GENERATING A TEXTUAL REPRESENTATION OF A RELATIONAL MODEL

HS-IDA-MD-01-011

Johan Torin

Submitted by Johan Torin to the University of Skövde as a dissertation towards the degree of M.Sc by examination and dissertation in the Department of Computer Science.

October 2001

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

__________________________
Johan Torin
To Anna
Abstract

To aid the software and database developer in the development process, specialised software is needed, known as CASE-tools. To form a productive work environment, multiple CASE-tools should be able to cooperate with each other, as it is likely that a single tool cannot give full support for the whole development process. An aid in the integration of tools is that information is stored in a central repository, available for any tool that needs it. A problem which arises is that not all tools are capable of directly accessing the contents of the repository. Thus export procedures are needed that transforms the repository contents into a format that an external tool can read and understand. If these transformation procedures are specified directly in the implementation language, modification can be difficult or even impossible.

This work proposes a general transformation model for storing rules in a repository. These rules operate on the information in a source model and transform it to a textual representation ready for export to external tools. An example ruleset is given that transforms an example relational model into SQL code.
Contents

1 Introduction ............................................. 1

2 Background ............................................. 3
   2.1 Database development ............................... 3
   2.2 Tool support for software engineering - CASE .......... 6
   2.3 Repositories ....................................... 9
       2.3.1 Metamodels .................................. 13
   2.4 Standards ......................................... 14
       2.4.1 UML - Unified Modeling Language .............. 15
       2.4.2 OCL - Object Constraint Language ............. 18
       2.4.3 CORBA ....................................... 21
       2.4.4 MOF - Meta-Object Facility .................... 21
   2.5 Transformations .................................... 24
       2.5.1 Metacase .................................... 25
       2.5.2 Models ....................................... 26
       2.5.3 Code ......................................... 35

3 Problem Description ................................. 38
   3.1 Motivation ......................................... 38
   3.2 Objectives ........................................ 39
3.3 Previous work .......................................................... 38
  3.3.1 Extensions to OCL .............................................. 40
  3.3.2 Extensions to the Meta-model ............................... 40
  3.3.3 Objectives restated ........................................... 41

4 Implementation ........................................................... 43
  4.1 Definition of concepts ........................................... 43
  4.2 An example model ................................................ 43
  4.3 A general transformation model ................................ 51
    4.3.1 Resulting text ............................................. 51
    4.3.2 Update approach .......................................... 53
    4.3.3 Events .................................................... 56
    4.3.4 Transformation model .................................... 59
    4.3.5 Example transformations .................................. 62
  4.4 Using OCL for transformations .................................. 68
    4.4.1 Extensions of OCL ......................................... 68
    4.4.2 Ordering .................................................. 71
  4.5 Building SQL using ALTER TABLE .............................. 72
    4.5.1 Description of source model rules ....................... 74
    4.5.2 Description of textblock rules .......................... 76
    4.5.3 Example rule execution .................................. 80
    4.5.4 Validation of end result ................................ 80
  4.6 Building SQL using less ALTER TABLE ........................ 81
    4.6.1 Description of source model rules ....................... 82
    4.6.2 Description of textblock rules .......................... 83

5 Analysis ................................................................. 89
  5.1 Hierarchical structure .......................................... 89
5.2 Discussion about transformation model ........................................... 90
  5.2.1 Propagating events ............................................................... 90
  5.2.2 Cardinality of the role <Class>ProvidesInfoFor .......................... 93
  5.2.3 Includes/IsIncludedBy ....................................................... 94
  5.2.4 Capabilities of the transformation model ................................. 94
5.3 The transformation rulesets ....................................................... 95

6 Conclusions ............................................................................. 96
  6.1 Summary ............................................................................... 96
  6.2 Future Work ....................................................................... 97

A Execution example of a insert .................................................. 103

B Execution example of an update ............................................. 105

C Execution example of a delete ................................................ 108

D Generation of SQL from a relational model utilising ALTER TABLE .... 110
  D.1 A relational to SQL transformation ruleset utilising ALTER TABLE ... 110
    D.1.1 Rules for handling source model objects ............................... 110
    D.1.2 Rules for handling textblock objects .................................. 114
  D.2 Executionlog of a relational model transformed to SQL .................. 119
    D.2.1 A relation with three attributes are inserted ......................... 119
    D.2.2 A primary key is added .................................................... 128
    D.2.3 A relation with a foreign key is added ................................. 139

E Output from PostgreSQL interactive terminal ............................ 154

F A more intelligent ruleset for generating SQL ............................ 156
  F.1 A relational to SQL transformation ruleset ............................... 156
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.1.1</td>
<td>Rules for handling source model objects</td>
<td>156</td>
</tr>
<tr>
<td>F.1.2</td>
<td>Rules for handling source model objects</td>
<td>161</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Phases of database design, adapted from Elmasri and Navathe (1994) . . . 5
2.2 An example of two models ........................................... 13
2.3 A simplified metamodel of the UML classmodel .................... 14
2.4 Layers of a repository .................................................. 15
2.5 An example of a classdiagram ......................................... 19
2.6 An example of an UML constraint .................................... 32
2.7 An example using OCL for specifying an UML constraint ........... 32
2.8 The four meta-data layers of the MOF ............................... 32
2.9 A relation with one key ............................................... 33
2.10 A relation with one foreign key ..................................... 33
2.11 A relation there a foreign key is part of the primary key .......... 33
2.12 A SQL statement for creating the table Record ..................... 36
2.13 A SQL statement for creating the table Sale ....................... 36
2.14 A SQL statement for creating the table Track ..................... 36
2.15 A SQL statement that adds a new column and constraint to table Track and then removes them ........................................... 37

3.16 Transformation meta-model framework .............................. 41
4.17 A meta-model of the relational model .............................. 45
4.18 A single relation/table ................................................. 46
4.19 A single relation with a primary key and additional attributes .......... 46
4.20 A relation (relationB) with a foreign key that references another relation (relationA) .......................................................... 47
4.21 Two relations, each with a primary key, which references each other through foreign keys ......................................................... 47
4.22 SQL-code that creates a single table ............................................ 47
4.23 SQL-code that creates a single table with a primary key and additional attributes ................................................................. 48
4.24 SQL-code that creates a relation (relationA) with a foreign key that references another relation (relationB) ................................. 48
4.25 SQL-code that creates two relations, each with a primary key, which references each other through foreign keys .......................... 49
4.26 A variant of figure 4.25 in which the SQL-code is generated in steps ... 50
4.27 How the resulting text is built up .................................................. 51
4.28 The Textblock class ..................................................................... 53
4.29 A general transformation model .................................................... 59
4.30 An example of how the general transformation model works; an object is inserted into the Relation class ....................................... 65
4.31 Objects which references other objects ........................................... 71
4.32 An UML object diagram that describes the final state of the example execution in appendix D ....................................................... 73
4.33 The final result from the transformation ......................................... 81
4.34 Order of generated tables ............................................................. 86
5.35 Information can be extracted at different level of the hierarchical model . 90
5.36 Two source model classes and a textblock class, rules builds textblock directly 91
5.37 Two source model classes and a textblock class, changes propagates .... 92
5.38 An excerpt from the general transformation model showing the textblock class 93
List of Tables

2.1 Layers of the UML architecture ........................................ 17
2.2 Predefined diagrams of UML. ............................................ 18
2.3 An example of a database relation/table populated with data ........ 28
2.4 The sale relation populated with some data .......................... 30
2.5 The track relation populated with some data .......................... 34
Chapter 1

Introduction

As the size and complexity of database applications grows, the database designer and administrator finds her/himself facing a likewise increasingly difficult task of understanding and grasping the finished product. To help them many tools exist today. Unfortunately, they do not integrate very well to provide an uniform and transparent work environment. For example, there exist very advanced modelling tools for conceptually describing a database. Also, there are a lot of tools for manipulating databases, but intelligent cooperation between the two tool environments is missing.

This dissertation takes the view that a higher grade of cooperation can be achieved by placing all information in a central repository to which all tools involved in database and software development have access. One problem is the export of information from the repository. As not all tools might be capable of accessing the repository, information within it must be exported in a format that the tools understand, and the most common format is a textual representation.

The result of this dissertation is a general transformation model in which rules for transformation can be specified. As text can be seen as one dimensional, the rules build the result in a hierarchical way, each rule creating a small piece which is then aggregated into bigger blocks.
Chapter 1. Introduction

The transformation model should be capable of handling most source models. As a proof of concept a complete ruleset that transforms a relational model to a textual form in SQL-92 is presented together with an example event and execution log.

This work is organised as follows: in chapter 2 the background to previous work in the field of research is given, and the standards used in this work are described. In chapter 3 the problem is further specified, and in chapter 4 the proposed solution is given. In chapter 5 an analysis of the work is performed. Finally, in chapter 6 the results are summarised and discussed together with suggestions for future work.
Chapter 2

Background

In this chapter preliminaries of this work are introduced.

In 2.1 database development and its process are described, in section 2.2 an introduction to CASE-tools is given and repositories are discussed in section 2.3. Various standards used in the field are introduced in section 2.4.

2.1 Database development

The number of database applications is increasing for every day, as is their size and complexity. Database developers thus need to use increasingly complex tools and methods for their work. Connolly, Begg and Strachan (1998, page 227) give three different phases of database design:

- Conceptual – The process of constructing a model of the information used in an enterprise, independent of all physical considerations.

- Logical – The process of constructing a model of the information used in an enterprise based on a specific data model, but independent of a particular DMBS or other physical considerations.
Chapter 2. Background

- Physical – The process of producing a description of the implementation of the database on secondary storage; it describes the storage structures and access methods used to achieve efficient access to the data.

In each phase some type of model is used to store and represent the information. It is thus necessary to transform a model (or more correctly, the information it represents) used in one phase into a model used in another phase. This process is not only done in a top-down manner (conceptual → logical → physical) but also in the reverse order (physical → logical → conceptual). An example of the latter is the reconstruction of a conceptual model from the information found in a database. Transformations are usually done manually, which tends to introduce errors into the process. Even if a transformation is done correctly, it can be a very timeconsuming task and is thus not performed as often as it should, leading to inconsistencies between models. Transformations between models are discussed in more detail in section 2.5.

Elmasri and Navathe (1994) support this view of database development, but list some additional concept. First, requirements from prospective database users are collected and analysed (see figure 2.1). From these the conceptual design is done in parallel with functional analysis. The latter gives a high-level specification of the transactions which are required for the database. The transaction specification is then used in conjunction with the logical model to create the physical model of the internal representation of the database. It is also used to create the application program design.

Figure 2.1 shows how database and software development processes use models in different ways. Although the application is developed in conjunction with the database, only one transformation is used; design → implementation. This difference of the development processes is supported by the simplified example given in Pressman (1997, page 34), where it consists of four phases; analysis, design, coding and testing. Transformation between conceptual models is done between the analysis and design phases. This gives a significant
Figure 2.1: Phases of database design, adapted from Elmasri and Navathe (1994)

difference in the modeling tasks between software and database development, and thus also a difference in the requirements for tools and techniques used to handle the models. The same tools used for software development can not be used, or even be extended, for database development because of the differences in development processes.

It can be noted however that both traditional and database development processes needs some kind of transformation of a model to textual form. In the former it is from some type of design model to code, and in the latter from some kind of logical model (e.g. the relational model, see section 2.5.2) to the internal representation of the DataBase Management System (DBMS) (e.g. in the form of SQL, see section 2.5.3).
Chapter 2. Background

Since a complete database application consists of both a database and some sort of User Interface (UI)\(^1\), development would require several different types of description techniques for different development phases (as shown in figure 2.1). To aid the software developer in performing this task, some sort of tool support for the activities involved would thus be beneficial. And for such a tool to support the whole development process, it would therefore need to support all these description techniques (Elmasri and Navathe 1994).

2.2 Tool support for software engineering - CASE

Due to the never ending development of new technologies in the field of computer science, software engineers and other participants in software development have a hard time keeping up with the demands from industry, resulting in a large backlog of applications (Holloway 1993). Some technique or method is thus needed in order to catch up. One solution is to use specialised computer software that aids in the development process; Computer-Aided Software Engineering (CASE).\(^2\) CASE environments allow the developer to create and validate system designs and specifications and to automate and enhance the manual methods of the last 30+ years (Marciniak 1994).

The term CASE is used to represent a wide variety of software tools and also matters regarding software development productivity in general. Although CASE has no single accepted definition, there are several descriptive explanations of the term. For example, Barker and Longman (1992) define CASE as:

The combination of graphical, dictionary, generator, project management and other software tools to assists computer development staff engineer and maintain high-quality systems for their end users, within the framework of a structured method.

---

\(^1\)Most DB applications have a UI, if not the interaction is performed through the SQL interface of the database engine, which however also is a kind of UI.

\(^2\)Sometimes the word "Software" is replaced with "System"
Chapter 2. Background

However, given this broad definition, anything that can aid in the development of software can be regarded as a CASE-tool.

