on compartment types and relationship types. The last advantages of two phases are that iterative processes in general are easier to develop and less error prone. Due to the possibility of developing the steps of the iterative process as components that are only stuck together to finish the implementation. Furthermore, the developer is able to already test the first phase when the implementation of the second phase has not even started.

The steps of the first model transformation phase are depicted in Figure 4.1. Before these steps can be executed the input meta model needs to be specified. When the input does not provide a meta model, the input meta model is given to the tool. The tool the model element classes for annotations and parameters, as well as a super class. The super class is used to deal with the arbitrary input by adding a list of references for annotations and letting all classes of the input meta model inherit from this class. With these preparations done, the execution of the first phase can be started. During the first step the model information from an arbitrary input is extracted and put into the template provided by the input meta model. In the second step the user annotates the extracted model and the tool creates the annotation instances and sets the references from the annotated objects to them. Furthermore, the references are added to the list that is provided by the super class. The information contained in these instances is than used to transform the input model to a NTRT model during the third step.

The steps of the second phase are presented in Figure 4.2. In the beginning of the second phase the user annotates the NTRT model from the first phase and the tool creates the annotation instances and sets the references from the annotated objects to them. Moreover, the references are put into the list of the super class. Afterwards, the user decides whether

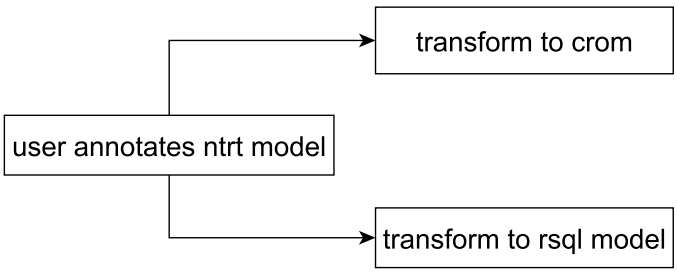

Figure 4.2: Steps of the Second Phase

the NTRT model is transformed to a RSQL model or a CROM model and the tool transforms the model accordingly.
Last but not least, the general model element transformation process begins by using all annotated objects or all natural types and role types during the second phase as a input for the transformation part of the tool. First, the transformation part decides based on the annotations of the input elements how they are transformed. When the decisions were made the associated transforming routines are executed. During the execution the children of an input element are changed based on their annotations, as well. The information needed for the creation of the fills-relations and the setting of the references to the sources and targets of the relationship types are saved. At the end of the transformation process when all input objects, natural types and role type are transformed the saved informations are used to create and set the fills-relations, as well as set the source and target references of the relationship types. Afterwards, the model tree root is generated and all elements of the transformed model are arranged into a model tree. Furthermore, the model transformation process was tested by applying it to the benchmark schemata that were investigated in Chapter 3. The test run of the SmallBank schema is describes in the next subsection.

4.1.3 Example SmallBank

In this subsection the model transformation process is applied to the SmallBank schema [19] to demonstrated its functionality. Hence, the first point of interest is the input for the transformation tool. For the SmallBank schema that are the following SQL create statements.

```
CREATE TABLE Accounts(
  custid  BIGINT  NOT NULL,
  name  VARCHAR(64) NOT NULL,
  CONSTRAINT pk_accounts PRIMARY KEY (custid),
);

CREATE TABLE Savings (
  custid  BIGINT  NOT NULL,
  bal  FLOAT  NOT NULL,
  CONSTRAINT pk_savings PRIMARY KEY (custid),
  FOREIGN KEY (custid) REFERENCES Accounts (custid)
);

CREATE TABLE Checking (
  custid  BIGINT  NOT NULL,
  bal  FLOAT  NOT NULL,
  CONSTRAINT pk_checking PRIMARY KEY (custid),
  FOREIGN KEY (custid) REFERENCES Accounts (custid)
);
```

To start the transformation a SQL meta model needs to be provided, since the input is only composed of create statements. The meta model must contain all informations of the create statements and it should have model elements similar to the following elements: a model class with a list of the create statements as a root for the model tree, a create statement class that contains references to a table, columns and constraints, a table class that includes the name of the table to be created, a column class composed of variables to save all possi-

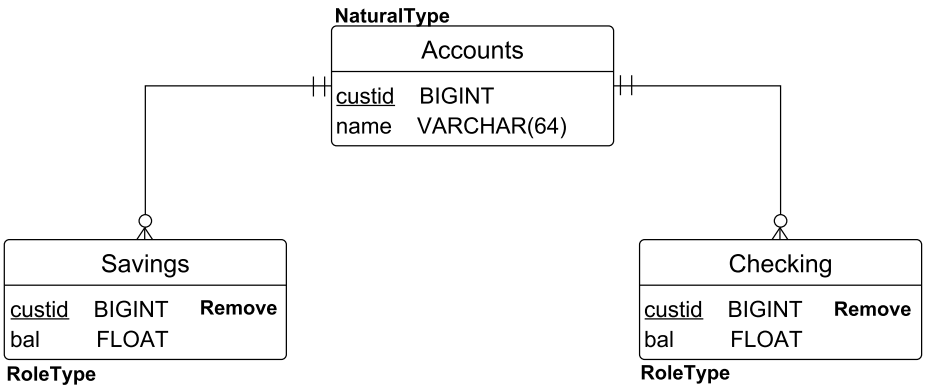

Figure 4.3: Annotated SQL Model of SmallBank

ble column informations and a constraint class that has variables to save the constraint type, referenced columns and so on. After the meta model is created it is provided to the tool. As described in the previous subsection, the tool adds an annotation class, parameter class and the super class as well as inheritance relations between the super class and all user defined model classes to the meta model. With these additions the preparations are finished and the tool starts with the extraction of the model information and puts them into the templates provided by the meta model. The result is depicted in the Figures 3.1 and 4.3.

In the next step the user add annotations to model elements. As explained in Section 3.2 the table *Accounts* should become a natural type and the tables *Savings* and *Checking* should become role types and should be renamed. To accomplish this the user needs to annotate *Accounts* with a *NaturalType* annotation and *Savings* and *Checking* with a *RoleType* annotations. Furthermore, he must add to the *newName* parameters of the *RoleType* annotations the values "SavingsAccounts" and "CheckingAccounts" to realize the renaming. For the fills-relations he must add these names as values to the *fills* parameter of the *NaturalType* annotation. Since the creation of the fills-relations is executed like this, the user should remove the foreign key constraints by annotating each of them with a *Remove* annotation. This is necessary due the fact that the fills-relations include their functionality. Last but not least, as the *custid* informations are provided via the fills-relations the user should remove the *custid* columns of *Savings* and *Checking* by adding *Remove* annotations to them. The schema with all annotations is depicted in Figure 4.3.

The annotations are used in the last step of the first phase to change the input SQL model to a NTRT model. The result of this is presented in the Figures 3.2 and 4.4.

At the start of the second phase the NTRT model is annotated. This time the annotations must provide information to create the compartment type *Bank* and the possible relationship types between *SavingsAccounts* and *CheckingAccounts*, as well as moving *SavingsAccounts* and *CheckingAccounts* into *Bank*. The first of the three can be realized by adding a *CreateCompartment* annotation to *Accounts*, *SavingsAccounts* or *CheckingAccounts* and setting the name parameter of this annotation with the string value "Bank". The creation of the relationship types can be implemented by adding a *CreateRelationship* annotation to one of the three main elements and setting the parameters *name*, *source*, *target*, *sourceCardinality* and *targetCardinality* with the string values "transfer", "SavingsAccounts", "CheckingAccounts", "one" and "one". Last but not least, moving of role types can be realized by annotating each

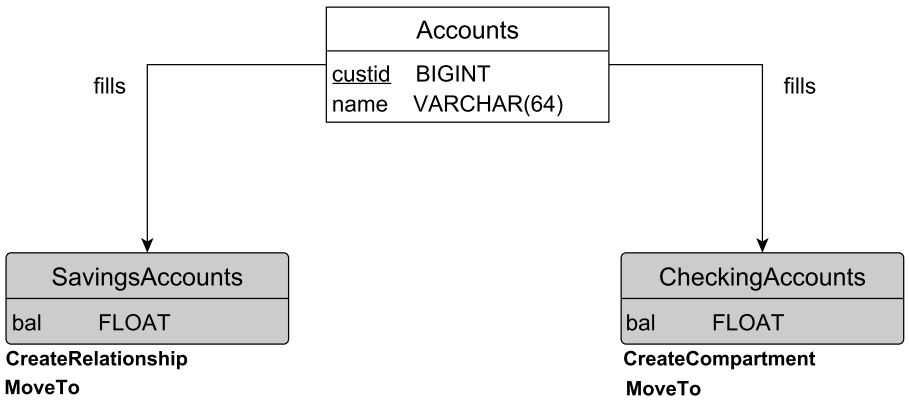

Figure 4.4: Annotated NTRT Model of SmallBank

of them with a *MoveTo* annotation and setting the parameter *moveTo* of both annotations to "Bank". The annotated Smallbank NTRT schema is depicted in figure 4.4.
After the first step the user must decide whether he wants to transform the NTRT schema to a RSQL schema or to a CROM schema. Depending on that decision, the tool calls the associated transformation methods that, in turn process the annotations. As explained in the previous subsection, the processing of *CreateCompartment* annotations differs between a transformation to RSQL and CROM. The result of the transformation to CROM is depicted in Figure 3.4, whereas the result of the transformation to RSQL is presented by the Figure 3.3. Note, that the RSQL figure displays a case where the user does not add the *CreateCompartment* annotation for *Bank* to the model, when the user adds this annotation a natural type *Bank* must be added to the figure to show the complete result.
With all of this it is demonstrated that the model transformation process with the specified annotations can semi-automatically transform the SmallBank schema to a RSQL or CROM schema. Moreover, a comparison with the SmallBank transformation of the Section 3.2 shows that the process can deliver the same results as if a human would do the transformation.

4.2 Benchmark Transformation

The second process behind the transformation tool is a model-driven benchmark transformation process. Hence, before this can be executed the model transformation introduced previously must be finished. Furthermore, a benchmark consists of a schema, data and queries. To transform a SQL benchmark to RSQL all of these need to be transformed. Therefore, this section is structured as follows. The first subsection deals with schema transformation, in the second subsection the data and query transformation are explained, since their transformation process is the same. Last but not least, the last subsection contains the explanation of the query transformation on examples.

4.2.1 Schema Transformation

The transformation of the benchmark schema is done by applying the model transformation to create statements or any other input of the benchmark that contains the schema infor-

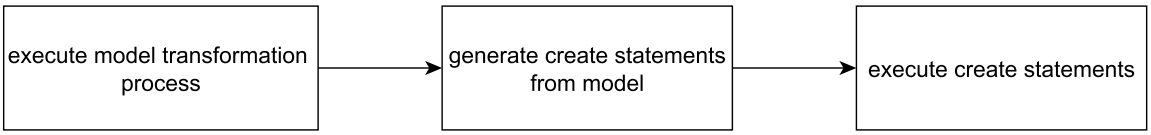

Figure 4.5: Steps of the Schema Transformation

mation. The only problem is that the process result is a model and models can not directly be used by DBMSs supporting RSQL, since they only take RSQL statements and queries as input like SQL DBMSs. To solve, this the tool provides its users with the option of generating RSQL create statements from the transformed RSQL schema. This is possible, because all necessary informations for the schema creation are included in it. In the end, these generated statements only need to be executed to create the benchmark schema. The described steps for the schema transformation are presented in Figure 4.5.

4.2.2 Data and Query Transformation

Like the schema transformation, the data and query transformation is based on a model-driven approach. Hence, the model transformation process needs to be executed for the benchmark schema before transforming them. During the development of the transformation two different approaches were tested, since it was not clear whether transforming the data with the insert statements or generating a data generator is the best approach.

The first approach is depicted in Figure 4.6. The approach consists of two steps, when the benchmark model is already transformed to RSQL. First, RSQL insert template statements are generated from the transformed model. This can be done, because the model contains all informations about tables, columns, and fills-relations necessary to create insert statements. The only thing that the model does not contain is data. Hence, an insert template statement for every natural type, role type and relationship type is needed. These insert template statements differ from normal insert statements in that the *values-clause* contains question marks instead of values. The question marks are important for the second step of the approach, because they will be replaced with direct values in the data generator. Moreover, it is theoretical possible to generate the data generator with the transformed model and the insert templates, since the model contains the types of the columns. Another requirement for the generation would be that the user provides intervals for every data type to the tool. When these are fulfilled, it is possible to create standard code for every data type that can replace the question marks in the insert templates. In the end, standard code snippets

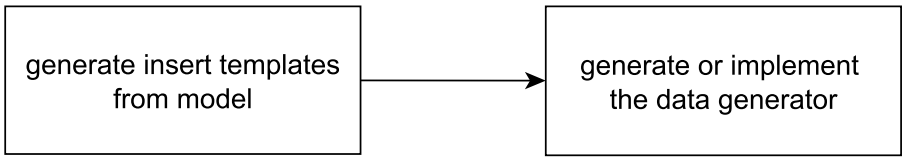

Figure 4.6: Steps of the First Data Transformation Variant

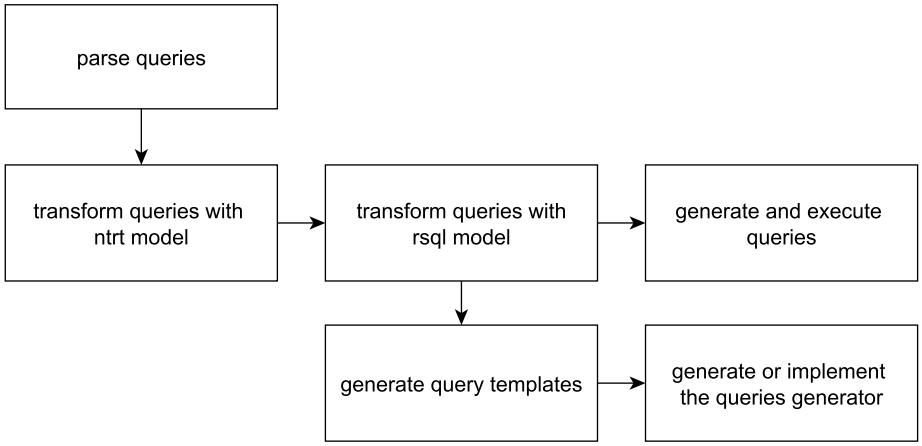

Figure 4.7: Steps of the Query Transformation

only need to be put together based on the model informations to obtain a data generator. The generating approach was not tested during the work, because the implementation of standard code for all data types is very time. Therefore, only the variant, where the data generator is implemented, was tested. Overall, the advantages of the first data transformation approach are high reusability and the problem of transforming insert statements, where from the table two natural types or role types were extracted, does not need to be considered. The problem described in the latter arises due to the fact that no satisfying way was found to specify if a insert statement of the table represents on the RSQL level one of the following tuples: natural type with extracted role type one, natural type with extracted role type two, extracted natural type one with role type or extracted natural type two with role type. Note, that the approach has a high reusability, because the insert templates are not bound to data. Hence, it is possible to use them in other software applications with the same schema as the benchmark. At the end of the test, it was decided to not use the first approach for the tool, because its degree of automatization is to low.