Pressman (1997) likens a software engineer and CASE-tools with a craftsman and his workshop: the craftsman must have a good overview of his set of tools and also understand how to use them efficiently. The software engineer must recognise that the standard handtools alone are not enough, and that there is a need for a workshop in which to place the tools. At a first glance CASE-tools look similar to those used in Computer Aided Design (CAD), which are used in, for example, the design of electronic circuits. However, the physical laws of physics involved in the design of hardware gives natural constraints that shape the behaviour of a CAD-tool. Similar constraints do not exist for software, and hence the CASE-tool can only provide cosmetic advice to the software designer regarding the design of software (Hamlet and Maybee 2001). However, since computer programs are better than humans at performing certain tasks, like checking the consistency and integrity of a software product, CASE-tools can be used to increase the effectiveness of some tasks in the software development process (Connolly et al. 1998). Consequently, the need for CASE-tools has increased as the systems the software engineer develops has increased in size and complexity, since the human being isn’t capable of grasping such large system in her mind.

Connolly et al. (1998) divide database CASE-tools into three categories: Upper-CASE, Lower-CASE and Integrated-CASE. Upper-CASE supports the initial phases of database development; from planning to design. Lower-CASE supports the activities performed in a later stage, namely the implementation, testing and maintenance of the database. Consequently, Integrated-CASE includes both the Upper-CASE and Lower-CASE part. creating an environment in which all phases of database development can be performed. As stated before in section 2.1, if a tool (or a set of tools) is to be really useful, it should support all phases of the development process. Thus Integrated CASE (I-CASE) is the
most interesting category.

Integrated-CASE

To be able to continue to provide automation services of day to day tasks in software development, the CASE environment must evolve to support even more advanced procedures. An important step in this direction is the integration of different CASE-tools, I-CASE, creating an environment in which each tool can perform its assigned task in cooperation with others. To be more precise, each tool must be aware of changes to the project done by another tool, adapting itself as necessary. For example, if an object in a model (handled by a modeling tool) has its name changed, then the name of the object in the resulting code (handled by a code generator tool) should also be changed.

Sommerville (1992) gives three different types of CASE-tool integration:

- Data integration – All tools work on a shared data model.
- User interface integration – The interface of each tool is integrated into a common user interface.
- Activity integration – The environment also contains a model for the software process and can coordinate the behaviour of the individual tools towards a higher goal.

As a solution to this Pressman (1997) describes a CASE integration architecture, consisting of the following components:

- An user interface that provides a consistent look and feel between the different CASE-tools.
- A tool coordination mechanism that allows the CASE-tools to work together.
- A database in which all information is stored.
Chapter 2. Background

- An object management systems that handles changes to the information.

The requirement of data integration implies that each tool must be able to somehow read the results from all other tools. While one way to achieve this is to arrange it so that the result of each tool can be read and understood by other tools, which then can react and transform their own information in response to a change, this method is cumbersome and inevitably results in loss of semantics as information propagates through the network of interconnected tools. The easiest and most flexible solution to the problem is to gather all information in one place from which all tools can read and write - a repository.

2.3 Repositories

A repository is by Websters\textsuperscript{3} definition:

place, room, or container where something is deposited or stored

While the above is true in the context of CASE, it does not give all the details. Barker and Longman (1992, page 372) define a repository as:

a mechanism for storing any information to do with the definition of a system at any point in its lifecycle.

Iyengar (1997) gives another definition:

A repository is a specialised, extensible database application that adds value to a database system by being customised to a particular domain such as application development.

Giving more details, Tannenbaum (1994, page 271) characterises a repository as follows:

- A repository is an integrated holding area.

\textsuperscript{3}Merriam-Websters Collegiate Dictionary
Chapter 2. Background

- The input, access, and structure of a repository and its contents are vendor independent. This means that a CASE-tool can be chosen on its merits as a tool and not because it is manufactured by a special company.

- The metamodel of the repository should be extensible. Although a standard metamodel may cover all of today's storage needs, one might safely assume that new methods and technologies will be developed in the future, and they may thus introduce new requirements for their metamodels.

- The contents of a repository should be retrievable via predefined "views" or "templates".

- Repository contents must be versionable and subject to user-implemented security restrictions. The first enables retrieval of older versions, and it also makes it possible to create different development branches of the content. The latter makes it possible to ensure that the information in the repository is not accessed or modified by a person without authorisation.

- The repository should include the capabilities required to perform all of the preceding essential functionality on a highly functional basis.

Despite the type of model being stored (logical data model, physical program structures, etc.), it can be related to other distinctly different model types that exist elsewhere in the repository (sometimes called ‘data dictionary’). Thus a repository can be used to store information that extends beyond what the individual CASE-tools will use.

The benefit of using a repository in the software development process is that all information regarding the system which is built, from all phases of the process, can be stored within the repository. This way the data can be interlinked, insuring that all parts of the

\footnote{Using a wide definition of CASE most, if not all, tools involved in the software development is a CASE-tool}
system being built are integrated. And since all information in the repository is interrelated, a CASE-tool can do various consistency checks on the data stored in the repository. Holloway (1993, page 66) lists the following checks:

- Multi-specification consistency – checks that spans over two or more specifications but on the same level.
- Interface verification – statements must comply with the high level interface definitions.
- Hierarchical completeness – finds statements where more information is expected but not yet defined. It can also find superfluous specifications.
- Integrity – prevents developers from making changes to the repository contents without proper authorisation.

Classifications

Repositories can be classified in various ways, and it can be helpful to make some distinctions between different types of repository. The simplest case is the standard files created by a CASE-tool. The purpose of such files is to act as persistent storage, and is normally only read by the tool that created it. The internal structure of the file can be more or less advanced, and at a certain level of complexity the file can be seen as a small dedicated database for the specific CASE-tool. Another class of repository emerges when other tools are allowed, and encouraged to read from and write to the repository. This is called a distributed repository\(^3\). As discussed before, a repository is most useful if it is available and open to any application that is used in software and database development. Therefore a distributed repository is an important component of the development environment.

\(^3\)The word *distributed* is usually used in computer science in a physical sense, meaning that something is dispersed on different computers. While this certainly is true in this case, it can also mean that information is distributed to different programs.
Chapter 2. Background

Services

To be available and open a repository should provide certain services. Iyengar (1997) regards the following as most useful according to the general consensus:

- Metadata or model/schema management that supports a semantically rich object model.

- Management of changes to the data in the repository, e.g. version and configuration management.

- Tool/object integration and registration services.

The last point allows CASE-tools to work together, and also makes the repository more meaningful. But in order to support integration, each tool must have some notion about the schema used to store the information. That is, schemas must be standardised if integration of tools is to succeed. However, if a tool is as extensible in regard to the handling of new schemas as the repository is in storing the information within it, new schemas can easily be added, although with some effort needed to customise the tool. One important feature of the repository is the openness of the datastructures contained in it. In many older CASE-tools the data was stored in a proprietary database which only allowed tool access for tools from a certain vendor. If the contents of the repository is accessible for any application, the ability to work together is increased.

To support these services, metadata and schema management is needed to provide a seamless integration of tools. To be able to provide a rich object model some sort of extensive schema must be supported. The means for describing the data in the repository is called a common meta-language or meta-model. This is discussed in more detail in the following section (2.3.1). More on the subject of repositories is given in section 2.4.4, where the repository standard MOF is introduced.
2.3.1 Metamodels

*Meta-data* is a detailed description of data instances, i.e. the definition, format and characteristics of populated data (Tannenbaum 1994). A *model* is the storage structure that is used to hold and store the meta-data. A model can be more or less complex, and two models can be very different but still describe the same domain, as shown in figure 2.2. Model (a) can capture *most* countries, but as some countries have more than one capital\(^6\)

![Diagram](image)

Figure 2.2: An example of two models

only model (b) can handle this. And it is, of course, possible to find a situation which model (b) cannot handle either, but it all comes down to how accurately the model should describe the world.

If each model were hardcoded into the repository, this would mean that it would be very hard to change it, or add new models. But as each model is built up from similar concepts (e.g. entity/class and relation/associations) it is possible to describe and store it using a *metamodel*. A simplified metamodel of the UML classmodel is shown in figure 2.3 (see section 2.4.1 for a description of UML).

\(^6\)e.g. Bolivia, Cote d’Ivoire, South Africa
Figure 2.3: A simplified metamodel of the UML class model

In this metamodel it is possible to store both of the example models of figure 2.2, (a) and (b). Thus, a repository can be divided into four layers as given in figure 2.4. The different layers have different purposes. The layer at the bottom represents the actual data stored in the repository. The second layer from the bottom describes the model of the underlying data, thus giving data about data, meta-data. The third layer describes the model of the model, metamodel, in which meta-metadata is stored. This layer is also known as the repository metamodel. The reason behind this layered approach is that given that the repository metamodel is expressive enough, it can describe all (at least most) models and therefore the repository is also capable of storing arbitrary information structures. The top layer (dashed in the figure) uses hardcoded concepts for describing the underlying model; it can thus not be changed easily.

Modeling languages are further discussed in a following section (2.4.1).

2.4 Standards

This section introduces some of the standards used in the field of research, namely UML (2.4.1), OCL (2.4.2), CORBA (2.4.3) and MOF (2.4.4).
2.4.1 UML - Unified Modeling Language

As systems grow more and more complex, visualisation and modeling of these become essential. The Unified Modeling Language (UML) was created as a response to this, and has become a widely accepted, well defined standard. UML is a language for specifying, visualising, constructing, and documenting the artifacts of software systems, and it also represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems (UML 2000). UML is an industry standard created and maintained by the Object Management Group (OMG).

Before UML was born there was no single leading modeling language. The disagreement between similar but slightly different concepts found in tools discouraged many users from entering the object technology market (UML 2000). The high cost of supporting many modeling languages motivated many companies producing or using object technology to endorse and support the development of UML. Using UML does not guarantee that a project will succeed, but it does improve several things. One of these is the integration of tools, since an agreement can exist on which concepts to use, and the attributes of these
Chapter 2. Background

concepts.

The primary goals of UML are as follows (UML 2000):

- Provide an expressive visual language that can be used to develop and exchange models.
- Provide mechanisms to extend the core concepts.
- Support specifications that are language and process independent.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the object tools market.
- Support the use of higher level concepts that aren’t included in the basic specification
  of UML, e.g. components, collaborations, frameworks and patterns.
- Integrate best practices.

The architecture of UML is based on a four-layer metamodel structure as described in

Table 2.1, which can be compared with figure 2.4 in section 2.3.1, Layers of a repository.

An UML model can be described in a combination of both graphical and formal notation,

the latter specified in OCL. There are 9 predefined diagrams in UML (Eriksson and Penker

2000), see table 2.2 for a list and a short description of each7. UML’s flexibility is increased

as it is possible to add new diagrams and elements to it, due to UML’s unique capability

to adapt and extend (Eriksson and Penker 2000).

With regard to software development the class diagram is probably of most interest, as

it is used to describe the structure of the software system. A class has a name and a set

of attributes and operations. The name is unique within the diagram and usually describes

what the class represents. Attributes are valueholders, i.e. names that are associated with

a value, and are usually of a certain (data) type, e.g. integer or string. Attributes marked

7For more information on UML refer to (UML 2000)
Table 2.1: Layers of the UML architecture, adapted from UML(2000), page 2-4

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta-metamodel</td>
<td>The infrastructure for a meta modeling architecture. Defines the language for specifying an model.</td>
<td>MetaClass, MetaAttribute, MetaOperation</td>
</tr>
<tr>
<td>metamodel</td>
<td>An instance of meta-metamodell. Defines the language for specifying the metamodells.</td>
<td>Class, Attribute, Operation, Component</td>
</tr>
<tr>
<td>model</td>
<td>An instance of a metamodel. Defines a language to describe an information domain.</td>
<td>StockShare, askPrice, sellLimitOrder, StockQuoteServer</td>
</tr>
<tr>
<td>user objects</td>
<td>An instance of a model. Defines a specific information domain.</td>
<td>&lt;acme_SW_Share_98789&gt;, 645.56, sell.limit.order, &lt;StockQuote_Svr.321123&gt;</td>
</tr>
</tbody>
</table>

by a plus sign (+) are publically available, i.e. any class can read and write the attribute value, while those marked with a minus sign (-) are private to the class itself. Attribute may also be underlined, which means that the value is common for all instances of that class. An operation is a predefined algorithm which can be called using its name and parameters.

The relationships between classes, called associations, are described with a name, roles and multiplicity. As with classes, the name of a relationship describes its function, but it may also tell in which direction the name should be interpreted. A class can have several relations and it can play different roles depending on the relation. Multiplicity tells how the classes are related to each other numerically. Figure 2.5 shows an example of a UML class diagram.

It can be difficult or even impossible to fully model a system or organisation using only the notations and concepts of the class diagram introduced above. Sometimes there is a need to specify exceptions or constraints that the system must fulfil, i.e. to more carefully describe states that the system is allowed to enter. UML has several ways of
Chapter 2. Background

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Describes the structure of a system using classes and relationships between them. The classes can represent information, products or organisations.</td>
</tr>
<tr>
<td>Object</td>
<td>Shows possible object combinations of a class diagram, i.e. an example of how classes might be instantiated.</td>
</tr>
<tr>
<td>Statechart</td>
<td>Describes the possible states of a class or system.</td>
</tr>
<tr>
<td>Activity</td>
<td>Describes actions and activities that takes place in a system.</td>
</tr>
<tr>
<td>Sequence</td>
<td>Shows how sequences of messages are sent between among a set of objects.</td>
</tr>
<tr>
<td>Collaboration</td>
<td>Shows how a set of objects collaborates to fulfil a task.</td>
</tr>
<tr>
<td>Use-case diagram</td>
<td>Defines the functionality of a system in both graphical and textual form.</td>
</tr>
<tr>
<td>Component</td>
<td>Shows the components that a software system is built of using a modified class diagram.</td>
</tr>
<tr>
<td>Deployment</td>
<td>Shows how hardware is utilised within a software system.</td>
</tr>
</tbody>
</table>

Table 2.2: Predefined diagrams of UML

specifying constructs. One is to use graphical notation, drawing a dashed line that cross relationships between classes and writing the constraint in natural language, or using simple logical operators such as XOR, see figure 2.6. A record can be produced by either one or more record companies or one or more bootleggers, not both (i.e. there cannot be a mix of recordcompanies and bootleggers that produces a record).