The data transformation approach that is used for the tool is the same approach that is used for query transformation, since data can be transformed with the associated insert statements. A visual representations of the approach is displayed in Figure 4.7. First, the tool parses the input insert statements, update statements, delete statement and select queries. Afterwards, the parsed information are put into a query graph. A query graph is a template provided by a SQL meta model. Hence, the user must provide this meta model to the tool before this process is started. To realize the next steps, a transformation trace was added to the model transformation process. The trace is needed for the statement and query transformation. Furthermore, it is realized by saving the old names of tables, natural types, role types, columns, operations and parameters in the model. The trace from the input to NTRT is saved in the NTRT model and from NTRT to RSQL it is saved in the RSQL model. This makes it necessary for the query graph to be changed to the NTRT stage and than be transformed to the RSQL stage. The second and third step of the query transformation approach deal with this. In the second step, the input query graph and the NTRT benchmark schema are used as input for a transformation module. The module uses both of them to create a NTRT query graph. The third step differs form the second step in the way that the associated transformation module takes as input the NTRT query graph from the second step and the

RSQL benchmark schema to create a RSQL query graph. After the third step, it is possible to take two different routes. During the first route, query templates are generated and a query generator can be implemented or generated from the templates and the transformed schema. This route is not used for the tool due the same reason the first data transformation approach is not used. Moreover, in some cases benchmark procedures create queries based on different variables like thread count. This makes it difficult to complete transform the procedures with templates. Hence, the second route is used for the tool. In the second route, RSQL insert statements, update statements, delete statements and select queries are derived from the RSQL query graph by using the RSQL model. Afterwards, these are executed by a RSQL DBMS. The advantages of this route are that the statement and query transformations can be done automatically and the dependency of the statement and query creation on variables in a procedure can be ignored. The latter is possible, since the queries and statements can be gathered and transformed in a warm up phase before the benchmark is applied. Hence, the procedure statements and queries needed for a specific benchmark run are transformed in any case. The disadvantages of the second route and the complete approach are explained in the limitation section.
Last but not least, the data of an insert statement is transformed in the second and third step. In most cases the data is only moved to the correct natural type or role type. However, a prefix is added to the primary key data that is provided by insert statements of natural types or role types, when for example a natural type was extracted from two role types and primary key columns of the roles types were moved to the natural type. Furthermore, in such cases the type of the primary key columns is changed when necessary. The prefixes are needed due to the fact that primary key values need to be unique. This is a problem, because it is quite likely that the role types had tuples with the same primary key values. Moreover, the user can use *SetRelation* annotations with set *RelationPartner* parameters, when he wants to fill columns that were added via annotations with data of a already existing columns. This type of annotation relates existing columns with the new columns by letting the transformation process add the name of the existing column to the trace of the new column. When the insert statements are transformed this lead to the copying of data from existing columns to new columns.

4.2.3 Statement and Query Transformation Examples

In this subsection the statement and query transformation process introduced previously is demonstrated by applying it to statements and queries of SmallBank. The first example to be described is the transformation of the following insert statement. Note, to understand this and the following examples it is necessary to read 4.1.3.

```
INSERT INTO Savings (custid,bal) VALUES (1, 5000);
```

When the user has provided a SQL meta model to the tool, the tool would begin the transformation by parsing the statement and putting the parsed information into a query graph. Afterwards, the SmallBank NTRT schema and the query graph are used as input for the transformation module of the second step. In the module, the following things are done by using the transformation trace: the table name *Savings* is changed to *SavingsAccounts*, *custid* is removed from the list of column names and from the values list *1* is removed. Furthermore, to consider the fills-relation between *Accounts* and *SavingsAccounts* a new insert statement entry is created in the query graph, when the input does not contains a *Accounts* insert statement with the *custid 1*. The table name of this new entry is *Accounts*,

the column name list contains *custid* and *name*, and the values list has as entries *Savings_1* from the *Savings* insert and an empty string. The empty string is a default value for columns with a string data type that the tool uses in such cases. Note, the user can specify the default value for new columns added by him.
The transformation of the third step does not lead to any changes in the query graph. Hence, the generation of the RSQL insert statements has the following result:

```
INSERT INTO NT Accounts (custid,name) VALUES (Savings\_1, );
INSERT RT SavingsAccounts(bal) VALUES (5000) INTO ( NT Accounts nt1 PLAYING
  (RT SavingsAccounts rt1)) WHERE nt1.custid = Savings\_1 ;
```

Note, that the config-expression of the *SavingsAccounts* insert statement is derived from the fills-relation between *Accounts* and *SavingsAccounts*. The fills-relations are saved in the SmallBank RSQL schema. Hence, the schema is necessary not only for the query graph transformation, but also for the generation of the RSQL statements and queries. Furthermore, to generate the where-clause of a relationship type insert statement the user must add to the model at least two *SetRelation* annotations with set *RelationshipPartner* parameters during the model transformation process. These annotations relate relationship type attributes with columns of the source and target role type.s When the annotations are not added than the relationship type insert statement can not be correctly generated. The next example deals with the transformation of the following select query:

```
SELECT custid, bal FROM Savings WHERE bal = 5000;
```

First, the tool parses this query and creates the query graph based on the provided meta model. Afterwards, in the scope of the second step the query graph is changed in the following way. In the from-clause *Savings* is replaced with *SavingsAccounts*. Furthermore, in contrast to the insert statement a new select entry is not generated for the fills-relation between *Accounts* and *SavingsAccounts*. Instead, the name *Accounts* is added to the from-clause. This is possible, because a select query can have multiple tables as a target. Hence, to consider the fills-relation the respective natural type names or role type names only need to be added to the from-clause. Theoretically, Accounts does not need to be added, since the RSQL schema is used for the generation of the query. Nevertheless, it is done, because when the user wants to compare SQL queries with RSQL queries then he needs to generate the SQL queries from the NTRT query graph to get a correct comparison and the adding of *Accounts* makes the generation of these SQL queries easier. Moreover, in the select-expressions, order-by-clause, group-by-clause and other select query parts only the column name must be change, since new select entries are not created.
The third step does not lead to any changes in the query graph. Consequently, the result of the generation is the following:

```
SELECT custid, bal FROM (NT Accounts nt1 PLAYING (RT SavingsAccount rt1))
 WHERE bal = 5000;
```

As the result shows the only the the from-clause changes from SQL select to RSQL select. The tool generates the RSQL from-clause by iterating over the from-clause entries of the query graph. For every entry it generates a config-expression. When one config-expression is generated, the tool checks whether a config-expression that represents exactly the same fills-relation exists and when one exits the generated config-expression is thrown away. This is done in this way as a trade-off for making the generation of SQL queries from the NTRT query graph easier. Moreover, the tool iterates over the from-clause entries a second time to generate for entries that represent a relationship type a relating-clause.

The transformation of update and delete statements only differ from the select query and insert statement transformation in one point. This point is that when through fills-relations a new update entry or delete entry is created the expressions in the set-clause and where-clause are moved in the query graph to the statement entry where the associated column is contained.

4.3 Limitations

In this section the limitations of the model transformation process and the benchmark transformation process are described. The model transformation process is limited in the areas of constraints, additional attribute and operation informations, and indexes. The first area is limited due to the fact that the model transformation should work with arbitrary input. This is a problem, because constraint is a broad term that mean different things in different areas of computer science. For example are constraints in the context of SQL things that are added to a create statement to specify primary keys, foreign keys and so on. In contrast to that, Java constraints are annotations that are used to validate a variable or the return value of a method. This makes it complicated to create an interface for providing a new constraint to the tool. Hence, the user can not add constraints to the model by annotations during the transformation. The limitation of the second area is based on exactly the same problem. For example in Java visibility informations are added to the attributes and methods and in SQL informations like a column is not allowed to have *NULL* as a value are added to the column definitions. Overall, this leads to the restriction that the model parser of the tool can only extract the additional informations of SQL columns. Furthermore, the user can not add additional information to specification of new operations and attributes. Last but not least, SQL indexes are ignored by the transformation tool, because their usage in context of RSQL is not researched yet.
The limitations of the benchmark transformation the following. First, the informations that are not parsed and processed by the model transformation are ignored for the benchmark transformation. Second, only queries and statements of natural type or role types that have not multiple fills-relations are transformed and generated, since the tool has not the means to let the user specify which tuple a statement or queries represents yet. This problem was described more in detail in Section 4.2.2 and it is a big disadvantage of the statement and query transformation approach used in the transformation tool. Moreover, execution code of the benchmark can not be transformed by the tool, because to be able to do this the tool must have a parser of every programming language and the time of the work was not enough to gather or implement these.

5 Implementation of the System Transformation

The second subtask of this work is to develop and implement a tool for transforming an arbitrary model to a RSQL or a CROM model. Additionally, it should be able to use the transformed model to change a benchmark to a RSQL benchmark. The second part of this task is the implementation of the transformation tool. A overview of the implementation is provided in this chapter. Hence, the following sections deal with the architecture, the parsing component, the annotation component, the transformation component of the tool and the limitations of its implementation.

5.1 Architecture

It was decided to implement the tool as an Eclipse plug-in [29] written in Java [30]. The reasons for this decision are the following: saving of implementation time due to the provision of a graphical user interface by Eclipse, automatic refactoring functions, and programming support. Furthermore, many custom plug-ins like Eclipse Modelling Framework (EMF) [31] or ATL [12] exist for Eclipse that can be used as a part of the tool. Last but not least, Eclipse and Java are platform independent. Hence, as a plug-in the tool is platform independent, as well.
As presented in Figure 5.1, Eclipse is the main element of the tool framework. The other part of the framework is the Eclipse plug-in MoDisco [32]. MoDisco provides a framework for the development of model-driven tools that are used to modernize software. The provided framework consists of four big components. The first component is the discoverer and work flow manager. Discoverers or model parsers are tools to extract model informations form different kind of inputs. A developer can use the manager to specify and register such discoverers. Furthermore, the manager provides the possibility of chaining discoverers, transformations and generations in a work flow. The second component is the model browser and the third component contains deals with querying and customization for the creation of views. A meta model library is the fourth component. The library contains for example a Java meta model. Note, that MoDisco is based on EMF. Hence, the model and meta models that are processed and managed by MoDisco are Ecore files. Ecore is the core (meta-)model at the heart of EMF. It allows expressing other models by leveraging its constructs. The model browser and the discoverer manager of MoDisco are used in the transformation tool of this work. Therefore, the tool will deal with Ecore models and meta-models. To be more specific the manager incorporates the parsing component of the tool into the framework and supports its specification and implementation. In contrast to that, the model browser is part of the annotation module and is used as a graphical user interface by it. Moreover, the main reason for using MoDisco in the tool is the meta model library, since it is a good base for parsing arbitrary input.
Besides the parsing module, the transformation tool is composed of a transformation com-

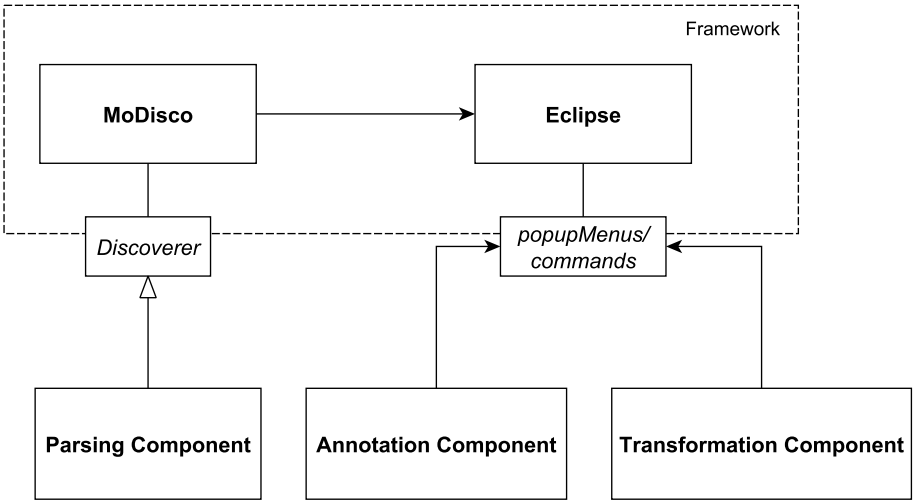

Figure 5.1: Architecture of the Transformation Tool

ponent and an annotation component as the Figure 5.1 displays. Both of them are plugged in the framework through the Eclipse extension point *command*. This point is called *popupMenus* in older Eclipse versions and is used to extent the functionality of context menus owned by Eclipse or other plug-ins through adding actions to them. Moreover, MoDisco is plugged into Eclipse over the *command* extension point, as well.

5.2 Parsing Component

The first part of the transformation tool is the parsing component. It consists of two parts. The first part is the discoverer that serves as a interface to the framework and is realized with MoDisco. The second part is the SQL parser that is implemented with Xtext [33]. Xtext is a framework for development of programming languages and domain specific languages. Moreover, it is a Eclipse plug-in and therefore easy to use. A good documentation and many tutorials are other advantages of Xtext. Additionally, it provides a code generator for parsers, compilers, interpreters and other components of a language infrastructure. Hence, SQL needs to be specified in a Xtext file to implement the SQL parser for the tool, since Xtext generates from the file the code of the parser. A SQL Ecore meta model is generated, as well. This model is used in the transformation module. Note, that through a Xtend file [34] a post processing is applied to the SQL meta model. During the post processing the model elements *SuperClass*, *SuperClassAttribute*, *SuperClassOperation*, *SuperClassParameter* and *SuperClassConstraint* are added to the meta, model as well as elements for the annotations and annotation parameters. The reasons for adding *SuperClass* and the elements for annotations and their parameters were explained in Subsection 4.1.2. The other elements are added to make the access to attributes, operations, operation parameters and constraints easier in the transformation component. Furthermore, Xtend is a flexible and expressive dialect of Java and in this case it is used to create extension for the post processing of Xtext. Last but not least, the two parts of the parsing component are connected by using a Xtext SQL parser instance as the injector of the discoverer.

5.3 Annotation Component

The annotation component is second part of the transformation and is composed of actions for transforming a model to a NTRT model, actions for transforming a NTRT model to a RSQL or CROM model and a scanner with a scan term library. The actions for the first and second phase of the model transforming process are added to the context menu of the MoDisco model browser. They have basically the same functionality namely adding annotations to a model element, editing of existing annotations and removing of annotations. Nevertheless, they are different in some aspects. First, the first phase actions can only be used for input models and the second phase actions for NTRT models. Second, different sets of annotations are available during the execution of the first and second phase actions. The reasons for that were described in 4.1.1. Third, the scanner is only used in the first phase, when an annotation is added to a model element. Than the scanner searches for the attribute, operation, operation parameter and constraint information of this element. Afterwards, the informations are saved respectively in model elements of the types *SuperClassAttribute*, *SuperClassOperation*, *SuperClassParameter* and *SuperClassConstraint*. Furthermore, these elements are referenced by variables in a SuperClass instance. During the search process the terms of the term library are used to find the model elements that contain the informations. The advantage of the scanner is that its usage allows the implementation of the transformation component to be less complex. Moreover, the tool provides the user with the possibility of adding terms to the scan term library and removing terms from it. During the second phase the scanner is not used, since the input model elements and the location of their children information are known by the tool. This is because the result of the first phase is saved in a model with a NTRT Ecore meta model that was created for the annotation and transformation component with EMF [31]. Besides the elements for natural types, role types, attributes and so on this meta model has the class *SuperClass* and the elements for annotations and their parameters, as well.