However, this way of specifying constraints cannot express more advanced constructs in an unambiguous way. Thus there is a need for a more formal way of defining constraints.

2.4.2 OCL - Object Constraint Language

Object Constraint Language (OCL) is a formal language, part of the UML standard (UML 2000) since version 1.1, used to express constraints. It can be used to add application-specific constraints to models, since a graphical diagram typically is not refined enough to provide all details. An alternative is to describe the constraint in natural language, which could result in ambiguities. On the other hand, creating the whole specification in a formal
language can be difficult and require extensive knowledge and experience in the field of mathematics. OCL was therefore developed to fill the gap between the two techniques. Although it is a formal language it is easy to learn to read and write (UML 2000).

OCL has many uses, and UML (2000) lists some:

- To specify invariants on classes and types in the UML class model. An invariant is a condition that must be true at all times for objects of an class.

- To describe pre- and post conditions on Operations and Methods. Thus it possible to specify and constrain the possible parameters and result of operations performed on
an object.

- To describe Guards, i.e. a condition that specifies whether to perform a specific activity or process, or, if it exists, an alternative.

- As a navigation language, to move from one specific context to other associated objects in the same model.

- To specify a derivation rule, i.e. how a value is calculated from values of other attributes (Eriksson and Penker 2000).

The OCL language consists of expressions that use operators and operands and returns a result value. Expressions in OCL are guaranteed not to change the state of the system; it will only return a value and cannot change anything stored in the model. Each expression is related to a certain context, which is a specific part of a model in which the expression should be evaluated. The context can for example specify which objects or classes which should be involved. It is specified by using the OCL context keyword:

context Record

As OCL is tightly bound with the UML modeling language, it is of course ideal for handling models that are described using UML. Figure 2.7 shows how an OCL expression is used to specify an UML constraint. In this example the price of a record should always be two times greater than the number of tracks on the record.

One great advantage of using OCL for documenting requirement specifications is that while it is a formal language with a strong connection to the models, it is still in a form possible to convert to real world programming constructs. This is described in Wiebicke (2000). There an UML model is transformed into a Java code skeleton. The constraints specified in the UML model using OCL are also transformed into Java code, thus avoiding loss during manual interpretation and code generation.
2.4.3 CORBA

Common Object Request Broker aArchitecture (CORBA) from OMG is an open, vendor-independent specification for an architecture and infrastructure for making computer applications work together, distributed over a network. It allows CORBA-based programs from any vendor, on almost any computer, operating system, programming language, and network, to interoperate with each other. The specification has two major parts: OMG Interface Definition Language (OMG IDL) and the two standardised protocols GIOP and IIOP.

CORBA applications consists of one or more objects of different types. There can be several instances of an object type or just one, depending on how the application looks like. For example is it possible to wrap a legacy application and present it as a CORBA object. Each object used in CORBA must have an interface specified in OMG IDL. The interface tells other objects how they should formulate requests, and how arguments and the result of the request should be arranged. The language independence of the interface definition contributes to the high interoperability of CORBA programs.

In some ways the requirements of CASE-tool integration as stated in section 2.2 are supported by CORBA as an implementation as it facilities easy communication between programs and systems, and evens out differences in protocols and architectures.

2.4.4 MOF - Meta-Object Facility

In conjunction with CORBA and UML, OMG has specified Meta-Object Facility (MOF) which defines a set of CORBA OMG IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models (Object Management Group 2000). Metamodels include the UML metamodel and the MOF meta-meta model. The MOF provides an infrastructure for object reuse and a repository which can store descriptions and definitions of the fundamental concepts that applications work with. MOF should not be seen as a database in the traditional sense. Instead, the primary intended
use is to store descriptions of models and metamodels, which can among other things be used to automatically build code skeletons for implementation. Storing metamodels in a common format also makes it possible to join two models, e.g. in a datawarehouse storage schema.

Both UML and MOF are based on a four-layer metamodel architecture and considerable effort has been expended in assuring that they are architecturally aligned. Thus it is possible to visualise the MOF meta-models using UML notation.

MOF meta-metamodel is the meta-metamodel for the UML metamodel, and as a result the UML metamodel may be considered as an instance of the MOF meta-metamodel. Most of the metaobjects in UML core metamodel can be mapped to those in the MOF core metamodel and conversely. Further, MOF and UML have been aligned to match those core objects found in the CORBA object model, although many concepts found in MOF/UML are missing in the latter. As such, MOF supports all datatypes of CORBA.

The meta-metamodel of MOF can be extended by inheritance and composition to define a richer information model that supports additional constructs. Thus, using a small set of primitive concepts as specified in the MOF, it is possible to define arbitrarily complex metamodels that may differ from the main philosophy of the MOF. All these metamodels and their models can be manipulated through the OMG IDL, including creating, accessing and updating information stored within them, all while their structural and logical consistency is maintained.

One application of the MOF would be as specified in 2.2, as a central repository for storing the result of the analysis and design of software development.

MOF uses a layered metadata architecture that is based on the traditional four-layered meta-modeling architecture. However, MOF reaches further by not limiting the number of layers of meta-information; as long as each level is described using either the primitive concepts of the top hardwired MOF model or concepts described in an underlying meta-model, there can be an arbitrary number of meta-levels. However, the underlying architecture
of MOF still consists of the traditional four layers (as shown in figure 2.8), which can be described as follows:

- The user object layer, consisting of the actual information or data.
- The model layer describes the user object layer, and thus is called meta-data. As the name suggest, this information is most often aggregated into models.
- The meta-model layer defines the structure and semantics of the meta-data, thus creating a language for describing meta-meta-data.
- The meta-meta-model layer is used to describe the storage structure and semantics of the meta-meta-data.

While the four layer meta-data architecture has the advantage of being able to support most meta-information, this is under the assumption that the overall semantics of the MOF model is rich enough.

The main four modeling concepts of MOF is:

- Classes which models MOF objects.
- Associations which models binary relations between objects in the MOF.
- Datatypes which models other data, e.g. primitive types
- Packages which can be used to modularise the models.

Classes has among other things attributes which defines value holders and operations which defines behaviour associated with a class.

Association is, contrary to UML, only binary. That is, at most two classes can be be involved in one association.
2.5 Transformations

This section introduces the concept of model (or scheme) transformation, giving some useful definitions in the area.

Models of software can be created from a sequence of transformations applied on initial specifications. This is an important feature especially in the database domain. Modeling software design as the systematic transformation of formal specifications into efficient programs, and building CASE-tools that supports it, has long been considered as one of the ultimate goals of software engineering (Hainaut 1996). Transformations can also be used for integrating databases by creating equivalent schemas (McBrien and Poulavassilis 1997), or for improving a database schema for conceptual, logical or internal optimisation (Halpin and Proper 1995).

Partsch and Steinbrüggen (1983) give a definition of transformation. First, a program is a formal description of a method. A class of related programs can be represented as a program scheme, which originates from a program by parameterisation. In reverse, a program can be obtained from a program scheme by instantiating the schema parameters. A transformation is a relation between two program schemes $P$ and $P'$. The transformation is said to be correct if a certain semantic representation (depending on the program schemes) holds between $P$ and $P'$. A transformation can be seen as an operator $T$ that replaces one type of construct $C$ in one scheme $S$ with another construct $C'$, producing a scheme $S'$.

There are three cases of transformation (Hainaut 1996):

- $S = S'$: If the scheme $S$ and $S'$ is equivalent then they are said to express the same semantics, and thus the operator $T$ is *semantics-preserving*.
- $C = \{\}$: If $C$ is empty this means that the operator $T$ is adding information to the scheme $S'$, and $T$ is said to be *semantics-augmenting*.
- $C' = \{\}$: If $C'$ is empty information is removed in the transformation and the operator $T$ is called *semantics-decreasing*.
Chapter 2. Background

The fourth case would be when both C and C’ is empty, thus making no change at all. Besides converting the scheme, individual instances in S should also be transformed, giving an operator t. A full scheme transformation is thus specified by the mapping \(<T,t>\).

**Previous work on transformations**

Partsch and Steinbrüggen (1983) reviews and classify transformation systems, and defines some important concepts. Hainaut (1996) uses the above definitions for specifying a general purpose data structure specification model Generic Entity Relationship (GER), which can be used on a wide variety of models e.g. ER-models. GER uses an extended NINF for the formal specification. The transformations are predicate based, meaning that a generic transformation scheme can be instantiated by substituting its parameters.

McBrien and Poulouvasilis (1997) give an unifying formalisation of the ER schema transformation process, and define 11 primitive transformations that can be applied on an ER schema to uniform it with another schema. A full formal definition can be found in McBrien and Poulouvasilis (1998).

Gustavsson and Lings (2001) presents an extension to OCL which together with an extended meta-model "allows for fuller interchange of models between CASE-tools". The goal of the work is to define a common, model independent, notation for exchanging design transformations between tools. This would thus makes it possible to use a transformation in many different tools, and also to adapt and extend a transformation to support, for example, a new tool.

2.5.1 Metacase

CASE-tools usually supports only a fixed number of methodologies, while software development organisations often make changes to the methods they use (Isazadeh and Lamb 1997). MetaCASE tries to solve this problem by making the methodology automation more dynamic. The aim of MetaCASE is to provide meta-tools that makes building quality CASE
tools quicker and easier (Gray, Liu and Scott 2000).

A CASE-tool customer uses the modeling techniques of the metaCASE tool to specify how the methodology works, and how it should be handled in the tool. One benefit of using a metaCASE-tool is that it can be adapted to several methodologies. Thus it requires less training to introduce an user to a new methodology, as they are used to the working environment. According to Isazadeh and Lamb (1997) cost is one of the major factors why organisations hesitates to adopt CASE-tools.

So, to be able to easily implement new methodologies a generic software solution is tractable, one which preferable is configurable by the end-user to adapt as neccesary.

In the database development many different models are used, both conceptually and to represent storage structures.

### 2.5.2 Models

The lowlevel language or functionality of a DBMS is usually to abstract and difficult for a normal user to understand. This is simplified by using a high-level description of the database schema using a datamodel which Connolly et al. (1998) defines as:

> An integrated collection of concepts for describing data, relationships between data, and constraints on the data in an organisation.

Over the years of research many models has been proposed, and Connolly et al. (1998) classifies them into three categories; object-based, record-based and physical datamodels. Some common data models in the first category are Entity-Relationshhip, Semantic, Functional and Object-Oriented. Physical data models describes how data is stored in the computer by specifying information about record structures, record orderings and access paths, details which in other models usually are hidden from the user through some highlevel language. Record-based (sometimes also called representational or implementational) models includes the network, hierarchical and relational model (see 2.5.2), the latter being the currently most used.
Chapter 2. Background

Relational model

In June 1970 the idea of the relational model was first presented (Codd 1970), an improvement over the navigation-based models (e.g., graph, network, or hierarchical) then used. The advantage over previous methods is that the relational model "provides a means of describing data with its natural structure only, […] without superimposing any additional structure for machine representation" (Codd 1970). Some other advantages of the relational model are:

- It provides a sound mathematical foundation for dealing with derivability, redundancy and consistency of relations. The relational model has an important characteristic, closure, meaning that all relational operations produce relations, making it possible to prove the correctness of many of these operations (Melton and Simon 1993).

- It provides independence of the data from the physical representation, independence of the relationships from the data, and independence of the implementation issues related to for example efficiency.

The relational model provides a basis for a high-level data language which gives maximum separation between machine representation of the information and programs using it. Until then, the database paradigms in use could not provide independence when considering ordering, indexing and access methods of the information, therefore implemented programs would break when parameters that they relied on was changed.

The basic unit of data is the relation, which is known as a table in SQL (see 2.5.3 for more on SQL) with the difference that the latter accepts duplicates. A relation consists of attributes which are called columns or fields in SQL, each associated by a datatype. Data in a column comes from the same domain of values. Data is stored in a (database) table in tuples (or rows). The number of rows in the table are likely to change many times during

---

8Elmasri and Navathe (1994), Connolly et al. (1998) and Melton and Simon (1993) have been the primary source of information for this section.
its lifetime as data is inserted and removed. The number of columns are however most often of a static nature, and will not change if not a need for storing more or less information is found. An example of a relation is shown in table 2.3. Data in the column RecId consist of numbers and thus is associated with the integer datatype, while the Artist and Record are of the character datatype. The last column Year (which is a date) could be implemented in a number of ways, depending on the support from the underlying database engine. One solution is to use a date datatype, but one could also choose to represent the attribute as an integer or characterstring. The relational model does not implicitly specify any datatypes, this is upto an actual database engine implementation to decide.