5.4 Transformation Component

The last part of the transformation tool is the transformation component. This component is structured as the follows. The first unit contains the actions for transforming an input model to a NTRT model and a NTRT model to a RSQL model or a CROM model. To realize these actions a RSQL Ecore meta model was created and the CROM meta model by Thomas Kühn [35] is used. Furthermore, every action has an associated ATL transformation module. The actions are added to the package explorer, a context menu of Eclipse, and the ATL transformation modules implement the model transformation process described in 4.1.2 through ATL rules.
The second unit deals with the actions for transforming a benchmark. The meta model for a query graph instance is the SQL Ecore meta model generated by Xtext. Furthermore, every actions is associated with an ATL transformation module and like the actions of the first part they are added to the package explorer. In this case the ATL transformation modules implement the query graph transformation of the benchmark transformation process explained in Section 4.2. To realize this the model transformation trace is needed. Hence, the modules have the NTRT model or RSQL model as an additional input depending on the transformation stage.
The last unit consists of actions for generating the RSQL create, insert, delete and update

statements, as well as the RSQL select queries from the the transformed query graph. Like the previous actions these are added to the package explorer. Furthermore, each of them has an associated ATL query. These queries request from the query graph the root of the graph and therefore the complete graph as a string. To construct the string helper functions are used to move through the graph. Moreover, they are used to transform statement and query entries to RSQL statement or query strings. These are added to the output string. When the helper functions have iterated through all graph nodes than the output string is given to the action. Afterwards, the action takes the string and writes it to a SQL file.
Last but not lest, ATL was chosen for the implementation of the transformation based on three reasons. The first reason is that ATL is able to do *model to model transformation* and *model to text transformation* via its modules and queries. The second reason is that many examples of application using ATL exist in the internet. Hence, the learning of ATL is easier as the learning of other model transforming languages. The last reason is that in contrast to other languages ATL is old and therefore it has reached a certain degree of stability.

5.5 Limitations

This section deals with technical limitations of the transformation tool. One technical limitation was found during the testing of the tool. The found limitation is that the Xtext SQL parser can not parse big files, since Eclipse throws Java heap source exceptions when for example Xtext tries to parse a file of the size of 156 MB. Some fixes and workarounds were applied to solve this problem, but none of them worked. Hence, the Xtext parser needs to be replaced to use the tool for the transformation of applications that contain big files. Furthermore, the scanner searches for the information of children only one step above and one step below the annotated element. The scanner was implement this way, because in most cases the children belonging to the annotated object can be found in this scope. Nevertheless, in future works the scanner should be developed further to be able to deal with all kinds of models.

6 Related Work

In this chapter related works are described and compared with this work. Hence, the sections of this chapter have the following content. The first section contains a description of the requirements that are used in the comparison. Afterwards, in the second section a overview of related works is given and some of the related works are introduced. The last section deals with the comparison.

6.1 Requirements

A set of requirements is needed to compare the works with each other. Therefore, such a set is introduced in this section. It consists of the following requirements:

- Software Levels: The levels that are changed during the transformation.
- Behaviour Preservation: This requirement is about whether the complete behaviour of the software is preserved during the transformation or only the expected behaviour. Expected behaviour is different from the complete behaviour, since sometimes behaviour is added to the software for example by refactoring and updating that is not expected by the developer. Such behaviour is not included in the term expected behaviour.
- Automatization Level: This describes how much of the transformation process is executed manually and how much is done automatically.
- Portability: The level of dependency the transformation process and its implementation have on a specific platform.
- Usability: The degree of difficulty to use the transformation process and its implementation.
- Overhead: The extent of things need to be done in the approach compared to a brute force approach.
- Reusability: The scope of inputs, contexts and domains the transformation process can be used on.
- Extensibility: The degree of difficulty to extend and refactor the transformation process and its implementation.
- Complexity: The complexity of the transformation process.
- Interoperability: The degree of which the transformation process and its implementation can be used in combination with other transformation processes.

These requirements were derived from the related works, the ISO/IEC 9126-1 [36] and 25010 [37] standards.

6.2 Overview of the Related Works

This section contains a overview of the related works and some related works are introduced. The core topic of this work is the shift of paradigm of software from object-oriented to role based. The shifting of the paradigm of software in general is a big topic in computer science. The most research on it was done in the area of transforming software from the procedural paradigm to the object-oriented paradigm [38, 39, 40, 41]. Furthermore, works exist that deal with the transformation of a software system in the way that the paradigm of it is changed from object-oriented to compartment-based [42, 43]. In contrast to these, only one work by Friedrich Steimann [45] was found in the area of transforming an object-oriented software to role-based software. Transformation approaches are not only used for paradigm shifts, but also for the improvement of software. For example Ladan Tahvildari et al. [44] has developed a framework that uses transformation approaches to improve the quality of object-oriented code during the re-engineering of software. Another example of this is the work of Munawar Hafiz et al. [46], where programs are transformed tot improve their security. Moreover, database can be improved by transformations, as well. A example for that is the work of Carl Curino et al. [28]. All in all, many works about the usage of transformation approaches in particular for paradigm shifts exist. As not all of them can be introduced here, the works by Carl Curion et al., Friedrich Steimann, Eunjoo Lee et al. [42], and Ying Zou et al. [38] are presented in the rest of this section. These are compared with this work in the next section.

Carl Curion et al. introduced in their paper with title "*Automating the database schema evolution process*" [28] the PRISM++ system. The PRISM++ system is a advancement of the PRSIM system that is developed by them, as well. Furthermore, it can semi-automatically migrate or evolve a database and automatically rewrite queries and updates that are associated with the database. This is done by using SQL-based schema modification operators (SMOs) and integrity constraints modification operators (ICMOs). SMOs describe how the tables in the old database schema are modified into those of the new database schema. The user of PRSIM++ specifies them by writing the specifications in SQL from that the PRSIM++ extracts the SMOs or writing them directly. Carl Curion et al. extended SMOs with ICMOs, because SMOs do not capture integrity constrains evolution and therefore, they don't support the evolution of updates. In contrast to SMOs, ICMOs can only be specified by the user through writing them directly. Besides these, PRISM++ executes the rest of the migration automatically by using of the SMOs and ICMOs to transform the schema. Afterwards, it checks and adapts the data tuples and rewrites the queries and updates based on the schema, SMOs and ICMOs. For the rewriting the following steps are executed. First, the SMOs and ICMOs are inverted. Than these are translated into an equivalent logical schema mapping expressed in the language of DED. In the last step, the rewriting is done using the DEDs by means of a chase-based algorithm. The algorithm is extend in their paper due to the fact that ICMOs can lead to negations that the chase-based algorithm can not handle.

Friedrich Steimann introduced in his paper "*Refactoring to Role Objects*" [45] a tool that automatically refactors legacy object-oriented code to role-objects. The role-objects are created by applying the role-object pattern to object-oriented code. As Steimann tested his tool, he noticed that the results of using the standard role-object pattern were not good. The code that the tool produced was hard to read and maintain. Furthermore, the precondition for refactoring were so strong that in practice the tool could not apply the pattern. To deal with this problem Steimann developed a light-weight variant of the role-object pattern and imple-

ment it in his tool. The tool was than tested again with the new pattern. The result of this test were that in comparison to the test results of the standard role-object pattern the applicability on average were three times higher and the line of codes produced were 3.7 times lower. The refactoring process used by the tool consist of identifying the roles and entities, extracting the core(standard pattern) or the state(new pattern) from the entities and set the relations through delegations and inheritances.
Eunjoo Lee et al. described in the paper "*A Reengineering Process for Migrating from an Object-oriented Legacy System to a Component-based System*" [42] a reengineering process for migrating object-oriented legacy systems to component-based systems. The introduced process is divided into two phases. In the first phase, relationships like inheritance and composition are used to create the basic components. This realized by creating functional units from the classes having composite relationships. Furthermore, classes connected via inheritance are put in the same component due to their strong cohesion. At the end of the first phase all redundant and needless classes are removed. During the second phase the components are refined by clustering similar components into one big component based on software metrics like cohesion or connectivity strength.
Last but not least, a generic re-engineering source code transformation framework was presented in the paper "*Incremental Transformation of Procedural Systems to Object Oriented Platforms*" [38] of Ying Zou et al. The frame work supports the incremental migration of procedural legacy systems to object oriented platforms. Furthermore, it is composed of three parts. The first part is a source code representation framework. It processes the input procedural code in the way that the abstract syntax tree of the code is parsed into XML documents by using a generic domain model for procedural languages. The second part is a set of transformations. These are used for the identification of object models in specific parts of the procedural source code. Moreover, the second part includes a clustering algorithm. This algorithm uses the identification results to decompose the procedural program into a set of smaller components. Note, this part is applied first during a migration. Afterwards, the first part generates a XML document representing the associated abstract syntax tree for every small component. The third part is a incremental migration process that is used at the end of the migration. This process is applied to the generated XML documents to gradually transform them into a object-oriented system by composing the system model step-by-step. Thereby, one step means that classes are extracted from one XML document and related to each other through inheritance, delegation, and so on.

6.3 Comparison

The transformation process that was created in this work changes object-oriented software to role-based software. Hence, it belongs in the same area as the work by Freidrich Steimann [45]. Nevertheless, they are different. For example Steimanns transformation tool is applied at the code level of a software application. In contrast to that, the tool of this work operates on the model level, as well as on the code level, since the model is used to change code or rewrite statements and queries. Moreover, the tool is similar to the PRSIM++ System by Carl Curion et al. [28] with regards to the requirement *Software Levels*. This fact is presented in the Table 6.1. Note, that the PRISM++ system is applied to model and code level, as well.

The next comparison requirement listed in the table is *Behaviour Preservation*. In this area the tool of this work, the PRSIM++ system, Steimanns tool, and the process developed by

Requirement	Curino	Steimann	Lee	Zou	This Work
Software Levels	model,code	code	model	code	model,code
Behaviour Preservation	expected	expected	complete	expected	expected
Automatization Level	semi-automatic	automatic	automatic	automatic	semi-automatic
Portability	■	■	-	-	■
Usability	⊟	■	■	■	⊟
Overhead	■	□	□	⊟	■
Reusability	⊟	□	⊟	□	■
Extensibility	■	⊟	⊟	⊟	■
Complexity	■	□	□	⊟	■
Interoperability	⊟	⊟	⊟	⊟	■

■ : high, ⊟ : middle, □ : low

Table 6.1: Comparison of Transformation Processes

Ying Zou et al. [38] only preserve the expected behaviour, since all of them change the inner structure of a software unit and the outer structure of the application or system that is transformed. The transformation process of Eunjoo Lee et al. [42] by comparison only transforms the outer structure. Hence, it preserves the complete behaviour.
Next, the degree of automatization is to compare. The transformation tool of this work was designed in the way that the user must add annotations to models during the execution of the process. Therefore, it is semi-automatic. The PRSIM++ system is also semi-automatic due to the usage of SMOs and ICMOs. The other transformations are fully automatic. Furthermore, the automatization has a direct influence on the usability and the overhead of a transformation process, because manual parts like annotations, SMOs and ICMOs must be learned by the user and managed by the transformation tool or system. Hence, the usability of the tool and the PRISM++ system is middle and the overhead is high. As a compensation for the bad usability and overhead the tool has the highest reusability of all works that are compared here. The high resuability is the result of using for the transformation process of it a model-driven approach that is executed partially manual through the annotations. Furthermore, the tool can take arbitrary inputs, because of the annotations. The other transformation processes were created for specific inputs. Hence, the resusability of them is lower.

The next requirement to discuss next is *Interoperability*. The tool has a high interoperability. In comparison the PRSIM++ system has a lower interoperability, because it completely transforms a database and so a collaboration with other systems or tools is only needed in the context of a framework for the transformation of a complete software system. Furthermore, the processes of the other works have a low interoperability, since their inputs can not be arbitrary and their transformation steps are mostly depended on each other. The dependence of the steps is the reason for their low extensibility, as well. In contrast to that the PRISM++ system and the tool are modular and therefore their parts can switched and extended more easily.
Last but not least, the comparison of the works in terms of *Complexity* and *Portability*. The transformation tool and the PRISM++ system have the highest process complexity due to

their manual parts. The portability of them and Steimanns tool are also high, because all of them were implemented with Java. For the transformations by Ying Zou et al. and Eunjoo Lee et al. statements regarding portability can not be made, since these were not implemented.

7 Conclusion and Future Work

The goal of this work was to specify and implement a tool for changing the paradigm of software from object-oriented to role-based . To reach this goal it was decided that the process of tool would be a model-driven semi-automatic transformation. Hence, to get a base for the specification of the transformation the schemata of the benchmarks SmallBank [19], SEATS [20], TPC-C [21], TPC-H [22], and AuctionMark [25] were investigated in the first part of the work regarding. The aims of it were to learn how a model changes when it is transformed to a CROM model or a RSQL model, to gather the informations that are necessary for the transformation and to determine the degree of automatization that can be reached. During the investigation, it became clear that when a computer should do the transformation information about context, domain, and meaning of objects needed to be provided or learned.

The second part of the work deals with the specification and implementation of the transformation tool. In the specification the results of the first part were incorporate in the way that the informations needed for the transformation will be added to the input model by annotations. The reason for that is that learning processes for arbitrary inputs are complex and time consuming. Furthermore, the user of the transformation needs to add annotations twice during the process, because before the model is transformed to a CROM model or a RSQL model it is changed to a NTRT model. The reason for the NTRT interim stage is that before compartment types and relationship types can be defined natural types and role types need to exist. Overall, the first phase of the transformation process, the model transformation, was specified as a iterative semi-automatic procedure. The functionality of the model transformation was demonstrated by transforming the SmallBank [19] schema with it.

The second phase of the process is a benchmark transformation process. This was specified as a example for a software system transformation. For the benchmark transformation three parts were defined and all of them make use of the transformed model. The first part is the schema transformation that only consists of the generation of RSQL create statements from the transformed benchmark RSQL schema. The second part is about data transformation and its work flow is same as the work flow of the third part that deals with query transformation. The reason for that is that the data is transformed with its associated insert statement. Moreover, the statement and query transformation is composed of the following steps: First, the statements and queries are parsed into a graph. Afterwards, the graph is transformed with the model to a RSQL graph. In the last step, RSQL insert, delete and update statements, as well as select queries are derived from the RSQL graph. As it was the case with the model transformation the functionality of the specified benchmark transformation process was demonstrated by the transformation of SmallBank statements and queries. Note, that all investigated benchmarks were transformed with the transformation tool, but only the transformation of SmallBank is described.
At the end of the work the specifications of the two processes were implemented into a Eclipse [29] plug-in that is composed of a parsing component, a annotation component and a transformation component. All of these have a interface to the framework that consist of

MoDisco [32] and Eclipse. Furthermore, for the implementation of the parsing component MoDisco and Xtext [33] were used. In contrast to that the implementation of the annotation component makes only use of MoDisco. The transformation component was implemented with ATL [12]. These implementations complete the transformation tool. Hence, the goal of this work is reached. A transformation tool was specified and implemented that is able to change the paradigm of software from object-oriented to role-based. Furthermore, the functionality of the tool was demonstrated, as well.
The last topic to discuss in this chapter are future works. One possible way to expand this work is to specify and add verifications to the transformation process to verify whether the model was correctly transformed in term of constraint conditions. A example for such a condition would be that in CROM every role type needs to be contained by a compartment type. Furthermore, a solution should be implemented for the problem of transforming statements and queries of tables that are natural types or role types with multiple fills-relations after the schema transformation. One solution for this would be as follows. First, the annotations of the types *NatualType* and *RoleType* would get a new parameter. Second, the user can set this parameter with a decision expression that is used to decide, which dynamic tuple a statement or query of the annotated object represents. A full automatic transformation approach would be another point to consider for future works. To realize this the transformation tool must get the information that are provided by the annotations must on its own. Moreover, the tool created in this work must be tested for big systems and code transformation should be developed. Last but not least, the found limitations must be removed as far as possible in particular the input file size limitation of the Xtext parser and the constraints of CROM should be incorporate in the transformation process.

Index of Abbreviations

OO	object-oriented
CROM	Compartment Role Object Model
RoSI	Role-based Software Infrastructures for Continuous-Context-Sensitive Systems
CROI	Compartment Role Object Instances
RSQL	Role Structured Query Language
DBMS	Database Management System
DDT	Dynamic Data Types
DQL	Data Query Language
DML	Data Manipulation Language
DDL	Data Definition Language
MDSD	Model-Driven Software Development
LOC	Lines of Code
OCL	Object Constraint Language
SEATS	Stonebraker Electronic Airline Ticketing System
OLTP	Online Transaction Processing
OLAP	Online Analytical Processing
SMO	SQL-based Schema Modification Operators
EMF	Eclipse Modelling Framework
ICMO	Integrity Constraints Modification Operator

Bibliography

[1] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. May 2000. Downloadable from : https://www.fernuni-hagen.de/ps/pubs/DKE-35-1.pdf [visited on 08.09.2015].

[2] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. 1997. Downloadable from :http://hillside.net/plop/plop97/Proceedings/riehle.pdf [visited on 08.09.2015].

[3] Thomas Kühn, Stephan Böhm, Sebastian Götz, Christoph Seidl and Uwe Aßmann. Combined Formal Model for Relational Context-Dependent Roles. 2015. In Submission in Software Language Engineering ACM.

[4] DFG Research Training Group 1907. RoSI.https://wwwdb.inf.tu-dresden.de/rosi [visited on 08.09.2015].

[5] Tobias Jäkel, Thomas Kühn, Hannes Voigt and Wolfgang Lehner. RSQL - A Query Language for Dynamic Data Types. July 2014. Downloadable from :http://dl.acm.org/citation.cfm?id=2628246 [visited on 08.09.2015].

[6] Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt and Wolfgang Lehner. Relationships for Dynamic Data Types in RSQL. 2015. Downloadable from : http://btw-2015.de/res/proceedings/Hauptband/Wiss/Jaekel-Relationships_for_Dynamic_D.pdf [visited on 08.09.2015].

[7] Edgar Jakumeit, Sebastian Buchwald and Moritz Kroll. GrGen.NET:The expressive, convenient and fast graph rewrite system. 2010. Downloadable from: http://link.springer.com/article/10.1007%2Fs10009-010-0148-8[visited on 08.09.2015].

[8] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahe and Didier Vojtisek. Kermeta Language Reference Manual. November 2010. Downloadable from: http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.documentation/build/pdf.fop/KerMeta-Manual/KerMeta-Manual.pdf [visited on 08.09.2015].

[9] Object Management Group. QVT-R. https://wiki.eclipse.org/MMT/QVT_Declarative_Languages#QVTr [visited on 08.09.2015].

[10] Object Management Group, Query/View/Transformation Specification v1.2. 2015. Downloadable from: http://www.omg.org/spec/QVT/1.2/ [visited on 08.09.2015].

[11] Object Management Group. MOF Model to Text Transformation Language, v1.0. 2008. Downloadable from: http://www.omg.org/spec/MOFM2T/1.0/ [visited on 13.09.2015].

[12] ATLAS INRIA & LINA. ATL. https://eclipse.org/atl/ [visited on 08.09.2015].

[13] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. The UML-RSDS toolset. 2012. Downloadable from: http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/ [visited on 08.09.2015].

[14] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya and P.VanGorp. Evaluation of model transformation approaches for model refactoring. March 2013. Downloadable from: http://www.sciencedirect.com/science/article/pii/S0167642313001871 [visited on 08.09.2015].

[15] DBZ, Deutsches Benchmarking Zentrum. http://benchmarkingforum.de/benchmarking-wissen/benchmarking-grundlagen/ [visited on 08.09.2015].

[16] IEEE Std 1061-1998. IEEE Standard for a Software Quality Metrics Methodology. 1998. ISBN: 1-55937-529-9.

[17] Richard Seidl and Harry Sneed. Modeling Metrics for UML Diagrams. The Magazine for Professional Testers, 12–20. September 2010. Accessible through: http://www.testingexperience.com/ [visited on 08.09.2015].

[18] Harry Sneed. Software Design Measurement. http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_informatik/smt/st/studium?subject=267&lang=en&leaf=1&head=2 [visited on 08.08.2015].

[19] Michael J. Cahill, Uwe Röhm and Alan D. Fekete. Serializable Isolation for Snapshot Databases. 2008. Downloadable from: http://dl.acm.org/citation.cfm?id=1376690 [visited on 08.09.2015].

[20] H-Store Project. Stonebraker Electronic Airline Ticketing System. http://hstore.cs.brown.edu/projects/seats/ [visited on 08.09.2015].

[21] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/ [visited on 08.09.2015].

[22] Transaction Processing Performance Council. TPC-H. http://www.tpc.org/tpch/default.asp [visited on 08.09.2015].

[23] Transaction Processing Performance Council. TPC BENCHMARK™ C Standard Specification Revision 5.11. February 2010. Downloadable from: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf [visited on 08.09.2015].

[24] Transaction Processing Performance Council. TPC BENCHMARK™ H Standard Specification Revision 2.17.1. November 2014. Downloadable from: http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf [visited on 08.09.2015].

[25] Visawee Angkanawaraphan. AuctionMark.http://hstore.cs.brown.edu/projects/auctionmark/ [visited on 08.09.2015].

[26] Visawee Angkanawaraphan. AuctionMark OLTP Benchmark Project Report. February 2011. Downloadable from: http://cs.brown.edu/research/pubs/theses/masters/2011/visawee.pdf [visited on 08.09.2015].

[27] H-Store. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/ [visited on 08.09.2015].

[28] Carlo Curino, Hyun Jin Moon, Alin Deutsch and Carlo Zaniolo. Automating the database schema evolution process. February 2013. Downloadable from: http://dl.acm.org/citation.cfm?id=2431356 [visited on 08.09.2015].

[29] Eclipse. https://www.eclipse.org/home/index.php [visited on 08.09.2015].

[30] Java. https://www.java.com/de/ [visited on 08.09.2015].

[31] Eclipse Modeling Project. Eclipse Modelling Framework. https://eclipse.org/modeling/emf/ [visited on 08.09.2015].

[32] Eclipse Modeling Project. MoDisco. https://eclipse.org/MoDisco/ [visited on 08.09.2015].

[33] Eclipse Modeling Project. Xtext. http://www.eclipse.org/Xtext/ [visited on 08.09.2015].

[34] itemis. Xtend. https://eclipse.org/xtend/index.html [visited on 08.09.2015].

[35] Thosmas Kühn. CROM Meta Model. https://github.com/Eden-06/CROM [visited on 08.09.2015].

[36] ISO/IEC9126-1. Software engineering – Product quality – Part 1: Quality Model. 2001.

[37] ISO/IEC25010. Software Product Quality Requiremens and Evaluation(SQuaRE). 2007.

[38] Ying Zou and Kostas Kontogiannis. Incremental Transformation of Procedural Systems to Object Oriented Platforms. November 2003. Downloadable from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1245356 [visited on 08.09.2015].

[39] Ying Zou and Kostas Kontogiannis. Quality Driven Transformation Compositions for Object Oriented Migration. 2002. Downloadable from: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1183004 [visited on 08.09.2015].

[40] Rosâkngela Penteado, Paulo C. Masiero, Antonio F. do Prado and Rosana T. V. Braga. Reengineering of Legacy Systems Based on Transformation Using the Object Oriented Paradigm. 1998. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=723184 [visited on 08.09.2015].

[41] Philip Newcomb. Reengineering Procedural Into Object-Oriented Systems. 1995. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=514712 [visited on 08.09.2015].

[42] Eunjoo Lee, Byungjeong Lee, Woochang Shin, and Chisu Wu. A Reengineering Process for Migrating from an Object-oriented Legacy System to a Component-based System. November 2003. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1245362 [visited on 08.09.2015].

[43] Chahat Monga, Ms. Aman Jatain and Dr. Deepti Gaur. MIGRATION STRATEGIES FOR OBJECT ORIENTED SYSTEM TO COMPONENT BASED SYSTEM. Downloadable from: http://airccse.org/journal/ijfcst/papers/4614ijfcst08.pdf[visited on 08.09.2015].

[44] Ladan Tahvildari and Kostas Kontogiannis. A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering. 2002. Downloadable from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1167824 [visited on 08.09.2015].

[45] Friedrich Steimann and Fabian Urs Stolz. Refactoring to Role Objects. May 2011. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6032483 [visited on 08.09.2015].

[46] Munawar Hafiz and Ralph E. Johnson. Improving Perimeter Security with Security-oriented Program Transformations. May 2009. Downloadable from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5068460 [visited on 08.09.2015].

[47] Andy Oppel. SQL Demystified, Page 7. 2005. Buy-able under: ISBN 0-07-226224-9.

[48] Kim B. Bruce. 2.1 Objects, classes, and object types. 2002. Downloadable from: //books.google.com/books?id=9NGWq3K1RwUC&pg=PA18 [visited on 11.09.2015].

[49] Development of an object-oriented DBMS, Pages 472 – 482. 1986. Buy-able under: ISBN 0-89791-204-7.