An important characteristic of a datatype is that it is atomic, i.e. its contents are undivisible and is handled as unit (even if an e.g. characterstring can be thought to be broken up into separate characters). Tuples are nonordered as the relational model does not specify this.

An important concept of the relational model is constraints, which are rules that ensures the integrity of the database. The relational models includes four different types of constraints; domain integrity, keys, entity integrity and referential integrity.

Domain constraints specify that each attribute A must be an atomic value from the domain dom(A), and thus governs the integrity of the tuple or entity. i.e. all values in that column must come from the same valuedomain, e.g. integers.

<table>
<thead>
<tr>
<th>Record</th>
<th>RecId</th>
<th>Artist</th>
<th>Album</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Rage Against the Machine</td>
<td>Rage Against the Machine</td>
<td>1992</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Rage Against the Machine</td>
<td>Evil Empire</td>
<td>1996</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Rage Against the Machine</td>
<td>Battle of Los Angeles</td>
<td>1999</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Persuader</td>
<td>Stockholm</td>
<td>1999</td>
</tr>
</tbody>
</table>

Table 2.3: An example of a database relation/table populated with data
All tuples in a relation must be distinct, e.g. no tuple can have the same values for its attributes as another tuple. Usually there is a subset of the attributes of a relation schema R that satisfies this property as well, e.g. no tuple has the same values for these attributes as another tuple. Thus this subset of attributes identifies each tuple uniquely, and the subset is known as a key. In table 2.3 the attribute RecId is distinct in the relation and thus identifies each tuple uniquely, while the attribute Artist does not and thus can not serve as a key. However Artist combined with Year could, since the relation does not have two tuples with these attributes identical.

Let’s assume the relation is populated as shown in table 2.3 and that the attribute RecId is an unique key. If a tuple ( 4, Erykah Badu, Baduizm, 1997 ) is inserted into the relation this would violate the key constraint of the relation (since there already exists a tuple with RecId = 4), thus the operation is not legal. Key constraints prevents multiple instances of the same key to exists simultaneously in a relation, and hinders that e.g. a social security number is used twice for two different persons.

The relation in table 2.3 can be written as shown in figure 2.9:

Attributes that are part of the identifying key of a relation is by convention underlined as shown on the attribute RecId. Notice that a relation can have several candidate keys, however, only the primary key is underlined.

Entity integrity constraints states that no primary key value can be null \(^9\). Allowing two keys to be of value null would mean that some tuples wouldn’t be uniquely identifiable.

The last type of constraint are the referential integrity that operates on relationships between relations. This is easiest shown giving an example. The relational schema of figure 2.9 is extended with a new relation (see figure 2.10)

\(^9\)N.B. Null \(\neq 0\). Null is a special value that essiiantially means ”no value".
Chapter 2. Background

The overlining of the attribute **RecId** denotes that it is a *foreign key*, i.e. the attribute is a primary key in some other relation. In a real database engine the actual target relation of the foreign key would somehow be described by giving the table name, i.e. "the attribute **RecId** of the relation **Sale** is a foreign key of the attribute **RecId** in the relation **Record**". Also note that it is not necessary for the foreign key attribute to be named exactly as the primary key it points to; e.g. **RecId** in the relation **Sale** could be named **fkcRecId** to denote that the attribute is a foreign key. Populated with some data the relation could look like shown in table 2.4:

<table>
<thead>
<tr>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>SaleId</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.4: The sale relation populated with some data

Now assume that the tuple (4, Persuader, Stockholm, 1999) is removed from the **record** relation. This would mean that the foreign key **RecId** of the relation **Sale** would refer to a non-existing tuple! This is avoided by using referential integrity constraints which controls the consistency of the relationship between the two relations, and would prevent the removal of the tuple.

An attribute can be both a foreign key and part of the primary key simultaneously. Suppose the relational schema (figure 2.9 and 2.10) is extended with another relation (figure 2.11) which is show populated with some data (table 2.5).

Each track is identified by a track number. This is however not necessary to identify it uniquely\(^{10}\). As each track is associated with a record, a track can use this **RecId** attribute

\(^{10}\)An alternative would be to use **Title** as a key, but that would only work if no two track has the same title
value as pointer to the record. Thus each track is uniquely indentified by the combination of the TrackNo and RecId attributes, as they forms a primary key.
Chapter 2. Background

Figure 2.6: An example of an UML constraint

Figure 2.7: An example using OCL for specifying an UML constraint

Figure 2.8: The four meta-data layers of the MOF, adapted from The Object Management Group (2000)
Record( RecId, Artist, Album, Year )

Figure 2.9: A relation with one key

Sale( SaleId, RecId, Date )

Figure 2.10: A relation with one foreign key

Track( TrackNo, RecId, Title )

Figure 2.11: A relation there a foreign key is part of the primary key
## Chapter 2. Background

<table>
<thead>
<tr>
<th>Track No</th>
<th>Rec Id</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Bombtrack</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Killing In The Name</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Take The Power Back</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Settle For Nothing</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Bullet In The Head</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>Know Your Enemy</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>Wake Up</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Fistful Of Steel</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>Township Rebellion</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>Freedom</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>People Of The Sun</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>Bulls On Parade</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>Vietnow</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>Revolver</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>Snakecharmer</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>Tire Me</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>Down Rodeo</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>Without A Face</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>Wind Below</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>Roll Right</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>Years Of Tha Boomrang</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Track No</th>
<th>Rec Id</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>Testify</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Guerrilla Radio</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Calm Like A Bomb</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Mic Check</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Sleep Now In The Fire</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>Born Of A Broken Man</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Born As Ghosts</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>Maria</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>Voice Of The Voiceless</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>New Millennium Homes</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>Ashes In The Fall</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>War Within A Breath</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>Rimshot</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>On &amp; On</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>Appletree</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Otherside Of The Game</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>Sometimes</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>Next Lifetime</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>Afro</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>Certainly</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>4 Leaf Clover</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>No Love</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>Drama</td>
</tr>
</tbody>
</table>

Table 2.5: The track relation populated with some data
2.5.3 Code

SQL

The history of SQL began in 1974 when D. Chamberlin at the IBM Research Laboratory in San José defined a language called *Structured English Query Language* or SEQUEL\(^\text{11}\). In 1976 the language was revised and released as SEQUEL/2, however the name was later changed to SQL for legal reasons. In 1987 SQL was standardised by ISO, which revised it in 1989, adding integrity enhancement features. SQL2, or SQL-92 as it is also called, was released in 1992, and was the first major revision of the language. By now implementations by a large number of vendors were available, each with their own SQL dialect. The current version of the standard is SQL3 or SQL-99, published in 1999. It adds many new features, however most of the original concepts still remain backwards compatible.

Although there exists several other query languages for databases (e.g. Object Query Language (OQL) which is part of Object Data Management Group (ODMG) Object Data Standard (Cattell, Barry, Berler, Eastman, Jordan, Russell, Schadow, Stanienda and Velez 2000)), SQL is still the ruling language of the industry. As Stonebraker (1990) says: "For better or worse, SQL is intergalactic dataspeak". Since almost all database products today have some notion of understanding SQL, it is difficult to introduce a new language onto the market.

The primary function of the SQL language is to support the definition, manipulation, and control of data in SQL databases. A SQL database uses many of the concepts introduced in section 2.5.2, i.e. tables, columns and rows. The basic SQL statements can be roughly be divided into two groups: data definition and data manipulation. Tables are defined and created using the *CREATE TABLE* statement, as shown in figures 2.12, 2.13 and 2.14 there the relations specified in figures 2.3, 2.4 and 2.5 respectively is created.

Figure 2.12 shows how a basic table with some fields and a primary key are created.

\(^{11}\)Connolly et al. (1998) and Date and Darwen (1997) has been the primary source of information for this section.
Chapter 2. Background

CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Album CHAR(50),
    Year CHAR(4),
    CONSTRAINT record_pk1 PRIMARY KEY (RecId)
)

Figure 2.12: A SQL statement for creating the table Record

CREATE TABLE Sale(
    SaleId INTEGER,
    RecId INTEGER,
    Date CHAR(20),
    CONSTRAINT sale_pk1 PRIMARY KEY (SaleId),
    CONSTRAINT sale_pk1 FOREIGN KEY (RecId) REFERENCES Record(RecId)
)

Figure 2.13: A SQL statement for creating the table Sale

CREATE TABLE Track(
    TrackNo INTEGER,
    RecId INTEGER,
    Title CHAR(20),
    CONSTRAINT track_pk1 PRIMARY KEY (TrackNo, RecId),
    CONSTRAINT track_fk1 FOREIGN KEY (RecId) REFERENCES Record(RecId)
)

Figure 2.14: A SQL statement for creating the table Track

36
Chapter 2. Background

In SQL it is possible to create keys (indices) of two types: primary or unique. While a table can only have one primary key, it is possible to have multiple unique indices. An unique key can be specified by:

CONSTRAINT table_un1 UNIQUE (TrackNo, RecId)

As can be seen from this example, the only thing that is different from a PRIMARY KEY constraint definition is the keyword UNIQUE. The usage of a constraint name (specified after the keyword CONSTRAINT) is encouraged for several reasons. First, it makes it possible to later drop (remove) the constraint, and secondly, should the execution of a SQL statement result in error due to a constraint violation, the name of the constraint can be found by looking at the error messages.

In some cases CREATE TABLE is not enough, for example if a column or constraint is to be added or removed after the table has been created. Then the command ALTER TABLE can be used, as shown in figure 2.15. First a new column NewColumn is added to the table Track. Then the column is made a foreign key that references the column SaleId in the table (just as an example). Finally, the foreign key constraint and the column itself is removed.

ALTER TABLE Track ADD COLUMN NewColumn INTEGER

ALTER TABLE Track ADD CONSTRAINT fk_track_newcolumn
    FOREIGN KEY (NewColumn) REFERENCES Sale(SaleId)

ALTER TABLE Track DROP CONSTRAINT fk_track_newcolumn

ALTER TABLE Track DROP COLUMN NewColumn

Figure 2.15: A SQL statement that adds a new column and constraint to table Track and then removes them
Chapter 3

Problem Description

Our goal is to investigate the feasibility of automatically transforming a model of some type, represented using a model in MOF/UML, to a representation in textual form. The model will be described using UML while the transformation rules will be described utilising the OCL language, a subset of UML.

3.1 Motivation

A complete model of an information system often consists of a set of models on different levels, each describing some subset of the information system modeled. To be able to trace changes done to the model one must be able to map between models on adjacent levels, so that actions taken in one level are executed on all interconnected levels.

The goal of this project is to investigate the possibility of transforming a model of some type into textual form, suitable for existing computer software, e.g., a database or programming language.

Transformations between different kinds of models is well researched. However, the mapping between a model and its textual form is usually described directly in the implementation language. It would thus be beneficial to specify the mappings as an independent
model, since changes to this could be made more easily.

In this project only the forward direction transformations are addressed, not transform-
mations from a textual language (e.g. SQL) to a model (e.g. relational model).

3.2 Objectives

The basic hypothesis of this work is that a general model that can describe the transfor-
mation between different types of models can be found and specified. By storing transformation
rules in this model they will be separated from the implementation of both the CASE-tool
and the repository.

Rules should be designed that use the general transform model for transforming a par-
ticular example model, e.g. a relational model or object model, into a corresponding textual
representation, e.g. SQL or XMI. One way to do this is to use delta (incremental) changes
to build a database from a relational model, utilising SQL ‘ALTER TABLE’ statements.

A more advanced aim is to consider full transformation to SQL-92, complete with con-
straints, creating tables utilising SQL ‘CREATE TABLE’ statements, while minimising the
use of ALTER TABLE statements as the latter makes the SQL-code more difficult to read
and understand.

While the above work is being done a further goal was to investigate whether OCL is
sufficient for transforming a model to text, or whether extensions are necessary.

3.3 Previous work

As stated in section 2.5, much work on using OCL for transformations has been done by H.
Gustavsson and B. Lings, as presented in Gustavsson and Lings (2001). An implementation
has been created, capable of transforming EER-diagrams to relations.

As many concepts and ideas they use should be adaptable for this problem too, a short
summary of their work is given below.
Chapter 3. Problem Description

Their goal is to "define a common, model independent notation for design transformations exchange between tools, so that the meaning of different transformation will remain consistent between different CASE-tools".

The major contributions to solving this problem are conservative extensions to the OCL language and a MOF UML-based meta-model.

3.3.1 Extensions to OCL

A feature of OCL is its state preserving nature, and it must thus be extendend to allow modification of a model. Gustavsson and Lings (2001) add an assignment operator, denoted by the symbol ':=' to the OCL language. The assignment operator can be used to to modify both properties and roles. While the assignment of a property is straightforward, roles are slightly more complex as they consists of collections of associations. This means that when a role is assigned a new value, the old collection is removed and replaced with the new.

In addition to modifying an schema object, it may be neccesary to add or remove objects. Thus the operators create and delete are added to solve this. Only objects previously declared can be created. This done by introducing declaration aliases which work like ordinary aliases, except that they may refer to an object that does not yet exist.