Appendix

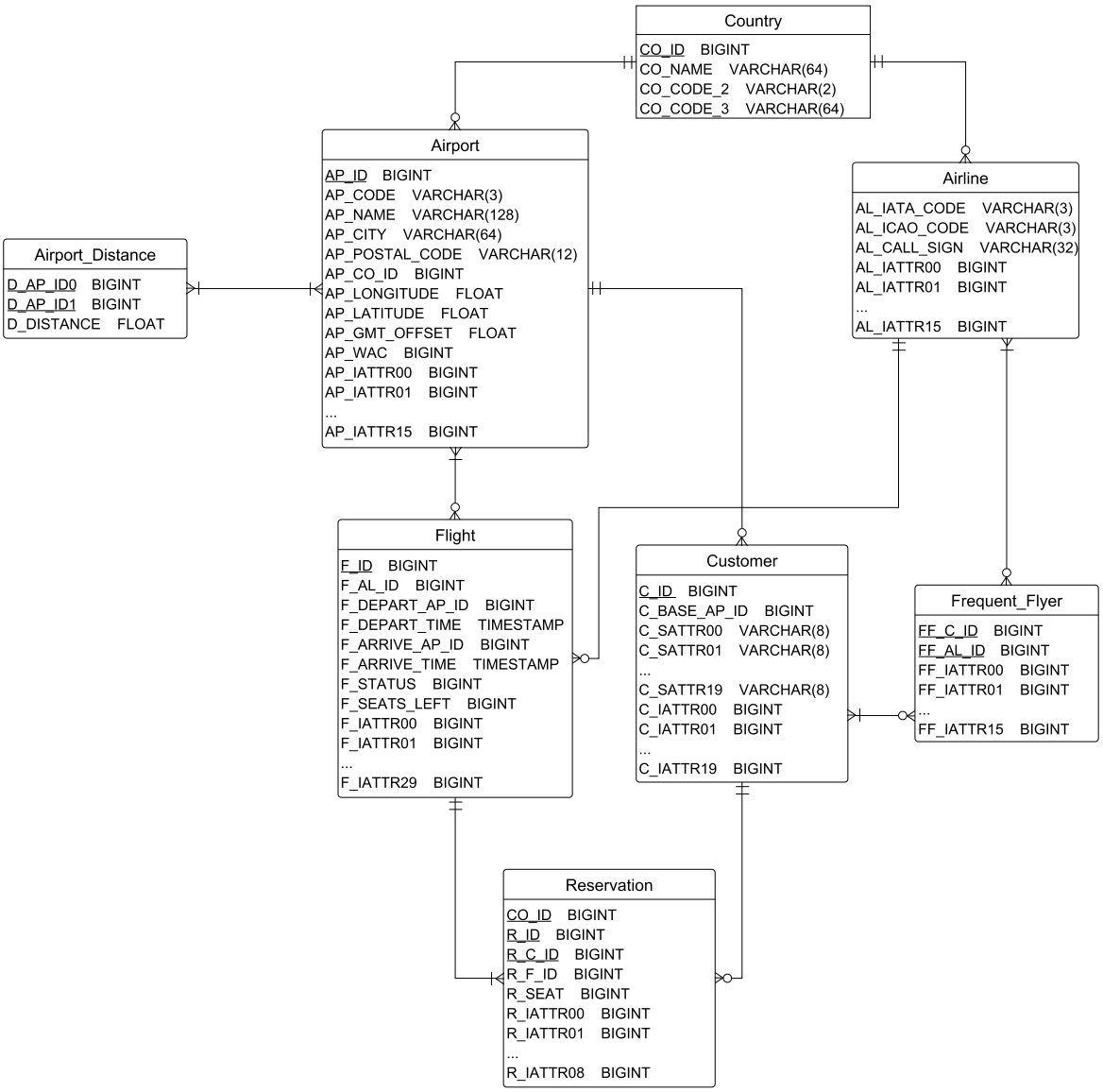

Figure 1: SQL Schema of SEATS

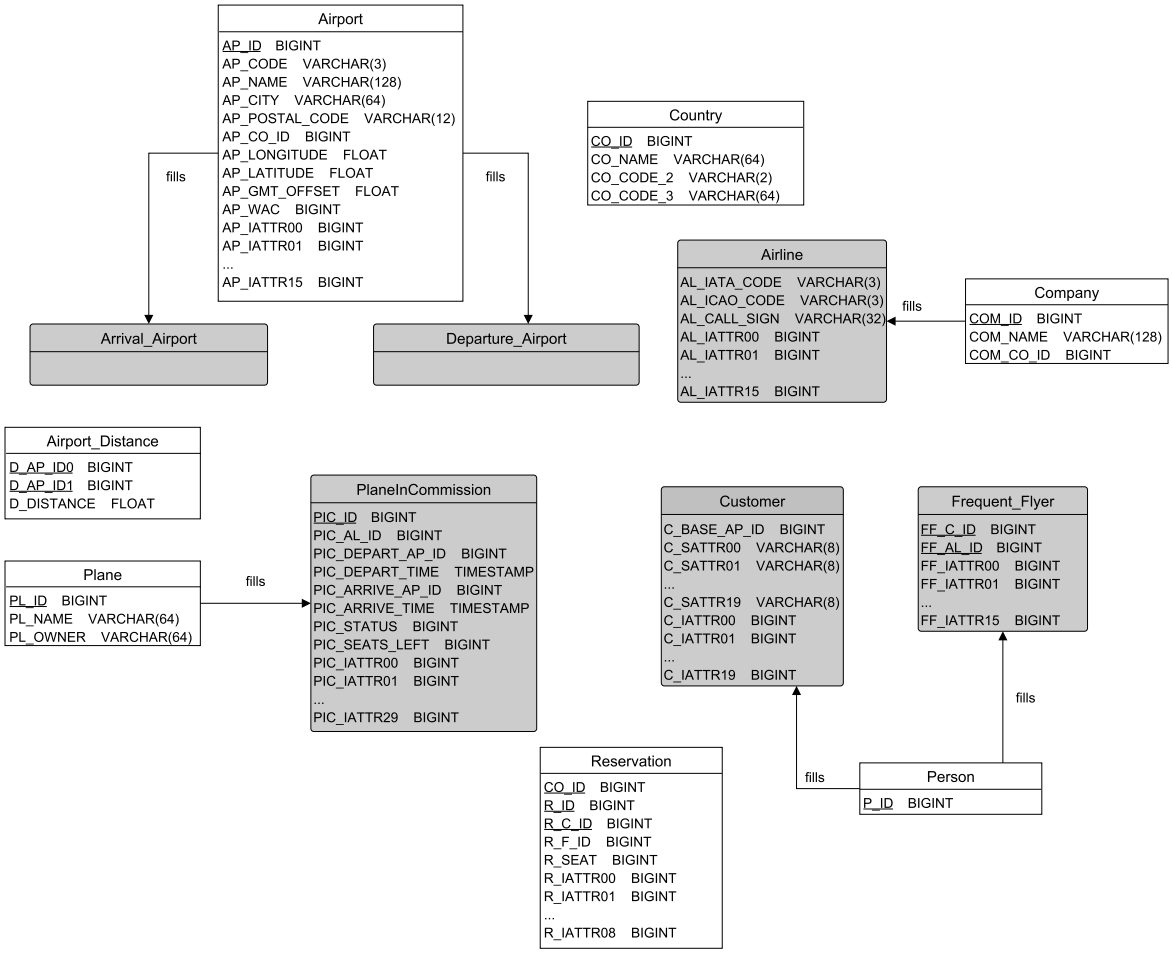

Figure 2: NTRT Schema of SEATS

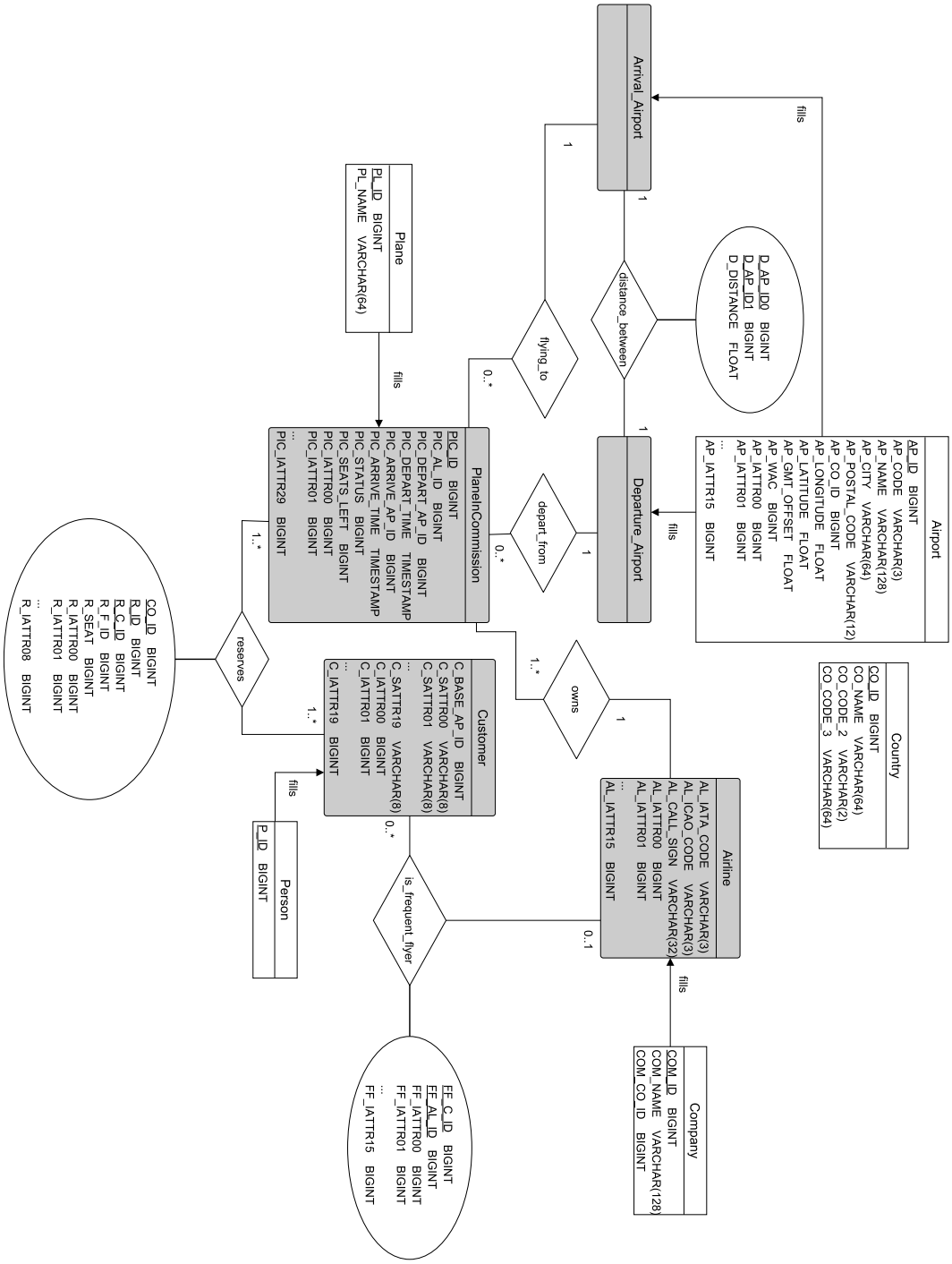

Figure 3: RSQL Schema of SEATS

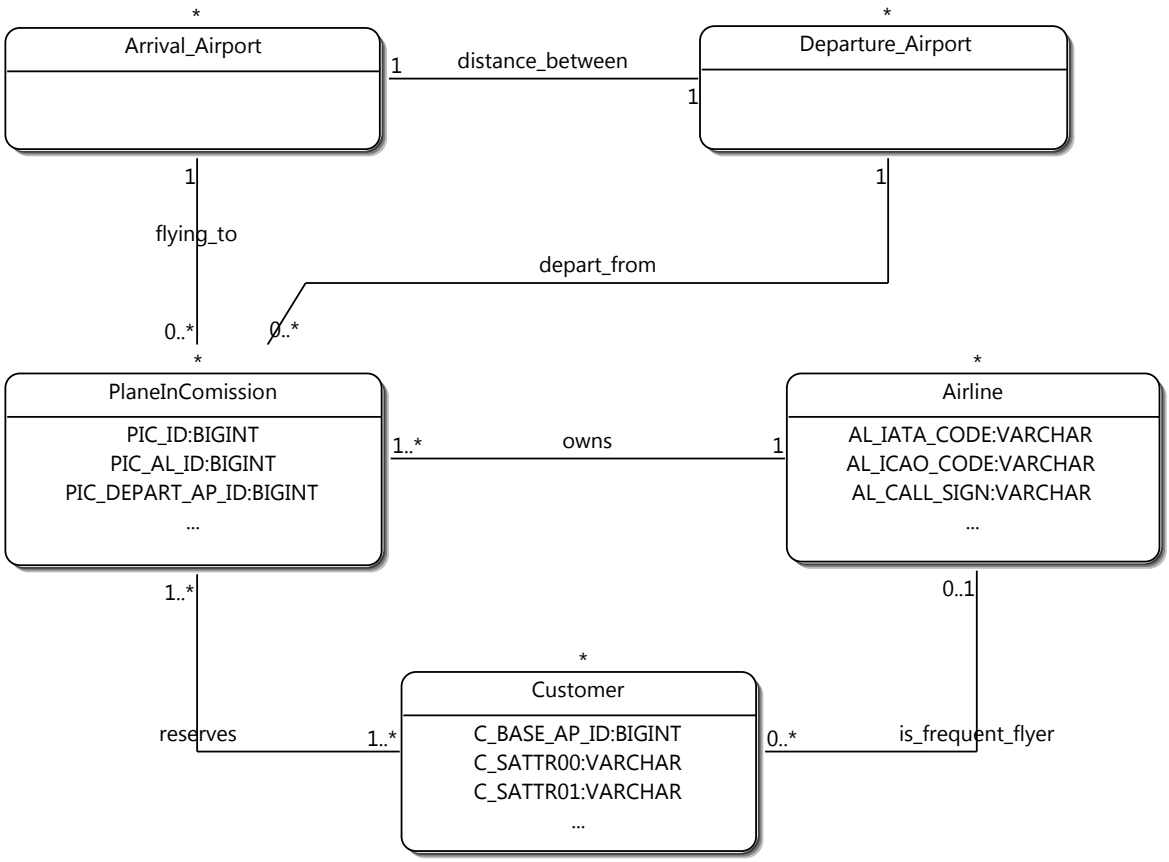

Figure 4: Content of the Compartment Type Flight(SEATS)

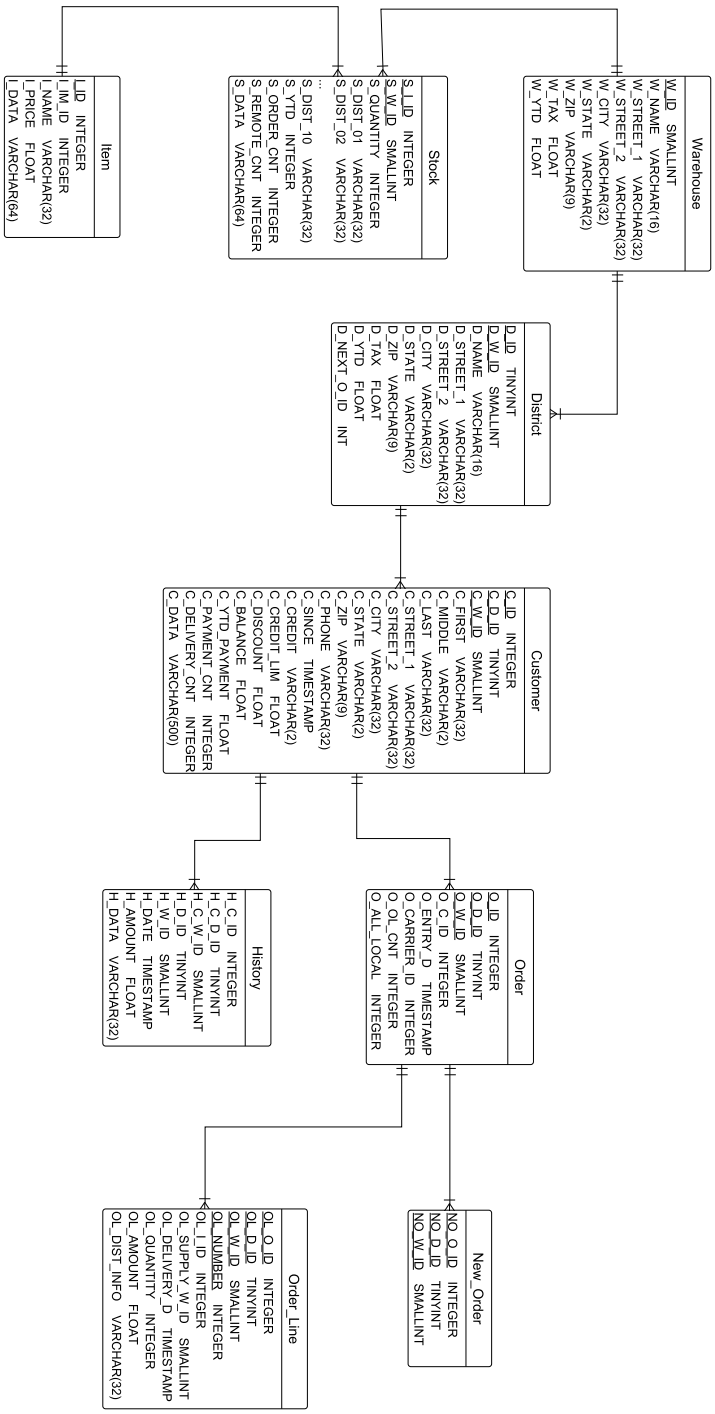

Figure 5: SQL Schema of TPC-C

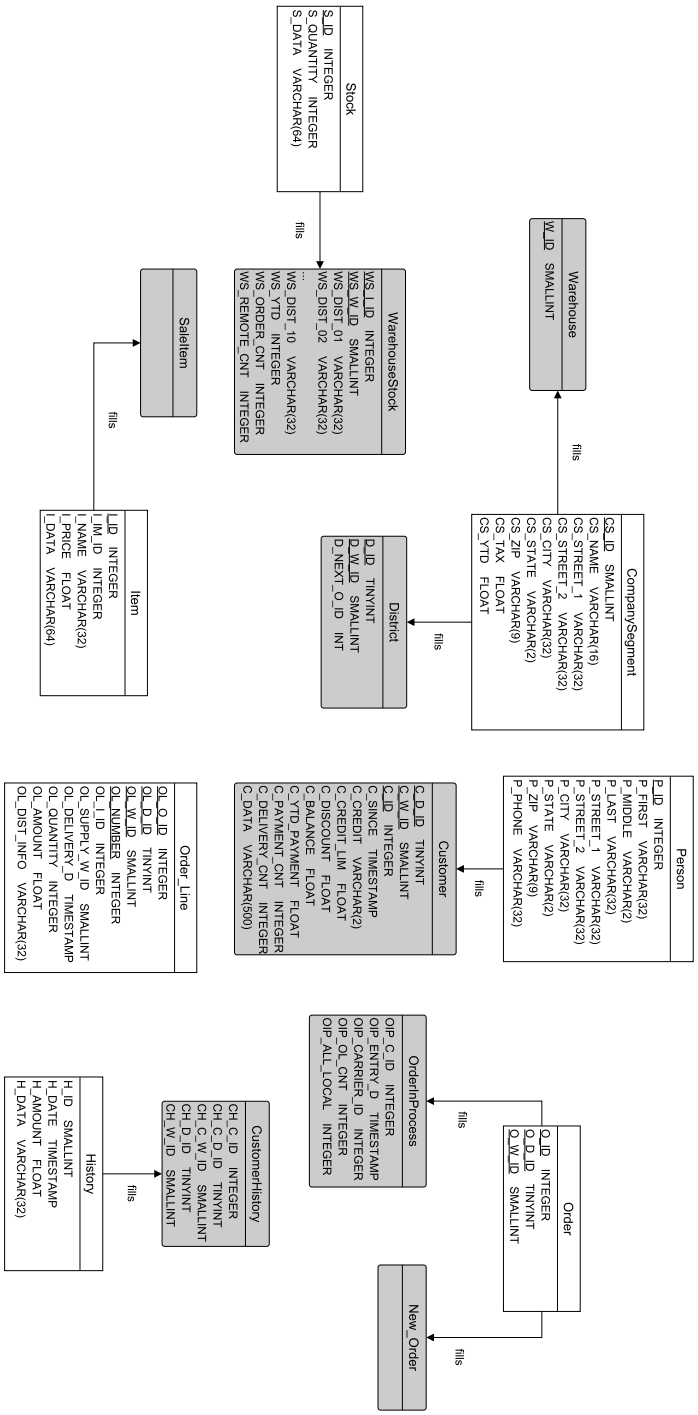

Figure 6: NTRT Schema of TPC-C

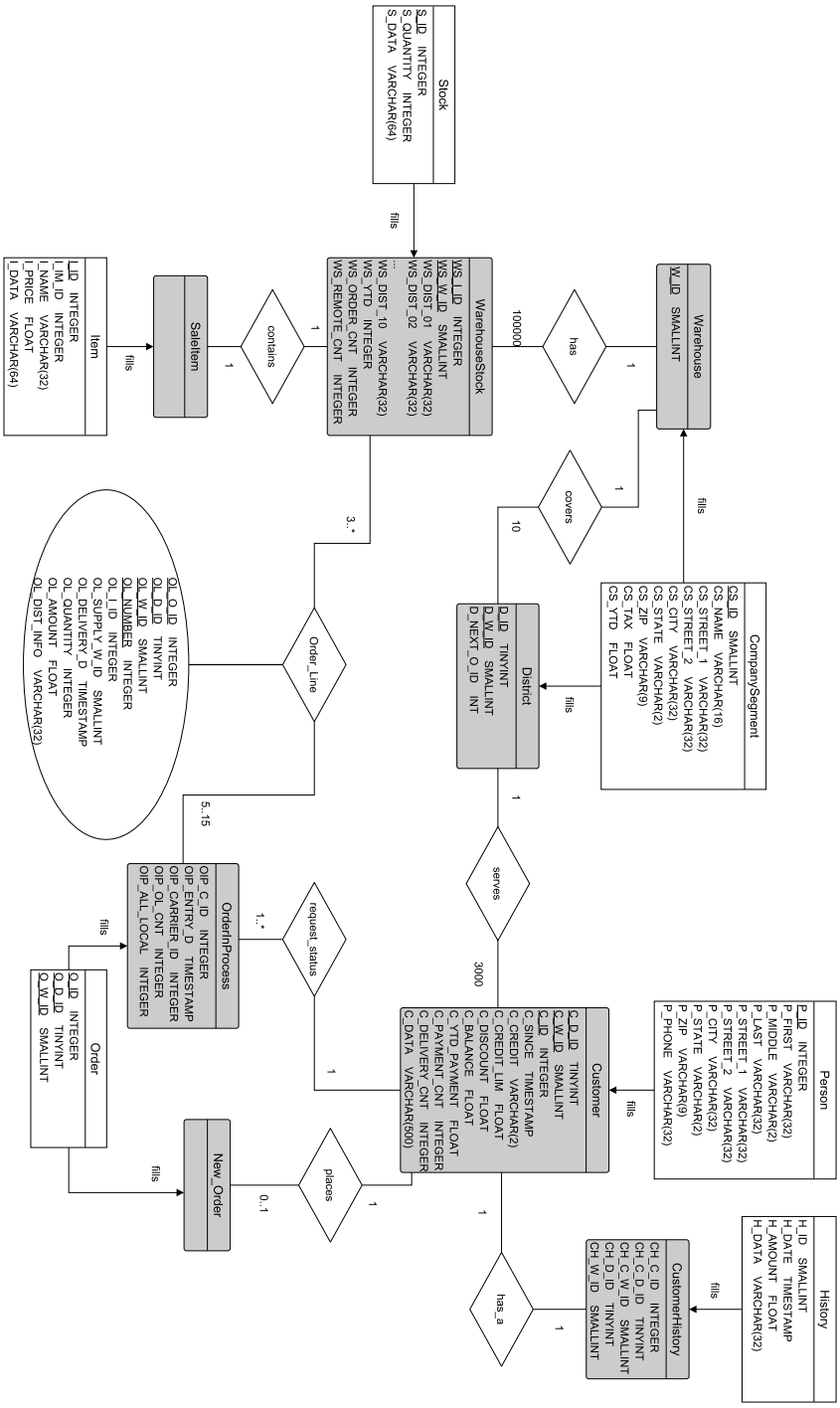

Figure 7: RSQL Schema of TPC-C

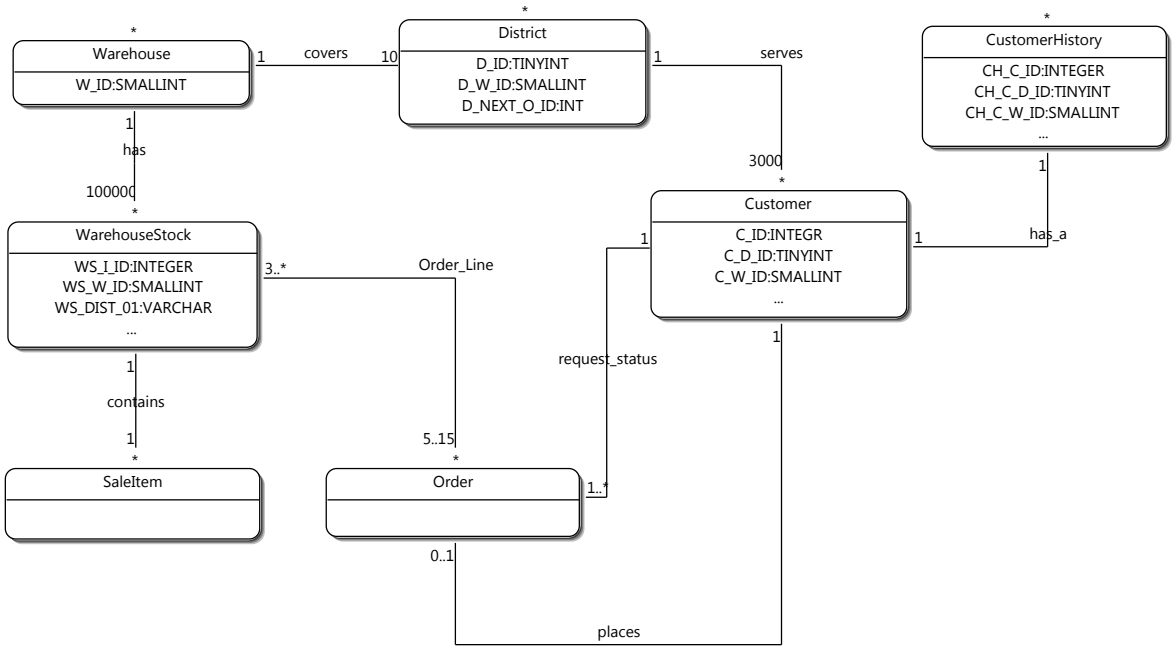

Figure 8: Content of the Compartment Type Company(TPC-C)

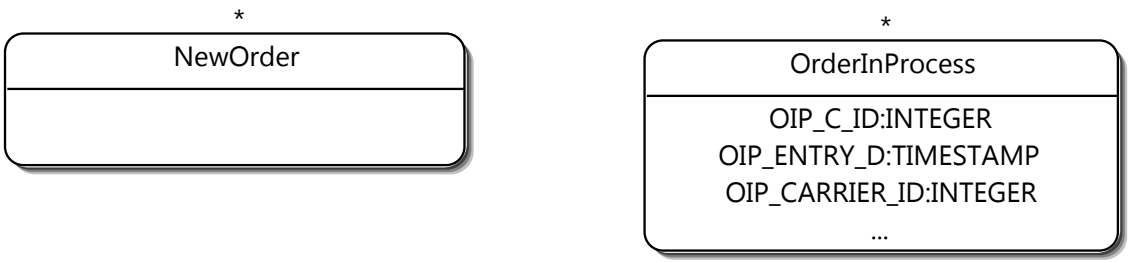

Figure 9: Content of the Compartment Type Order(TPC-C)