Another extension to OCL is the separator symbol, ";", used to separate components of an expression that is to be interpreted independently. Thus several modifications can be grouped together to form a single OCL expression.

3.3.2 Extensions to the Meta-model

Transfer of a model and its accompanyng transformation procedures requires that a common meta-model is used to express them. Figure 3.16 shows the suggestion from Gustavsson and Lings for this common meta-model.

The most significant feature of this model is the ability of a model element to directly refer to a transformation pattern, and thereby transitively refer to rules and procedures.
The class ModelElement is a superclass of all other classes in the repository. This means that the transformation procedures can use information from several different independent models, and this without having to specifically alter the meta-model.

Patterns are used to store choices done by the user regarding transformations. For example, if a model element can be transformed in two different ways, the user can choose one, and this will be ‘remembered’ for all later transformations.

### 3.3.3 Objectives restated

Since this work in much depends on the ideas and concepts introduced above, it can be worthwhile to discuss and restate the objectives.

Since the overall framework for transforming models presented by Gustavsson and Lings (2001) is proven to work\(^1\), it has been used as a basis for approaching this problem too. A question that arises is which extensions are necessary for handling text, since it is a totally different target format.

Likewise for OCL, many extensions designed for modifying a model in real time has already been proposed. Thus, it should be investigated whether further extensions to OCL are necessary for handling text.

\(^{1}\text{In the sense that a proof-of-concept implementation exists.}\)
Chapter 3. Problem Description

With this done, procedures should be built to transform relations to SQL, first utilising ALTER TABLE statements, and then if possible with CREATE TABLE statements.
Chapter 4

Implementation

This chapter presents a solution to the problem introduced in chapter 3. First some important concepts are defined in section 4.1. In section 4.2 an example source model, a relational model, is presented together with the results from the transformation. In section 4.3 the overall strategy for how text is built up is presented, the handling of updates and the requirements for event declaration are discussed, and the general transformation is presented together with some example transformations.

4.1 Definition of concepts

The general transformation model is a storage structure for rules. When one or more rules are grouped together they form a ruleset, capable of taking data from a particular information domain, i.e. source model, and generating a textual representation according to some standard or convention.

4.2 An example model

The general transformation model and the transformation rules should be able to transform any type of model. However, it is useful to show some example model and the desired result
as it may serve as a basis for the discussion. A specific instance in model transformation is that of the relational model to SQL code.

In figure 4.17 a meta-model of the relational model is shown. A relation has attributes, each of which can take part in building an unique key. Attributes can also take part in a foreign key, which references a primary key. Foreign- and unique keys each belong to a specific relation.

A simplification that is made in the following example is that a relation can have multiple unique keys. However, in the following examples only primary keys are used.
Figure 4.17: A meta-model of the relational model
Chapter 4. Implementation

This model can be used to create the following examples:

- A relation with attributes, figure 4.18, transformed to SQL-code, figure 4.22.
- A relation with a primary key, figure 4.19, transformed to SQL-code, figure 4.23.
- A relation with a foreign key that references another relation, figure 4.20, transformed to SQL-code, figure 4.24.
- Two relations with foreign keys that reference each other, figure 4.21, transformed to SQL-code, figure 4.25. While this is the optimal solution, a simpler but not optimal method would be to divide the creation of tables into three groups: the actual creation of tables and their attributes; the creation of primary keys; and the creation of foreign keys. This is shown in figure 4.26.

To the above, additional intermediate examples could be developed, but they would not show more interesting features regarding key constraints. Note that some possible permutations has been omitted in the SQL-examples below to increase the understandability.

```
relationA(attribute1, attribute2, ..., attributeN)
```

Figure 4.18: A single relation/table

```
relationA( keyattribute1, keyattribute2, ..., keyattributeN, attribute1, attribute2, ..., attributeN )
```

Figure 4.19: A single relation with a primary key and additional attributes

46
relationA( keyattribute1, keyattribute2, ..., keyattributeM )

definition: A relation (relationB) with a foreign key that references another relation (relationA)

relationA( keyattribute1, keyattribute2, ..., keyattributeN, foreignkeyattribute1, foreignkeyattribute2, ..., foreignkeyattributeM )

definition: Two relations, each with a primary key, which references each other through foreign keys.

CREATE TABLE relationA(
  attribute1  INTEGER,
  attribute2  INTEGER,
  ...,        ...
  attributeN  INTEGER
)

definition: SQL-code that creates a single table
Chapter 4. Implementation

```
CREATE TABLE relationA(
    keyattribute1 INTEGER,
    keyattribute2 INTEGER,
    ...
    keyattributeN INTEGER,
    attribute1    INTEGER,
    attribute2    INTEGER,
    ...
    attributeN    INTEGER,
    PRIMARY KEY (keyattribute1, keyattribute2, ..., keyattributeN)
)
```

Figure 4.23: SQL-code that creates a single table with a primary key and additional attributes

```
CREATE TABLE relationA(
    keyattribute1 INTEGER,
    keyattribute2 INTEGER,
    ...
    keyattributeN INTEGER,
    PRIMARY KEY (keyattribute1, keyattribute2, ..., keyattributeN)
)
```

```
CREATE TABLE relationB(
    keyattribute1 INTEGER,
    keyattribute2 INTEGER,
    ...
    keyattributeN INTEGER,
    foreignkeyattribute1 INTEGER,
    foreignkeyattribute2 INTEGER,
    ...
    foreignkeyattributeN INTEGER,
    PRIMARY KEY (keyattribute1, keyattribute2, ..., keyattributeN),
    FOREIGN KEY (foreignkeyattribute1, foreignkeyattribute2, ..., foreignkeyattributeN)
        REFERENCES relationA(keyattribute1, keyattribute2, ..., keyattributeN)
)
```

Figure 4.24: SQL-code that creates a relation (relationA) with a foreign key that references another relation (relationB)
Figure 4.25: SQL-code that creates two relations, each with a primary key, which references each other through foreign keys.
CREATE TABLE relationA(
    keyattribute1 INTEGER,
    keyattribute2 INTEGER,
    ...
    keyattributeN INTEGER,
    foreignkeyattribute1 INTEGER,
    foreignkeyattribute2 INTEGER,
    ...
    foreignkeyattributeN INTEGER
)

CREATE TABLE relationB(
    keyattribute1 INTEGER,
    keyattribute2 INTEGER,
    ...
    keyattributeN INTEGER,
    foreignkeyattribute1 INTEGER,
    foreignkeyattribute2 INTEGER,
    ...
    foreignkeyattributeN INTEGER
)

ALTER TABLE relationA ADD CONSTRAINT pk_attributes_a
    PRIMARY KEY (keyattribute1, keyattribute2, ..., keyattributeN)

ALTER TABLE relationB ADD CONSTRAINT pk_attributes_b
    PRIMARY KEY (keyattribute1, keyattribute2, ..., keyattributeN),

ALTER TABLE relationA ADD CONSTRAINT fk_attributes_a
    FOREIGN KEY (foreignkeyattribute1, foreignkeyattribute2, ..., foreignkeyattributeN)
    REFERENCES relationB(keyattribute1, keyattribute2, ..., keyattributeN)

ALTER TABLE relationB ADD CONSTRAINT fk_attributes_b
    FOREIGN KEY (foreignkeyattribute1, foreignkeyattribute2, ..., foreignkeyattributeN)
    REFERENCES relationA(keyattribute1, keyattribute2, ..., keyattributeN)

Figure 4.26: A variant of figure 4.25 in which the SQL-code is generated in steps.
4.3 A general transformation model

In this section a general model that can describe the transformation between different types of models is presented. The basic idea is to produce a rule based system that hierarchically builds the result, i.e. the text. In the first subsection (4.3.1) some concepts are introduced and defined according to this context.

4.3.1 Resulting text

Although the transformation should be able to function without having to modify either the source meta-model or the data within it, it must store its result somewhere. Therefore, each transformation will result in a character string, a textblock, with varying size. Each textblock is either the result of a transformation of a source model object, or the concatenation of several other textblocks. This is shown in figure 4.27. In this figure each box represents

![Diagram](image)

**Figure 4.27:** How the resulting text is built up

a textblock. At the lowest level the information comes from atomic textblocks. Usually
the word 'atomic' is used in the sense 'undivisible', which a string of characters clearly is not. However, if the aim is the actual semantic value of the string, the word atomic is appropriate. The origin of atomic strings can be either:

- the source model of the transformation

- metadata from the transformation

- some external source

These are further explained below:

**Source model** This is the actual model that the transformation rules collect information from. Using the concepts of UML and MOF this would mean that it can be essentially one of two things:

- A property of an object, i.e. a variable value or the result of a method on the object (the latter will not be examined in this dissertation).

- An association, which basically relates to the objects at either side of the association (or values of them as explained in the point above).

The atomic text may also be converted between representation forms, e.g. decimal to hexadecimal. These conversion routines could perhaps be specified in OCL, but this is not investigated further here.

**Transformation metadata** This is information that is derived from the transformation process, e.g. the size of a textstring or the number of rules that apply to a special class.

**External information** This is information primarily of two different variants:

- Static – this is text that binds other values together. For example, in generating SQL code (as shown in figure 4.22) there is text that is the same in all statements,
Chapter 4. Implementation

eg. 'CREATE TABLE', '{' and 'INTEGER'. In this application, atomic should be interpreted as 'not useful to divide', i.e. while 'CREATE TABLE' could be broken into 'CREATE ' and 'TABLE' this would only mean that one extra rule would have to be added.

- Dynamic – this is text that stems from some variable source, totally external to the information in the repository. While it probably is desirable to gather all information in one place (i.e. the repository), the nature of today’s computing environments will mean that some information will only be available from some external source. Additionally, some information can be created without using the repository contents, e.g. date and time strings and random numbers.

From the discussion above a sub-part of the transformation model can be created, as shown in figure 4.28.

![Textblock class](image)

Figure 4.28: The Textblock class

4.3.2 Update approach

A question that arises is when and how models are transformed. This is how changes are detected and handled. In the discussion below four different alternatives have been isolated:

- Continuous transformation

- On demand transformation
Chapter 4. Implementation

- Full transformation
- Partial transformation

These are explained below.

**Continuous transformation:** the transformation is performed continuously. This might be a good way of hiding the actual implementation details of the transformation from the user, perhaps through some declarative language. However, as the models involved grow in size and complexity, so also does the number of transformations that need to be performed. Still, hiding implementation details from the user is an important feature.

**On demand transformation:** the transformation is done when the data is needed. As the result of a transformation will be information in textual form, this could be generated when needed. As an example from the database domain; if a number of relations is transformed to SQL-code, the transformation could be done at the moment the SQL-code (the resulting textblock that describes the whole database) is accessed. This will insure that the data in textual form corresponds to the data in the model. However, if we want to continuously create a database from a model in the repository, this would mean that the textual form (textblock) would need to be polled periodically. Thus the behaviour of the first method described above is recreated.

**Full transformation:** a total transformation of the whole model is triggered as a result of an update event on an object. This ensures that a transformation is only done when necessary, and as the change event is propagated through the whole model, it can be detected and handled as required. As proposed in the example above, the update of the SQL-code could trigger a reconstruction of the database. But as a single update to an object will trigger all transformations in the whole model (because we do not know which values the transformations depends on), this results in large (and possibly lengthy) computations.
**Partial transformation:** a transformation of a changed object or submodel is initiated as a result of an update event on an object. Only those textblock objects that depend on the changed object are re-transformed. This requires that changed objects can be distinguished and separated from unchanged objects, and that subtransformation can be applied upon the changed objects. It also requires that dependencies between data sources (in the source model) and subtransformation result parts are known.

While all approaches may have their advantages, the latter method (partial) is chosen for this project. It can, however, be shown that all other update methods above can be emulated:

- Full transformation can be emulated by running partial transformation as if all objects have changed prior to the transformation.
- Continuous transformation can be emulated by iterating full transformation.
- On demand transformation can be emulated by running one full transformation when needed.

Using partial transformation means that a minimum\(^1\) of computations will be performed during transformations, while still making sure that the result is consistent with the repository contents. The task of writing rules will also be simplified as the problem can be divided into smaller units, each solved separately.

A disadvantage of this method is that it cannot handle circular dependencies, e.g. if object A is transformed into object B which is used as a source by object A, this will trigger an endless loop of transformations. It is impossible to create such constructs using standard text strings anyway, so this is not a problem that will affect the result of any transformation. However, the tool (or repository) that handles the rules must make sure that this situation cannot occur, either by preventing them from being inserted into the ruleset, or by checking

---

\(^1\)With regard to the methods presented here.
Chapter 4. Implementation

as they are executed.

While the above by no means is a formal proof of the advantage of this method, the
task of choosing the optimal one is out of the scope of this dissertation. The requirements
for OCL that come from choosing this update approach will be presented below in section
4.4.

4.3.3 Events

As mentioned above, using partial transformation means that it is neccesary to detect
changes to the source model and handle these according to some chosen method. The event
types that result from alteration of the source model are:

- Creation of objects – This means that a new instance of a class is added to the source
  model. Information in the source model object should be elicited according to some
  rule associated with it.

- Updates on objects – Update events are triggered when one or more attributes of an
  object are changed, and thus make it neccesary to propagate the new information to
textblock objects that depend on this.

- Deletion of objects – As a source object is removed then either the corresponding
  transformation object is removed or any values used in an aggregating transformation
  object are removed.