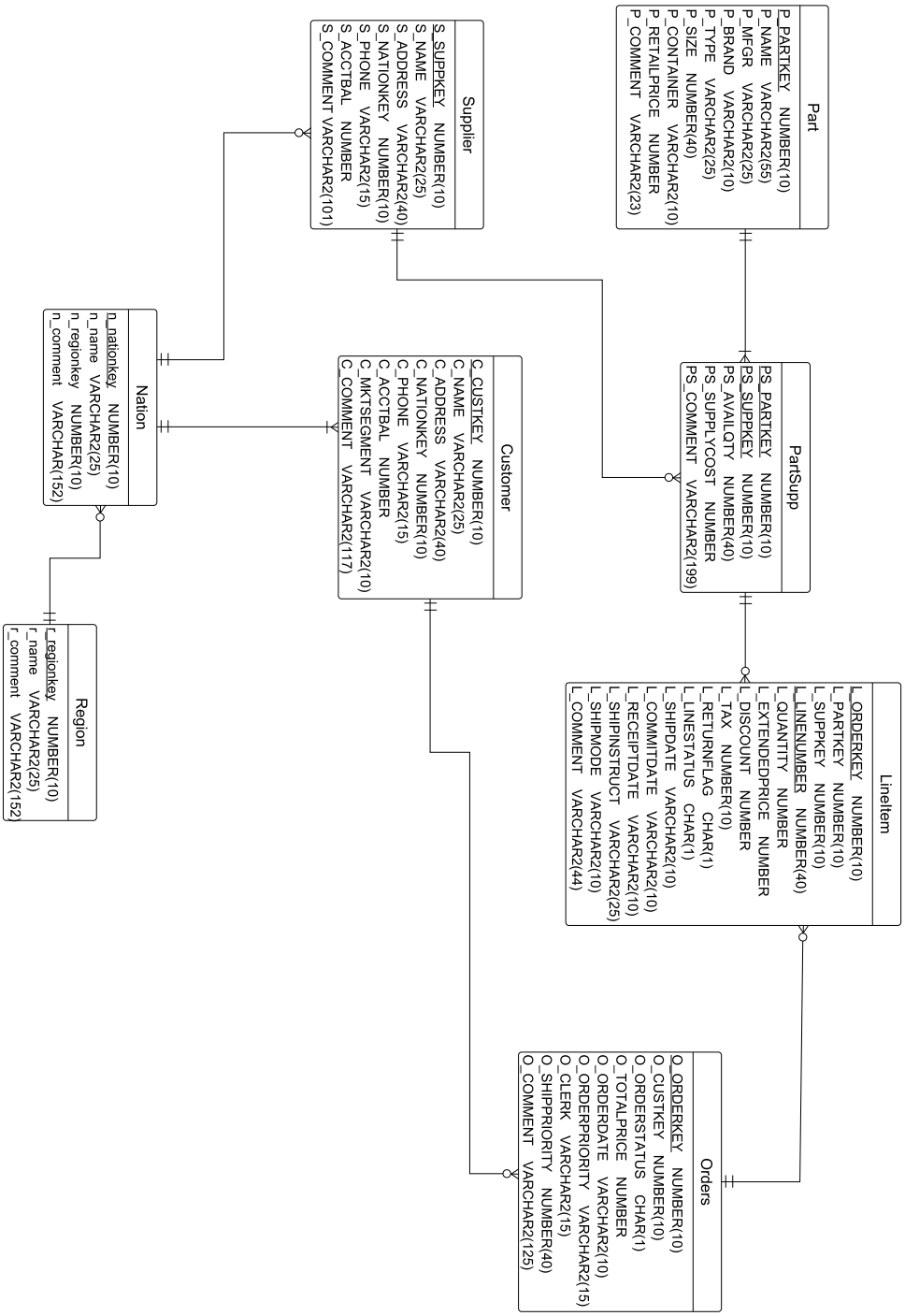

Figure 10: SQL Schema of TPC-H

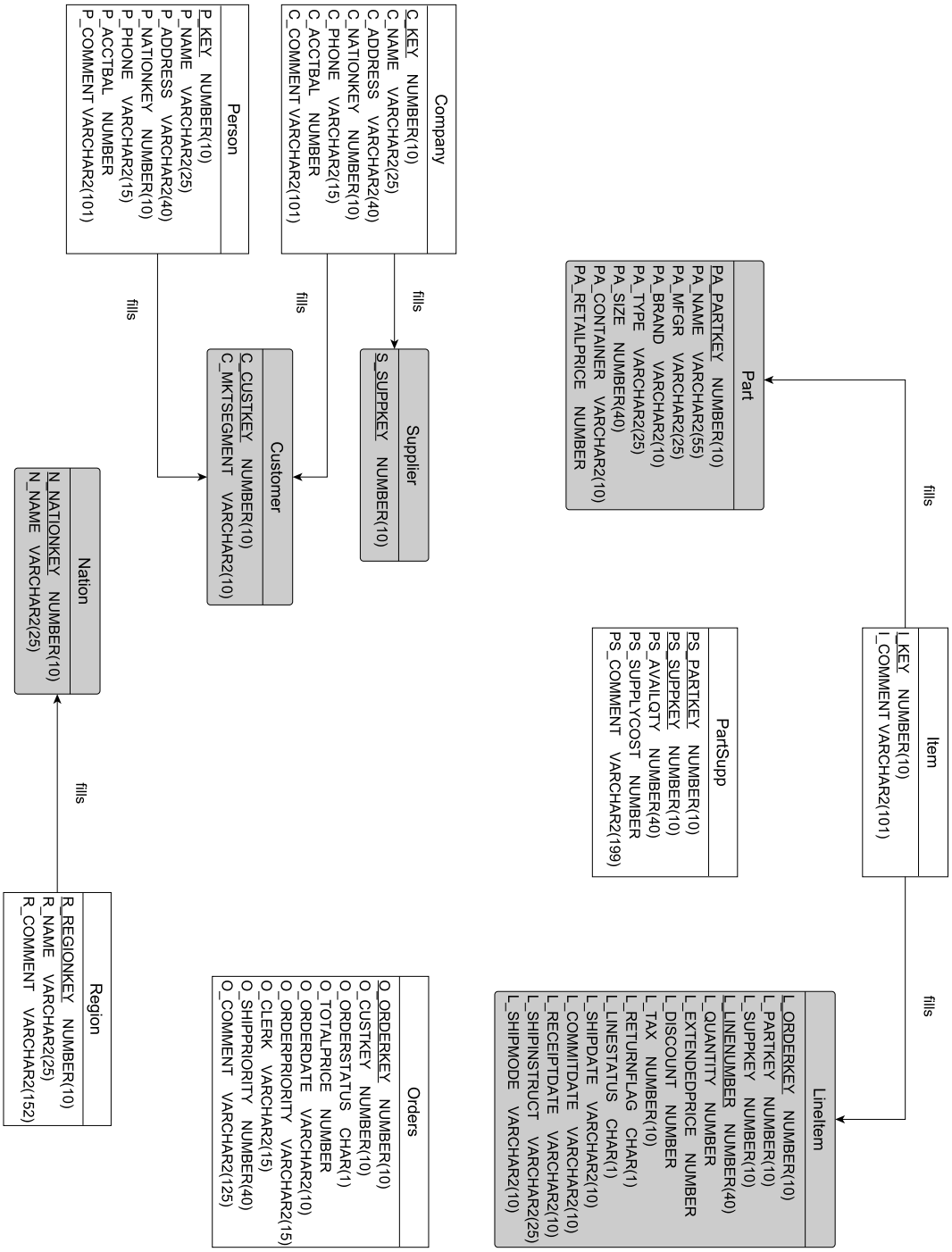

Figure 11: NTRT Schema of TPC-H

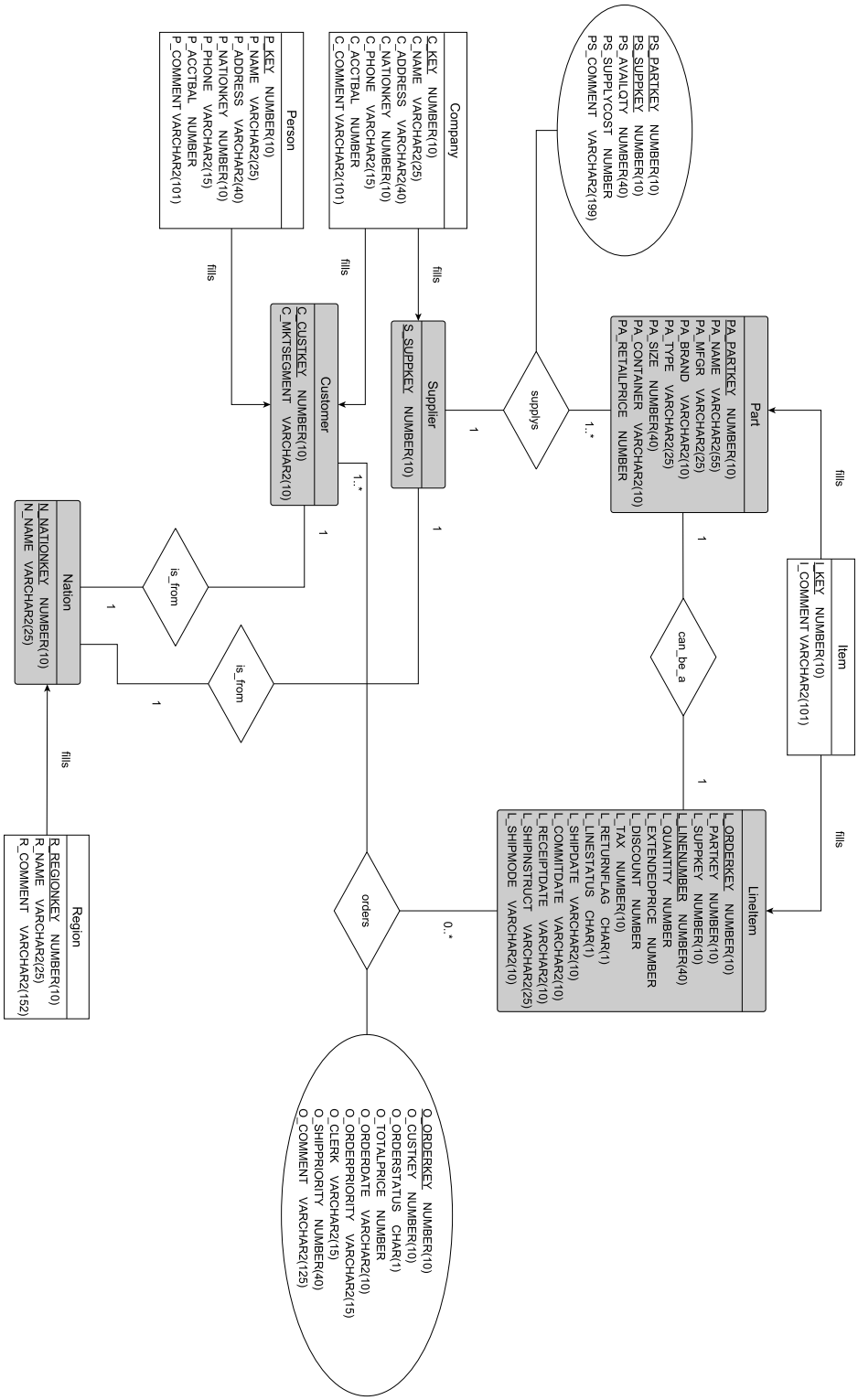

Figure 12: RSQL Schema of TPC-H

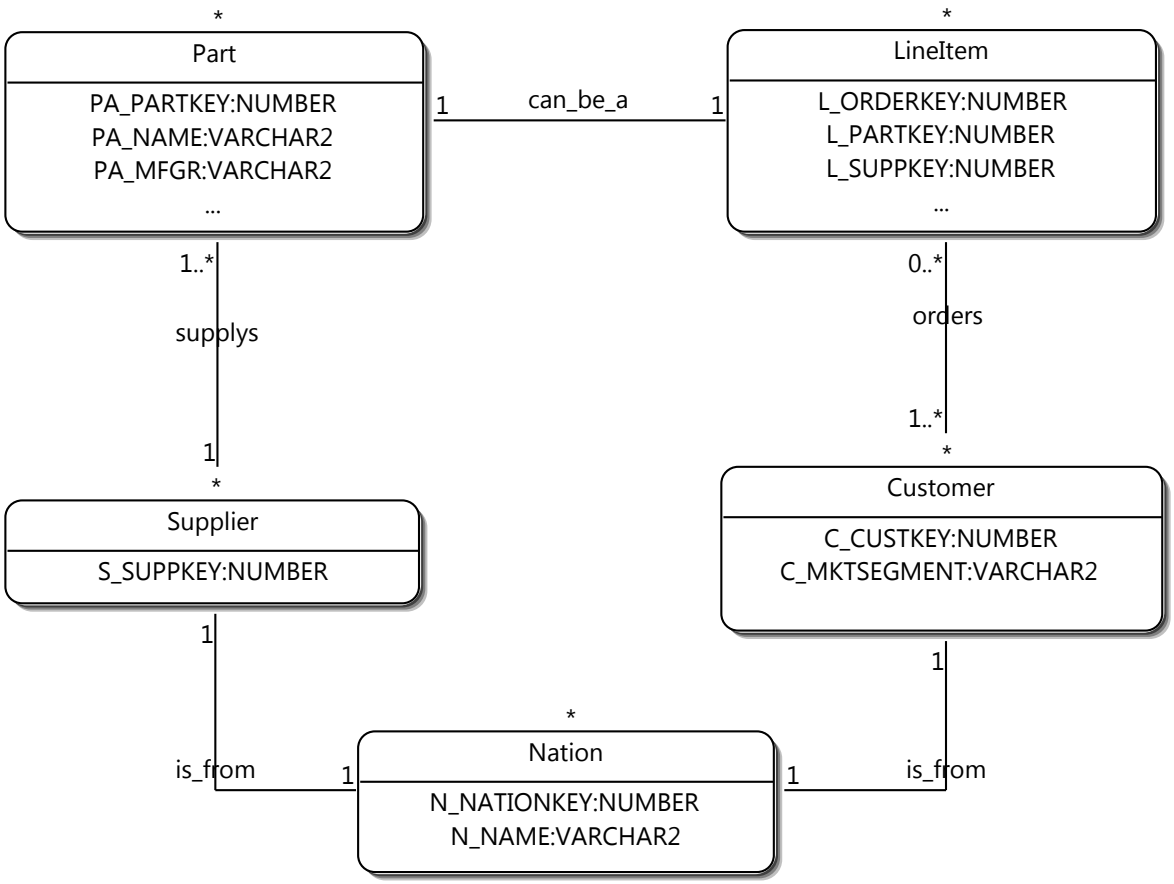

Figure 13: Content of the Compartment Type Business(TPC-H)