An event handler can therefore be specified as follows:

Class: <sourceclass specified as an OCL-context>
Type: <insert | update | delete>
Further, it would be advantageous if an event handler could be specified to match more than one event type, e.g., a rule could be valid for both inserts and updates. This is because the handling of a class is often identical for inserts and updates. Thus, slightly modified, the syntax for an event handler looks like this:

Class:  <sourceclass specified as an UCL-context>
Type:  <commalist consisting of 'insert', 'update' and 'delete'>

An event specified for a certain class will only trigger on changes made to instances of this class, and then only changes as specified by the event specification type.

It would be useful if different events could be specified for different object values. The event specification can be extended with a condition part as follows:

Condition:  <OCL-expression>

An empty condition should always evaluate to true. Furthermore, the event usually leads to some modification of the state\(^2\), and therefore an action should be executed if the condition evaluates to true.

Declaration:  <declaration of objects>
Action:  <UCL-expression>

In the declaration part types for new objects can be given, and new objects can then actually be created by a create() construct. As their information source objects are deleted, any associated textblock object should be removed and thus a delete() should be called on it as shown in the example below.

Some simple examples of how event specifications works (cf. figure 4.17):

Class:  Unique
Type:  insert

\(^2\)It should be noted that modifying the state of the system breaks one of the basic features of OCL
Chapter 4. Implementation

Condition:  PrimaryKey = true
Declaration: Textblock T
Action:     T.create(); T.Update:=TRUE

This will make the event specification valid for all instances of the class 'Unique' where PrimaryKey is true\(^3\) and thus create a textblock object named T for every created Unique object, each with the boolean flag Update set to true. A deletion event would be specified as:

Class:       Textblock
Type:        update, delete
Condition:   Update = true AND
              UsesInfoFromUnique->empty
Declaration:
Action:      self.delete();

Each insert or update event should result in a transformation of a source model object to a value of a textblock object. This can be implemented in the action expression by navigating to the source object and then specifying a variable to collect information from.

Class:       Relation
Type:        update
Condition:   RelationProvidesInfoFor->notEmpty
Declaration:
Action:      RelationProvidesInfoFor.Update:=true

Each update event made to the class 'Relation' in the source model will be propagated to each corresponding textblock object.

\(^3\)This could be written as simply 'PrimaryKey'
Chapter 4. Implementation

While event specification does not add any new features to the OCL language, the dynamic creation and deletion of objects does, and will be further discussed in section 4.4.1.

4.3.4 Transformation model

Each text string, atomic or composite should be represented as an object. These objects corresponds to each box found in figure 4.27. Each object should specify the resulting text as a string, but should also specify the dependencies on other objects from which it is built. The meta-model for storing this data is presented in figure 4.29.

![Diagram of transformation model]

Figure 4.29: A general transformation model

59
Chapter 4. Implementation

All objects are instances of ModelElement. This includes objects of classes not described by this metamodel. However, to be able to make associations from this meta-model to objects stored in other meta-models, the overall repository metamodel must be extended. This is further described in Gustavsson and Lings (2001).

This extension makes it possible to create associations between a textblock-object (in the transformation model) and the source object, thus enabling OCL navigation from a textblock to its source object.

One assumption that is made in the general transformation model and the action expressions is that associations are reflexive. I.e. if a collection of associations, A-uses-B, is expanded with a new association this means that the collection of associations in the reverse order, B-is-used-by-A, is expanded as well. This insures that integrity is preserved, in contrast to if reflexivity was handled by the user.

The transformation works as follows: when a ModelElement is either created (inserted), updated (modified) or deleted (removed), one or more rules will be executed upon the object.

Patterns

A feature of the transformation model is patterns. A pattern is generalized solution that can be implemented and applied in a problem situation, and thereby eliminate one or more of the inherent problems (Eriksson and Penker 2000). A rule could use its condition to check for patterns associated with the ModelElement which caused the event. The pattern thus decides which rules should be executed. These patterns are setup at the time the ModelElement is created (or modified), either by an automatic routine that recognise the type of the object, or by the user.

The existence of an association between a ModelElement object and a Pattern object can be checked as part of the condition. Thus no specific implementation of this part is needed, except for the possible creation of these associations, but they should be handled
by the tool that creates the ModelElement object. One usage of patterns is when the source model is missing vital information needed to complete the entire transformation. By asking the user this extra information could be gathered and stored as patterns, and then utilised at the time of transformation. The advantage of this method is that the existing storage model does not have to be altered, something which might be impossible or difficult if it has been designed and built by some other part, eg. a company.

Rules

Rules could be divided into two groups: those which run in the context of source model classes, and those which run on the textblock class. The former group is responsible for propagating the actual event to the corresponding textblock. As the textblock object is modified it will cause another update event, which in turn will trigger a rule that executes in the context of the textblock class. Information is then collected by action expressions which navigate from a specific textblock object to one or several other source objects. These source objects can either be atomic values from objects in the source model or other textblocks. It may be necessary to convert information from atomic source model objects into a textual representation before they are stored in the textblock.

Each textblock can be placed into a group by looking at the attribute Type. It can be set by the rule that creates the textblock, and then used in the condition part of rules. This is not necessary as in most cases the same values that are used by a rule to determine the type can be used by another rule at a later time to make the same decision. It, however, makes the rules easier to read and understand as the rule designer can explicitly group textblock objects.

Rulesets

Each rule belongs to a ruleset which covers a certain transformation or domain. Since a Textblock is a ModelElement, it too can be transformed by a rule and used to generate a
Chapter 4. Implementation

Textblock object, thus creating the hierarchical structure of figure 4.27.

The association between a rule and the textblock object it creates is not necessary in all cases, but can be used by the rule implementor to trace a textblock to a certain rule.

The order of the rules is important and should be preserved. This is because one rule may be dependent on the result of another rule.

Includes/IsIncludedBy

The Includes/IsIncludedBy association on the textblock object could be used (but is not in the examples provided) to make rules in a large rulebase less complicated. If set up correctly, a text object which is updated could follow isIncludedBy and set Update in all objects. Thus a rule does not have specific knowledge about which textblocks should be updated. However, this requires that the association be created at the time of creation of a textblock object, which in turn requires knowledge about the model and also results in more code. Includes/IsIncludedBy is also necessary if a source model object is used in more than one transformation. This is because if a rule navigates from a textblock object T to a source model object, e.g. a Relation object, and this object has been used in two different transformations which have created two textblock objects, T1 and T2, the rule cannot decide which textblock object to use. This can be solved by setting up associations at create time from the first textblock object T to either T1 or T2. Another way to solve this is to create an association from the rule that created the textblock through the role IsGenerated/Generates, and then use this association in the condition of the navigation.

4.3.5 Example transformations

In this section some examples of transformations are given. The examples are tied to the example relational meta-model (figure 4.17) and its accompanying SQL-code examples presented in section 4.2.
**Code conventions**  In the following examples of rule execution some conventions are used
to describe the state of the system and how different actions are performed to change it.
First, the current state of the system is described by this construct:

```
-- Rulebase --
-- Objectstate --
-- Eventqueue --
-- Action / Rule triggering --
-- External events --
```

In the `rulebase` section all rules are specified who are used in the example.

The `objectstate` section lists all objects and their variable values. To avoid creating large
text-masses by repeating certain entries, e.g. rules; if no values are changed then only the
objecttype and objectidentifier is stated. Textstrings that contains linefeeds are wrapped
on that location. Since the executionlog uses indentation to increase readability, this means
that a string wrapped by a linefeed may restart at a columnposition greater than 1. That
means that a linefeed broken string restarts at the same horizontal position as the indented
string started at. For example, if the code looks like this:

```
  Text:...........
  Year:CHAR(4),
```

then the real value of Text is:

```
  Year:CHAR(4),
```

(including a line feed)

The `eventqueue` is the queue of events that currently exist in the system. If objects are
added, changed or removed this will create new events which are added at the bottom of the
queue, thereby creating a first-in-first-out behaviour. Three different eventtypes are used:
insert, update and delete. Each of these has slightly different information associated with
them:
Chapter 4. Implementation

- `<oid>` insert

  The object `<oid>` has been inserted.

- `<oid>` update `<value>` `<attribute>`

  The attribute `<attribute>` in object `<oid>` has been assigned the value `<value>`.

- `<oid>` delete

  The object `<oid>` has been deleted.

- add association `<oid>`. `<rolename>`={ `<object1>`, `<object2>`, ..., `<objectN>` }

  Specifies how an associations is created from the object `<oid>` through the role `<rolename>` to the objects `<object1>` to `<objectN>`. For example:

  add association Rel1.ChildAttribute={Attr1,Attr2,Attr3}
  add association Attr1.ParentRelation={Rel1}
  add association Attr2.ParentRelation={Rel1}
  add association Attr3.ParentRelation={Rel1}

  Note that the construct only specifies the associations which is added. Thus the set of new object references should be unioned with the existing set in the association.

  There is no corresponding construct for removing associations (although there could be one).

In the Action / Rule triggering section one or several rules that matches the contextclass, eventtype and condition are stated. The next objectstate section will depict changes done by the executed rules (ie. executed actionexpressions inside the rules).

execute `<rule>` on `<name>` [yields `<new object>`]

The rule `<rule>` is executed on the object `<name>`, optionally creating `<new object>`.

External events shows how the repository is changed by the user or the tool. In this section the following metacommands are used to depict changes to the state that are performed by other sources than the rulesystem, ie. a tool. This section is not used in each execution cycle as rule may trigger other rules, and thus external events does not interact with the repository.
Chapter 4. Implementation

- **insert** `<name>::<type>`
  
The object `<name>` of the type/class `<type>` is inserted into the repository.

- **update** `<name>.<attribute>::<value>`
  
The attribute `<attribute>` in the object with the name `<name>` is assigned the value `<value>`.

- **delete** `<name>`
  
The object `<name>` is removed from the repository.

In addition to this the hash character (`'#'`) is used to depict comments added to the executionlog.

To exemplify how rules are written, some logs of executions is shown below.

**A simple ruleschema handling an insert**

The first example (see appendix A) describes how an object (class instance) is inserted into the repository. In figure 4.30, a UML object diagram, the final object state is shown.

![UML object diagram](image)

Figure 4.30: An example of how the general transformation model works; an object is inserted into the `Relation` class

The following is a detailed description of what happens in the executionlog.

1. The system starts empty, with no objects, no associations, no pending events and thus no execution of rules.
2. Next, two rules are inserted into the repository. These makes up the rulebase that are used in the following execution.

3. Two events are created as a result of the insertions of the rules.

4. Since no rules match the contextclass and eventtype, no action is taken.

5. The relation object is inserted into the repository.

6. This creates an insert event that is appended to the eventqueue.

7. Since Rule1 matches the context class and the eventtype, and the condition is empty (and thus is evaluated to true), the actionexpression of Rule1 is executed. This will create the object T1, as specified in the declaration of the rule, and associate it with the Rel1 object through the association RelationProvidesInfoFor. Finally the boolean flag Update in the the textblock object T1 is set to true.

8. The insertion of the textblock object T1 will create a new insert event.

9. Rule1 does not match this event (wrong contextclass) but Rule2 will, and thus it is executed. The rule will concatenate together both static text and text from the source object Rel1. It uses the association UsesInfoFromRelation to get the attribute Name, and the value inside it.

The ruleset would thus initialises the attribute Text of the textblock object T1 to the following:

```sql
CREATE TABLE Artist ( )
```

This is not correct SQL-code since it does not contain any attribute definitions. However, this can be avoided by extend the condition to not include relations which does not have any attributes.
A simple ruleschema handling an update

The next example (see appendix B) continues by showing how a value in a source model object is changed. The update event is propagated to the textblock object which then updates itself.

The following is a detailed description of what happens in the execution log:

1. The system starts where it left in the previous example, with the exception of the rules. Rule2 is slightly modified to handle update events too, and is thus renamed Rule2b. Also, a new rule, Rule3, is added to the ruleset.

2. The attribute **Name** is assigned a new value 'Group' which replaces the old 'Artist'.

3. This creates an update event that is added to the eventqueue.

4. Rule3 matches the event and executes. Since Rel1 can be used as an information source for several different textblock objects, the rule will follow all associations from Rel1 to textblock objects and change the attribute **Update** to TRUE.

5. This will create a new update event on the textblock object T1.

6. Rule2b (which now also matches for update events) executes and rebuilds the attribute **Text** in the object T1 by eliciting information from the Rel1 object.

A simple ruleschema handling an delete

Appendix A is an example of what happens if an object in the sourcemodel is removed. Just like in the previous example the event is propagated to the textblock object(s) that needs it. Note that while the removal of an object creates a delete event, the event is propagated by changing an attribute of the textblock object(s), thus creating an update event.

1. The execution continues where the previous left off, once again with new rules added to the system.
2. The Rel1 object is deleted from the repository. The object will still exist for an
   eventhandler, but is invisible to users of the repository.

3. This creates a delete event.

4. Rule4 is executed as a result to the delete event, and it will follow all associations
   from Rel1 to textblock objects and change the attribute **Update** to TRUE. The Rel1
   object is finally fully removed from the repository after the execution of the rule.

5. An update event is created as a result of the execution of Rule4

6. Since the collection of associations through the role UsesInfoFromRelation from the
   object T1 is empty, Rule5 is executed. Since no association to a Relation object
   exists this means that the datasource for the textblock object T1 is gone, thus it
   should remove it self as a consequence.

7. The delete event created as a result if the removal of the T1 does not match any rules.

### 4.4 Using OCL for transformations

In this section we investigate whether OCL is sufficient for transforming a model to text, or
whether extensions (section 4.4.1) are neccessary. In section 4.4.2 some errors in the OCL
standard are discussed.

#### 4.4.1 Extensions of OCL

Some requirements are solved by the existing contracts and features of OCL, for example the
basic fact that OCL must be able to handle text at all. This is solved by the **String** datapatype,
which allows variables to be declared which can hold and represent ASCII textstrings of
arbitrary lengths. Furthermore, OCL has datatypes for integers, realnummers and boolean
values, which can be useful for temporary store descisions made during the transformation.
The resulting text can be seen as a number of strings that are to be put together in a certain order. OCL must thus be able to concatenate textstrings. This is done using the *concat* operator (property) in OCL.

\[
\text{string}.\text{concat(string2 : String) : String}
\]

To be able to create any string consisting of any characters, eg. linefeeds, escape and control characters, or even binary strings, it must be possible to create these. The OCL language has constructs for creating arbitrary strings, even binary strings. It is for example possible to specify a linefeed by "\n".

An important feature that is missing in OCL is the conversion of values of other types than string, ie. integer, real or bool, to string. Otherwise they cannot be concatenated with strings, and can thus not be used in the resulting text. There exists a limited form of casting by which objects can be re-typed, however an object can only be cast into a supertype of itself.

A solution to this problem would be to add an *asString* property to the type *OclAny*. This is the supertype of all other types, and thus any type can be converted to strings as shown in this example here:

\[
\text{stringAttribute}.\text{concatenate(integerAttribute.asString)}
\]

Something which is missing from the OCL language is an operator for finding the start position of one string within another. This operator is know as eg. *Instr* or *Pos* in other languages. It is likely that some information is stored in structured strings, eg. "<firstname>,<lastname>"", and thus the information must be parsed out. Assume that the operator *Instr* takes two parameters, the position to start the search at and the string to search for. Then it could be used as in this example:

```
Name:='John Smith';
Lastname:=Name.substring(Name.InStr(0," ")+1,Name.size)
```

The variable 'Lastname' would thus contain the value 'Smith'. This operator could also
be used for checking whether a certain value or string is presented in an already generated string.

Modifying the state of the repository, i.e. adding or removing objects, or changing attributes of objects requires extensions to the OCL language (Gustavsson and Lings 2001). Starting with objects, two new constructs were introduced in section 4.3.3, namely create() and delete(). Without these no objects can be added or removed and thus the transformation rules cannot work dynamically. The alternative would be that the user (or tool) adds and removes objects as necessary.

Next, the rules must be able to modify attributes of an existing object at runtime (e.g. updating the resulting text of a transformation). In the examples in section 4.3.3 the assignment operator is introduced, which is used in two distinct ways. The first is assigning a single value to an attribute of an object:

\[ \text{T.Update} := \text{TRUE} \]

The second is the more unusual behavior of assigning a collection to a role of an object:

\[ \text{UsesInfoFromRelation} := \text{UsesInfoFromRelation} \cup \text{Union(Rel1)} \]

This example adds the object Rel1 to the collection of objects which is related to this object through the association UsesInfoFromRelation. For more information on these constructs see Gustavsson and Lings (2001).

In addition to the self alias, which refers to the contextual object, the alias thisrule could be used to refer to the actual rule object that executes. This in order to be able to associate a textblock object with a rule through the role IsGenerates/Generates.

In many cases there is a need to detect cyclic dependencies to an object. That is, through some chain of associations, an object references itself. The only way of detecting a cyclic reference is to follow it and see if the association chain points back to the root object. As a solution to this problem a new operator/method is proposed:
collection->selectClosure(
    expr-which-returns-a-collection : OclExpression) : collection

selectClosure will calculate the transitive closure from the given collection according to the following algorithm. By evaluating expr-which-returns-a-collection a collection is obtained. This collection is then used for a new evaluation which result is unioned with the previous result. This is repeated until no more elements/objects are returned, i.e. the new collection is equal to the previous. Some examples are illustrated with the use of figure 4.31.

![Diagram of objects and their references]

Figure 4.31: Objects which references other objects

{A}→selectClosure will return {B,C,D,E,F}
{D}→selectClosure will return {F}
{F}→selectClosure will return {}
{G}→selectClosure will return {H,I,J,K,L,G}

In the last example a cyclic dependency exists since G is a member of the resulting set. This can be detected using OCL, as shown below:

    object->selectClosure(navigate.through.roles)->includes(object)

If the above returns boolean true, a cyclic dependency exist from object to itself.

4.4.2 Ordering

OCL has constructs for specifying ordered lists through its Sequence type, which is a subtype of Collection:
Sequence { 1, 435, 3, 7 }
Sequence { 'Agnus', 'Denise', 'Paula' }

Through the select and reject operands collections can be created, and thus also a sequence.

Ordering is for example needed if the resulting text depends on a certain order of the elements, as in the example meta-model presented in figure 4.17. The actual order of fields (attributes) is not technically important as both the relational model and real world implementations of it will accept any order of fields when doing, for example, queries. However, the visual appearance is much dependent on the order of fields, and the user might be very confused if it changes arbitrarily. Therefore, if fields are to be created (as shown in eg. figure 2.12), they must be sorted according to some attribute. In the case of the example relational model the attribute Order can be used to accomplish this.

Even if objects (or associations to objects) may be ordered (by a tool) in a certain way, it may be the case that the transformation domain has two totally different sorting requirements. It would be impossible to physically order in two ways.

The Collection type has a property called sortedBy which could be used to order collections of values. However, standard (UML 2000) contradicts itself by first specifying sortedBy as:

```
collection->sortedBy(expr : OclExpression) : Boolean
```

and then stating that it "results in a Sequence containing all elements of collection". Clearly the boolean result type is erroneous and should be replaced with "Sequence(T)". The OCL standard is known to contain some errors, as pointed out by Finger (2000). This is, however, not investigated further in this dissertation.

### 4.5 Building SQL using ALTER TABLE

In appendix D a ruleset and an execution example are provided which show the transformation of a relational model to a textual representation utilising SQL statements. This is
done using ALTER TABLE statements for creating foreign key constraints. A description of the rules and their execution is presented below.

These examples insert objects in an order which is advantageous to the rulesystem to make the rules execute in the 'right' order. However, it does not really matter in which order objects are added. This is because each rule and the corresponding textblock must be able to update itself if a value in the source model is changed. As this would generate more events and execute more rules this has been avoided to make the execution shorter and simpler to read.

The conditions in the rules only check for partial consistency. For example, the rules prevent a foreign key without attributes, but do not prevent the number or types of attributes differing between the foreign key and the primary key it refers too. In reality the rules could be extended to guard this kind of integrity, but an even better solution would be to incorporate constraints into the source model.

In figure 4.32 the final objectstate is shown as an UML object diagram. The dashed lines mark a connection through the general transformation model.

![UML diagram](image)

Figure 4.32: An UML object diagram that describes the final state of the example execution in appendix D

73
Chapter 4. Implementation

The ruleset presented in appendix D can be divided into source model rules (see D.1.1), ie. rules which context is one of the classes in the source model, and textblock rules (see D.1.2), ie. rules that executes in the context of the textblock class. The first group is described in section 4.5.1 and the latter in section 4.5.2.

4.5.1 Description of source model rules

In section D.1.1 a number of rules are presented. All these rules operates in the context of a source model class. Since all these rules are similar in style, only three will be discussed here. The first rule handles inserts in the Relation class.

```
Rule_Relation_Ins:Rule
    Context    Relation
    Type       insert
    Condition
    Declaration Textblock T
    Action     T.create() ;
                T.UsesInfoFromRelation:=self ;
                T.Update:=TRUE
```

When a new object is added to the Relation class it will generate an insert event, and as a consequence, the rule Rule_Relation_Ins will execute. This rule will create a new object T, an instance of the class Textblock as specified in the declaration. It will then add a reference of it self (the Relation object) to the association/role UsesInfoFromRelation. As the object is new, the association is empty until then. The last thing the rule does is to change the value of the textblock attribute Update to boolean true. This is done to signal that an update is needed on that particular object, ie. it must be regenerated from the source model. As the textblock object is created it will create a new insert event, and thus the next rule (Rule_Relation_Upd) will be triggered automatically.

Update events are handled in a similar way as shown in the rule below.

```
# Handles updates to Relation objects.
Rule_Relation_Upd:Rule
```
Chapter 4. Implementation

Context Relation
Type update
Condition RelationProvidesInfoFor->notEmpty
Declaration
Action RelationProvidesInfoFor->forAll( T : Textblock | T.Update:=TRUE )

When a change occur on a Relation object (ie. the name changes) this is propagated to the corresponding textblock object by setting the attribute to true. As it may be case (although not in this example) that the relation object is used as a source for many different textblock objects, the Update variable is changed for all objects the rule finds by navigating the role RelationProvidesInfoFor.

Because the T textblock object is modified it will generate an update event which will be handled by another rule later on. This will also prepare it for passing the condition evaluation of that rule.

When a Relation object is deleted this will generate a delete event, which is handled by the following rule:

# Handles deletes to Relation objects.
Rule_Relation_Del:Rule
Context Relation
Type delete
Condition RelationProvidesInfoFor->notEmpty
Declaration
Action RelationProvidesInfoFor->forAll( T : Textblock | T.delete() );
RelationProvidesInfoFor:=[] ;
-- Propagate the change to the Model textblock.
PartOfModel.ModelProvidesInfoFor.Update:=TRUE ;

A textblock object has no use without its information source. Therefore the role RelationProvidesInfoFor is navigated and all textblock objects 'in the other end' is removed. After this the association RelationProvidesInfoFor is cleared to insure that no textblock tries to navigate to the now non existing source model object. Finally, all objects that depends on the textblock object (or the source model object) must be made attentive on the fact that

75
information has been removed. Thus the rule navigates upwards in the hierarchy and sets
the Update flag on the textblock object belonging to the same model as this object did.

This concludes the explanation of the source model rules.

4.5.2 Description of textblock rules

Unlike the source model there is not as much duplication of rules in the textblock context.
Since a textblock should be built in the same way regardless of it is new or old, a rule can
be valid for both insert and update events simultaneous. Delete events are handled directly
in the source model rules, thus there are no need for delete rules in the textblock rules.

Textblock rules can be divided into three groups according to their relationship with
other rules/textblocks:

- Rules that only read information from the source model and not from other textblock
  objects.
- Rules that reads information from other textblocks and which result are read by other
textblock rules.
- Rules that only reads information from other textblocks.

It is of course possible that rules in the latter two groups also read information from the
source model. Below rules from these three groups are shown.

Rules that only reads information from the source model

Below is a rule that builds a text string suitable for a SQL table definition. It does this from
information in the source model regarding attributes in the relational model. The resulting
textblock is then read by further rules (see next section).

# Handles fields of string type.
Rule_TB_StringField:Rule
   Context Textblock
**Chapter 4. Implementation**

**Type**
- `insert`, `update`

**Condition**
- `UsesInfoFromAttribute->notEmpty AND UsesInfoFromAttribute.Length>0 AND Update=TRUE`

**Declaration**

**Action**
```
Text:=''
Text:=Text.concat(UsesInfoFromAttribute.Name);
Text:=Text.concat(' ');
Text:=Text.concat(UsesInfoFromAttribute.UsesType.Name);
Text:=Text.concat('(');
Text:=Text.concat(UsesInfoFromAttribute.Length.asString);
Text:=Text.concat('),
RelationProvidesInfoFor.Update:=TRUE;
Update:=FALSE
```

The condition of the rule make sure that the following statements are valid:

- The rule has an association to an attribute object.

- The variable `Length` of the `(object)` `Attribute` has a value greater than zero. This is important since this rule only handles attribute of type string. Another way of checking this would be to follow the association `UsesType` (see figure 4.17) and make a condition on the value of `Name` on the object `Type`.

- Finally, `Update` must be true. This is not really necessary if events only are generated by other rules, thus implying that an update is inevitable.

If the context class, eventtype and condition matches, then the action expression is executed. It will (re)build the contents of the textblock by using either information from the source model (as the name of the relational attribute) or static information stated in the rule (as `' '`). When the textblock has been updated with the new contents it must propagate the changes 'upwards' in the hierarchy. This is done by navigating to the source model, over the relation that this attribute belongs to, and then back to the textblock that results from
that relation. By setting the Update flag an update event is created that will will trigger another rule (ie. the rule described in the next section).