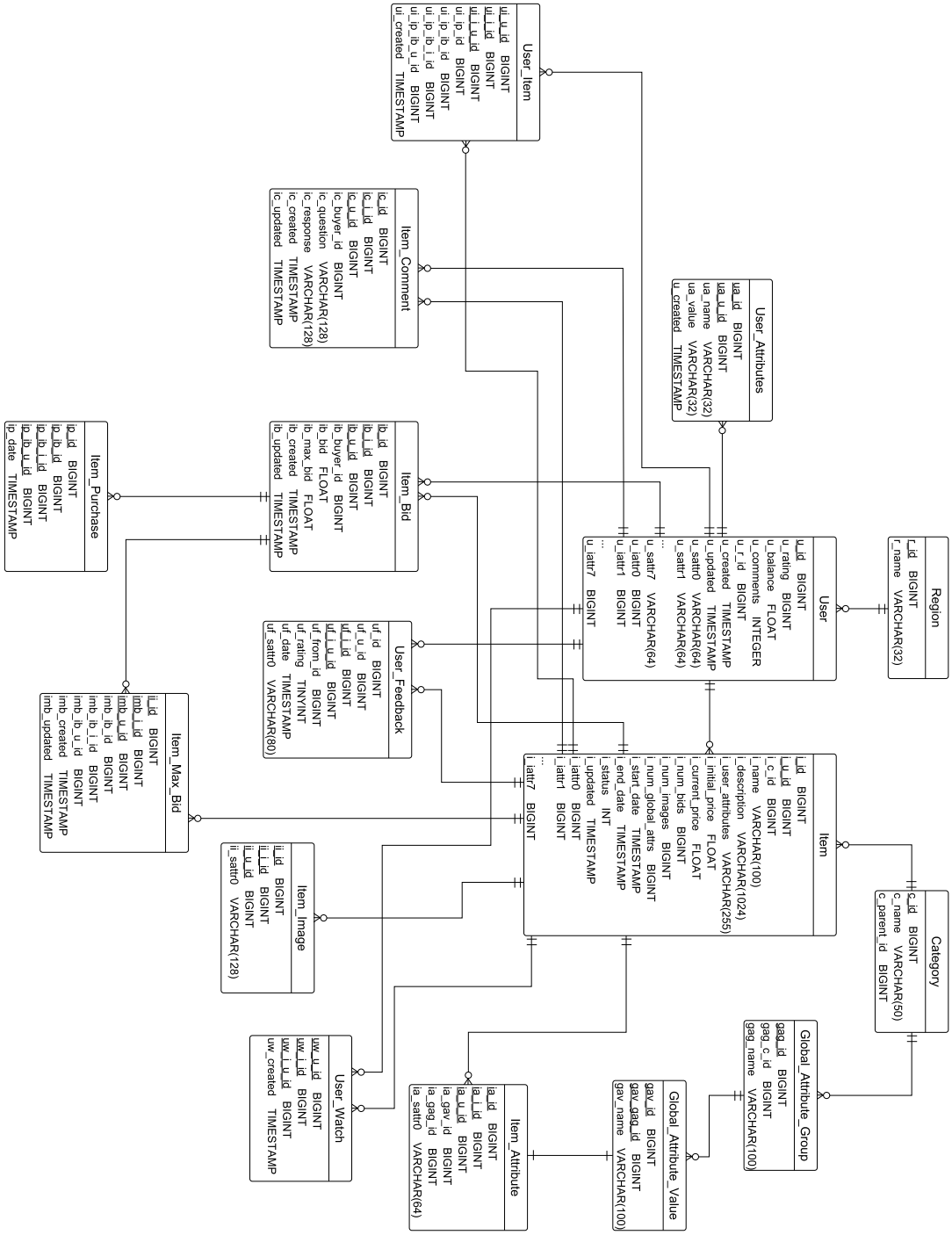

Figure 14: SQL Schema of AuctionMark

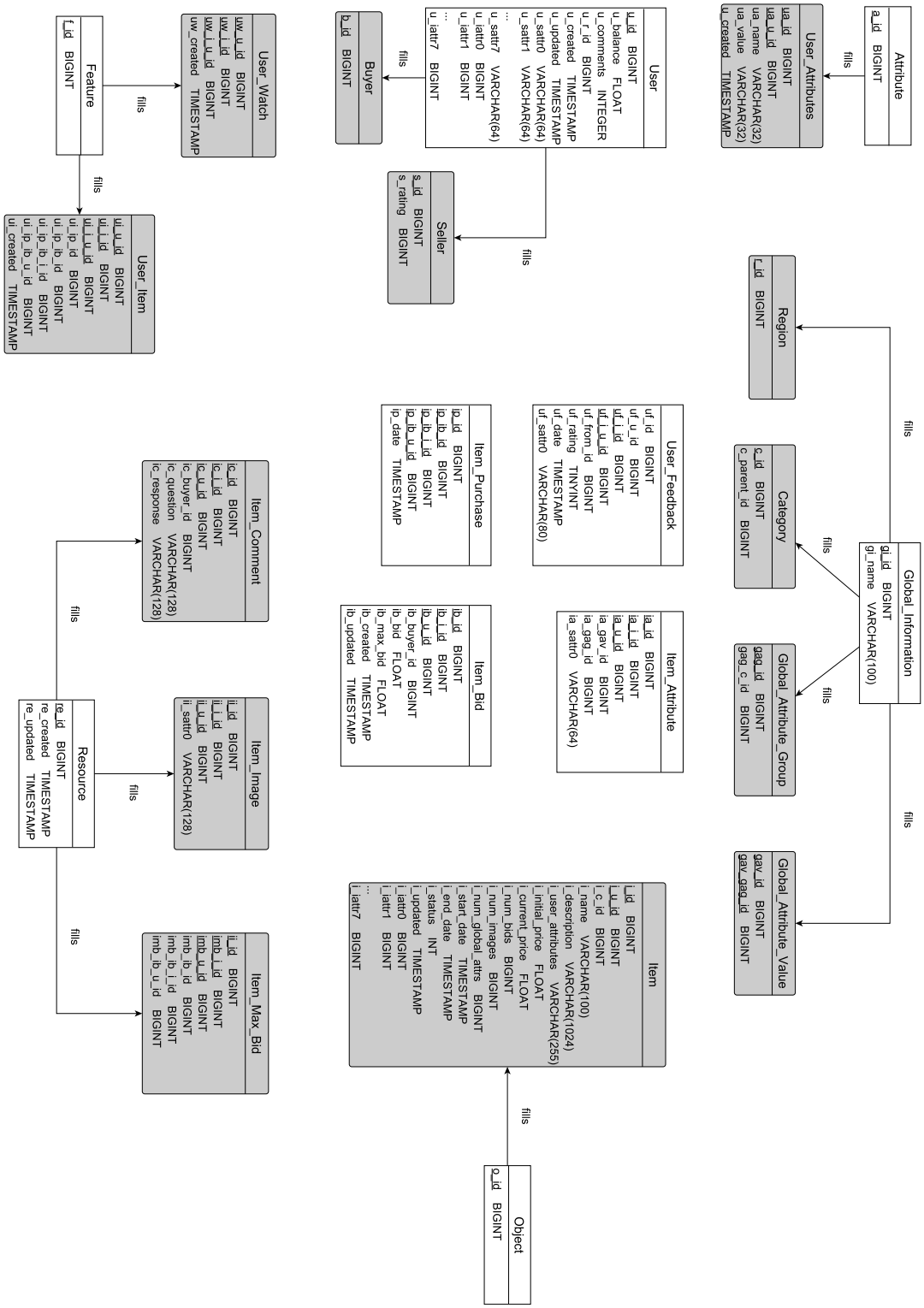

Figure 15: NTRT Schema of AuctionMark

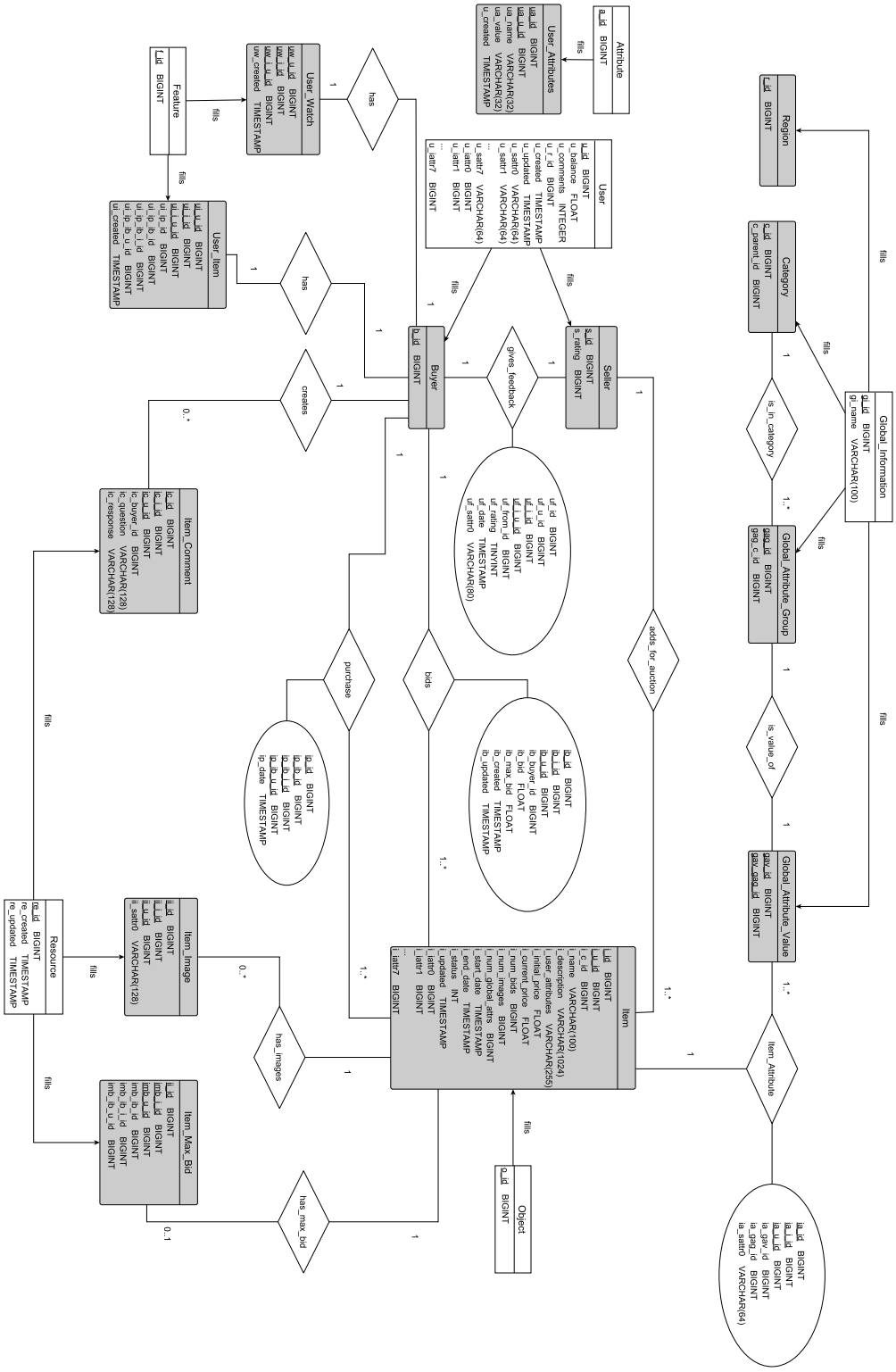

Figure 16: RSQL Schema of AuctionMark

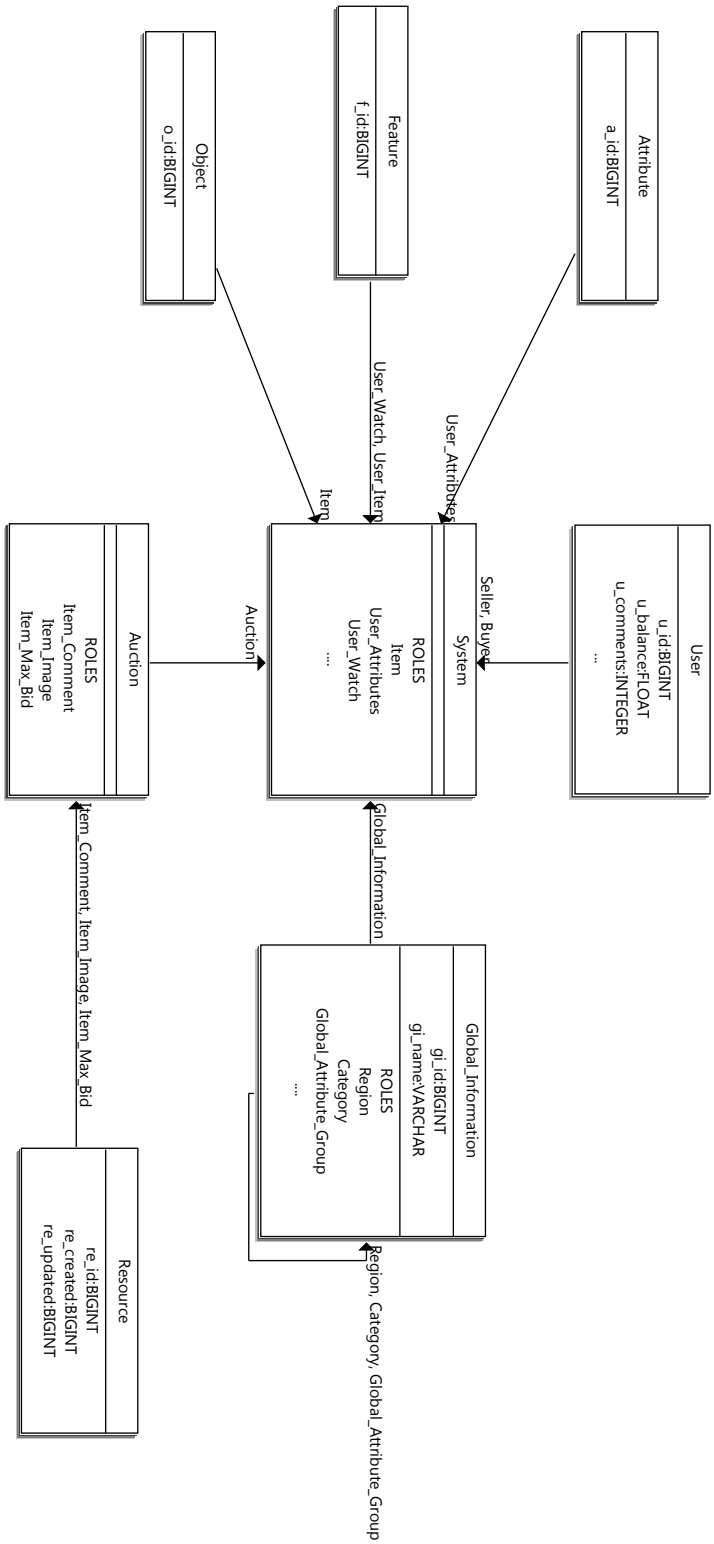

Figure 17: CROM Basic Schema of AuctionMark

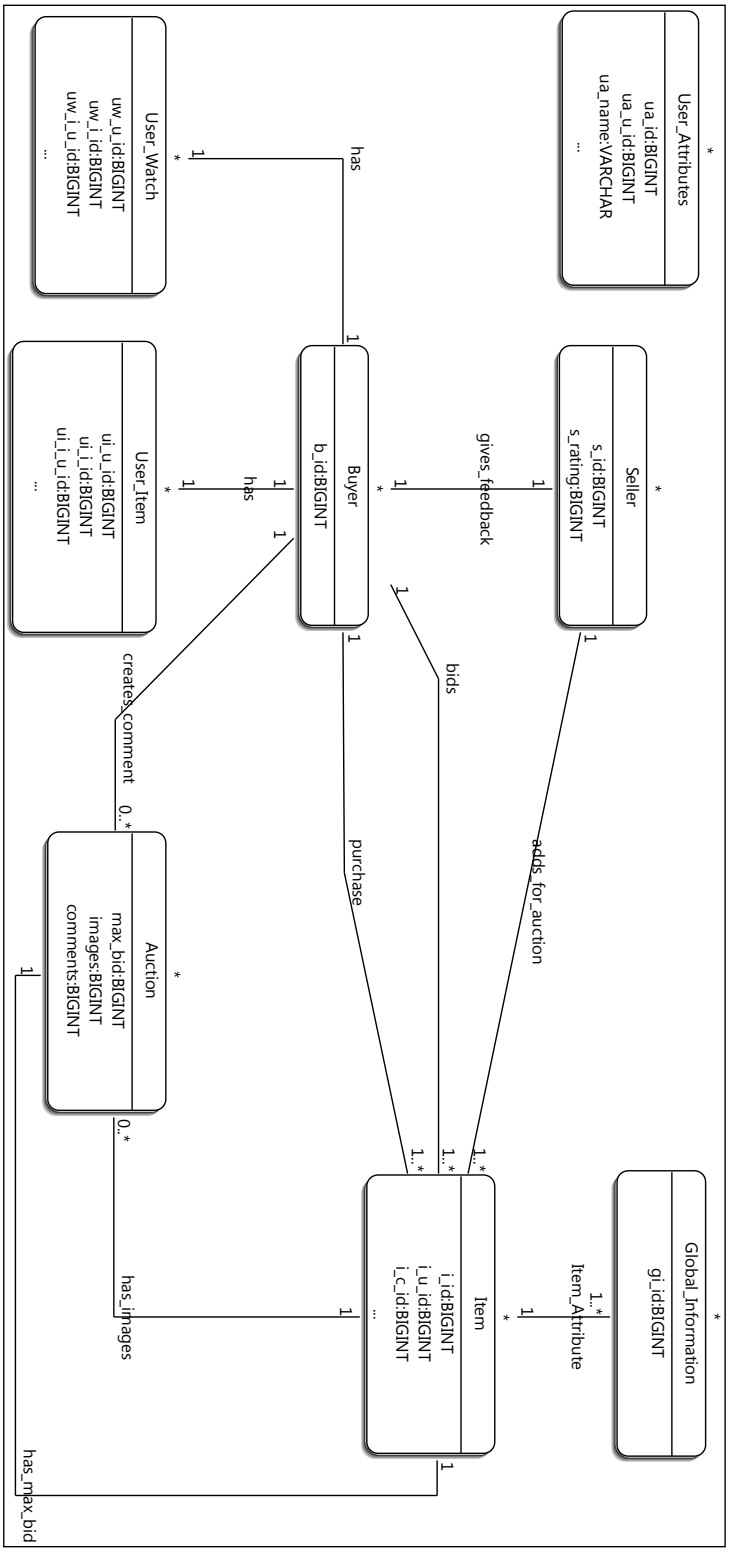

Figure 18: Content of the Compartment Type System(AuctionMark)

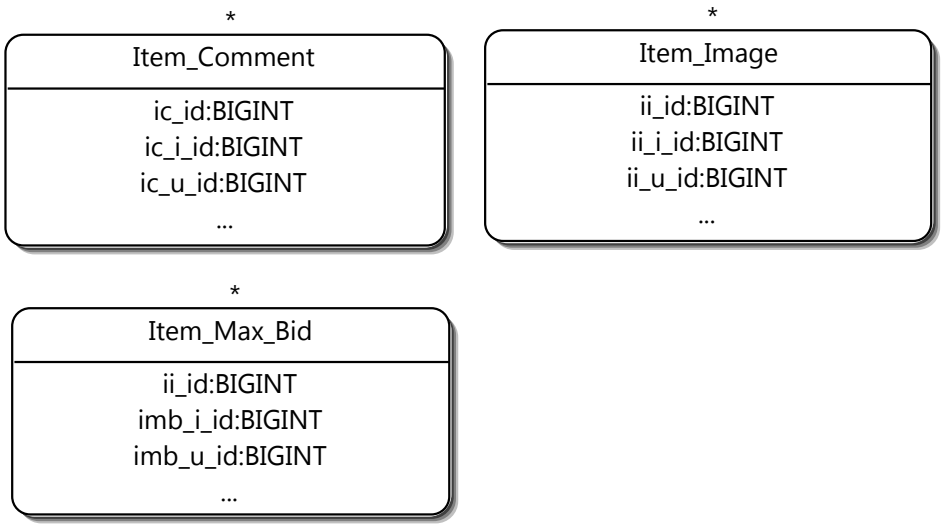

Figure 19: Content of the Compartment Type Auction(AuctionMark)

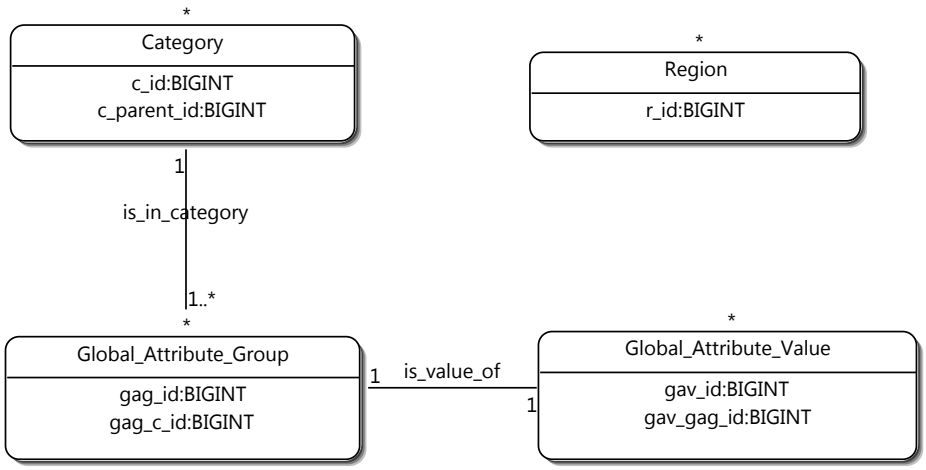

Figure 20: Content of the Compartment Type Global Information(AuctionMark)