**Rules that read other textblocks and are read by other rules**

The following rule builds a SQL table definition from attribute and unique (≈primary key) textblocks. It also utilise information from the source model (by eliciting the name of relation).

```plaintext
# Handles valid relations.
Rule_TB_Relation:Rule
  Context   Textblock
  Type      insert, update
  Condition UsesInfoFromRelation->notEmpty AND
             UsesInfoFromRelation.ChildAttribute->notEmpty AND
             Update=TRUE
  Declaration Textblock Waist
  Action     -- Create and initialize temporary object.
             Waist.create() : Waist.Text:='';
             -- Collect and build the attribute part
             UsesInfoFromRelation.ChildAttribute.
             AttributeProvidesInforFor->forAll(T : Textblock |
             Waist.Text:=Waist.Text.concat(T.Text) ) ;
             -- Collect and build together Unique indices.
             UsesInfoFromRelation.ToUnique.
             UniqueProvidesInforFor->forAll(T : Textblock |
             Waist.Text:=Waist.Text.concat(T.Text) ) ;
             -- Put all parts together
             Text:='CREATE TABLE ' ;
             Text:=Text.concat(UsesInfoFromRelation.Name) ;
             Text:=Text.concat(' (\n') ;
             -- Add the waist part, except the last ",\n"
             Text:=Text.concat( 
             Waist.Text.substr(1,Waist.Text.size-2) ;
             Text:=Text.concat(' \n') ;
             -- Propagate the change to the Model textblock.
             UsesInfoFromRelation.PartOfModel.
             ModelProvidesInforFor.Update:=TRUE
             -- Remove temporary objects.
             Waist.delete();
             -- Update complete.
```
Update:=FALSE;

It does this in a similar way to the previous rule, i.e. the condition makes sure the context is correct and that information enough exists from the source model. Just as an update to an attribute object was propagated here, an update to a relation object must be propagated to the overall textblock model object. This is done by navigating the model and modifying the update flag on the Model textblock object.

This rule creates a new object as a temporary storage facility and then deletes it. Actually, this is not needed in this case as the text could be concatenated directly to the context textblock object. But in some cases it may be necessary to use this method, for example if transformation metadata as the length of a certain string is to be stated in the resulting text. Then a textstring can be built and placed in temporary object, the length calculated and added to the result.

A textblock could also be broken up into several textblocks by using the association Includes/IsIncludedBy. The rule designer must then take care that a delete rule is added that removes those extra objects should the primary textblock object be deleted.

Rules that only reads information from other textblocks

The rules that resides at the top of the hierarchical structure are actually slightly simpler than those presented above. This since they only have to concatenate the result from the textblocks below, and do not have to propagate any changes upwards.

Rule TR Model: Rule

<table>
<thead>
<tr>
<th>Context</th>
<th>Textblock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>insert, update</td>
</tr>
<tr>
<td>Condition</td>
<td>UsesInfoFromModel-&gt;notEmpty AND Update=TRUE</td>
</tr>
<tr>
<td>Declaration</td>
<td>Text:='' ;</td>
</tr>
<tr>
<td>Action</td>
<td>UsesInfoFromModel.ConsistsOfRelations. RelationProvidesInfoFor-&gt;forAll(T : Textblock</td>
</tr>
</tbody>
</table>

79
Text:=Text.concat(’\n’) ;
-- Add foreignkey (ALTER TABLES) if they exist.
UsesInfoFromModel.ConsistsOfRelations.ToForeignKey.
ForeignKeyProvidesInfoFor->forAll( T : Textblock |
    Text:=Text.concat(T.Text) ;
    Text:=Text.concat(’\n’) ) ;
Update:=FALSE

This rule works in the same way as the two other rules presented above. As the textblocks it concatenates has been carefully prepared, no further processing is needed.

This completes the presentation of the textblock rules. The rules not presented here are either of a similair type (for handling integer attributes) or rules that prevents invalid information from being included in the result.

### 4.5.3 Example rule execution

In section D.2 an example execution log of the ruleset is presented. The system starts in D.2.1 with only two objects, namely the type objects which attributes associates with. Three attributes and a relation is then inserted. This will generate a chain of events and rule executions that finally produces a SQL table definition.

In D.2.2 a primary key is added to the relation and it is incorporated into the table definition.

In D.2.3 a new relation is added. This relation has two attributes, one which is a foreign key to the previous relation.

The final result of the transformation can then be found in the object T1 and is presented below in figure 4.33.

### 4.5.4 Validation of end result

The result of the example rule execution was validated through the interactive terminal included in the PostgreSQL database system (The PostgreSQL Global Development Group 2001). This was done by importing the final result of the transformation to the console.
CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4),
    PRIMARY KEY (RecId),
)

CREATE TABLE Sale (
    SaleId INTEGER,
    RecId INTEGER
)

ALTER TABLE Sale ADD CONSTRAINT fk_RecID_Record
    FOREIGN KEY (RecId) REFERENCES Record(RecId)

Figure 4.33: The final result from the transformation

The only change done was to place a terminator character (",";) at the end of each SQL statement as PostgreSQL requires this\(^4\). The output of PostgreSQL is provided in appendix E.

4.6 Building SQL using less ALTER TABLE

In appendix F an alternative ruleset is presented, capable of generating SQL that uses considerably less ALTER TABLE statements that the ruleset in appendix D.

This is done by detecting circular foreign key references and generate these with ALTER TABLE. A disadvantage of this method is that all foreign keys of a relation which is part of a cyclic reference chain will be generated with ALTER TABLE, even the ones that could have been generated as base table constraint definitions.

\(^4\)This can be avoided by making some trivial changes to the rules, but then the result would not conform to SQL-92.
A relational schema that contains no cyclic foreign key dependencies will be generated as SQL utilising no ALTER TABLE statements at all. Tables will be placed in the order necessary to create foreign key references.

Just as in the previous example, the ruleset presented in appendix F can be divided into source model rules (see F.1.2), ie. rules which context is one of the classes in the source model, and textblock rules (see F.1.2), ie. rules that executes in the context of the textblock class. The first group is described in section 4.6.1 and the latter in section 4.6.2.

4.6.1 Description of source model rules

The rules in this section functions mostly like the ones presented in section D.1.1 and then discussed in section 4.5.1. Therefore, only differences will be pointed out.

Rule_LessAlter_Model_Ins

Instead of creating just one textblock, it now creates three, each with a different purpose. The Live textblock holds a valid transformation result at all times, assuming that one single rule execution is atomic. The Temp textblock is cleared and then rebuilt as a consequence of a change in the model. The result is then copied to the Live textblock. Finally, the Alter textblock holds the textual representation of all foreign keys belonging to a relation that is part of a cyclic reference chain. These ALTER TABLE statements are concatenated to the end of the Live textblock after each transformation.

All rules

In most cases a change in a source model object will be propagated to the corresponding textblock object. The exception is the relation class, there changes propagates directly to the corresponding Model object.
4.6.2 Description of textblock rules

Compared to the previous ruleset, the textblock rules are both larger and more complex. In general, all textblock rules propagates a change to the Temp Model testblock. An exception to this the Relation textblock which builds the Temp and Alter Model textblocks instead. This since the whole model result must be rebuilt if something changes.

Rule_LessAlter_TB_Model.Temp

```
Rule_LessAlter_TB_Model.Temp:Rule
    Context Textblock
    Type insert, update
    Condition Update=TRUE AND
    Type='TEMP' AND
    UsesInfoFromModel->notEmpty AND
    UsesInfoFromModel.ModelProvidesInfoFor->
        select(Type='LIVE').Update=FALSE

Declaration
    Action -- Clear TEMP och ALTER textblocks.
    Text: '';
    UsesInfoFromModel.ModelProvidesInfoFor->
        select(Type='ALTER').Text='';
    -- Make all relations first rebuild themself and
    -- then the temporary textblock.
    UsesInfoFromModel.ConsistsOfRelations.
        RelationProvidesInfoFor->forAll(Update:=TRUE);
    -- Live textblock should update when
    -- transformation is finished.
    UsesInfoFromModel.ModelProvidesInfoFor->
        select(Type='LIVE').Update:=TRUE;
    -- Ok, we are done.
    Update:=FALSE
```

This rule will clear itself (TEMP) and the ALTER textblock and then set the Update flag on all relations. This will force them to rebuild themself, and to rebuild both the TEMP and ALTER textblock (as needed). This rule can not trigger again if not the previous transformation has finished. However, it should be possible to remove this condition since this rule clears all information that a transformation either creates or depends on.
Thus, each transformation starts from the same assumptions, regardless if the previous transformations was interrupted.

By setting the Update flag on the LIVE textblock the rule Rule_LessAlter_TB_Model_Live will trigger eventually.

**Rule_LessAlter_TB_Model_Live**

# Update the live textblock from the temporary  
# if no transformation is running.  
**Rule_LessAlter_TB_Model_Live:** Rule  
Context Textblock  
Type insert, update  
Condition Update=TRUE AND  
Type='LIVE' AND  
UsesInfoFromModel->notEmpty AND  
UsesInfoFromModel.ConsistsOfRelations.  
RelationProvidesInfoFor->forAll(Update=FALSE)  

Declaration  
Action Text:=UsesInfoFromModel.ModelProvidesInfoFor->  
select(Type='TEMP').Text ;  
Text:=Text.concat(UsesInfoFromModel.  
ModelProvidesInfoFor->select(Type='ALTER').Text ;  
-- Ux, we are done.  
Update:=FALSE

This rule will not trigger until the Update flag on all relations is false, ie. the transformation is done. It will then collect the text in the TEMP and ALTER textblocks and concatenate them. In this way the contents of the LIVE textblock is always valid and can be elicited by some external tool.

**Rule_LessAlter_TB_Attribute**

The textblock rules for handling attributes has been integrated into one rule, the differences was minor.
Rule_LessAlter_TB.Unique

Unique keys are now prefixed by the CONSTRAINT keyword and also includes the name of the constraint.

Rule_LessAlter_TB_ForeignKey

This rule utilises the (otherwise unused) Type variable to store some of the result, namely "ALTER TABLE <name-of-table> ADD". This is done to be able to use the foreign key textblock to two uses: either included in the table definition; or as an ALTER TABLE statement.

A cleaner and nicer way to accomplish this would be to store all information in the variable Text. However, this would require the use of the InStr operator (as proposed in section 4.4.1). This way the header could be removed by searching for " ADD " as shown below:

Text.substring(Text.InStr(0," ADD ")+6,Text.size-Text.InStr(0," ADD ")-6)

Rule_LessAlter_TB_NormalRelation

This rule builds complete CREATE TABLE statements, including foreign keys.

The differences in this rule as opposed to the previous ruleset can be found mainly in the condition. First of all, only relations for which all foreign key references exists should be generated. Figure 4.34 shows an example configuration of six relations which references eachother. Relation A and B does not reference any other relation and can therefore be generated in any order. As relation C references relation B, the latter must already exist if the creation of C should succeed. The condition of Rule LessAlter TB NormalRelation will make sure that only relations with all foreign key references resolved will be generated. D references both B and C, thus they must already exist.

However, if two relations references eachother, as E and F does, this can not be resolved
by delaying the creation of either of them. This situation will be handled by the next rule 
(Rule_LessAlter_TB_CyclicRelation).

The rule detects relation which contains cyclic foreign key references by introducing a 
new operator, selectClosure, as discussed in section 4.4.1.

```
Rule LessAlter TH NormalRelation:Rule
  Context Textblock
  Type insert, update
  Condition Update=TRUE AND
               UsesInfoFromRelation->notEmpty AND
               UsesInfoFromRelation.ChildAttribute->notEmpty AND
               UsesInfoFromRelation.ToForeignKey.ToUnique.
               UniqueIsPartOfRelation.RelationProvidesInfoFor->
               forall( T : Textblock | Update=FALSE ) AND
               UsesInfoFromRelation->selectClosure(ToForeignKey.
               ToUnique.UniqueIsPartOfRelation)->excludes(self)
  Declaration Text:='CREATE TABLE ' ;
               Text:=Text.concat(UsesInfoFromRelation.Name) ;
               Text:=Text.concat( '\n' ) ;
               -- Collect and build the attribute part
               UsesInfoFromRelation.ChildAttribute.
               AttributeProvidesInforFor->forall( T : Textblock |
               Text:=Text.concat(T.Text) ) ;
               -- Collect and concatenate Unique indices.
               UsesInfoFromRelation.ToUnique.
               UniqueProvidesInforFor->forall( T : Textblock |
               Text:=Text.concat(T.Text) ) ;
               -- Collect and concatenate foreignkeys.
               UsesInfoFromRelation.ToForeignKey.
               ForeignKeyProvidesInforFor->forall( T : Textblock |
```
Chapter 4. Implementation

```
Text:=Text.concat(T.Text);
Text:=Text.concat(',' );
-- Remove the last ",\n"
Text:=Text.concat(Text.substr(1,Text.size-2));
Text:=Text.concat('\n\n');
-- Add result to end of the temp Model textblock.
UsesInfoFromRelation.PartOfModel.
   ModelProvidesInfoFor->select(Type='TEMP').
   Text.concat(Text)
-- Update complete.
Update:=FALSE ;
```

**Rule_LessAlter_TB_CyclicRelation**

This rule transform any relation which contains cyclic foreign key references, ie. as table E and F does in figure 4.34. It does this by inverting the condition used in Rule_LessAlter_TB_NormalRelation, thus a relation is converted by one of these two rules.

All foreign keys will be transformed into ALTER TABLE statements and concatenated into the ALTER Model textblock, while the table definition will be concatenated into the TEMP Model textblock.

**Rule_LessAlter_TB_CyclicRelation:**

- **Context:** TextBlock
- **Type:** insert, update
- **Condition:** Update=TRUE AND
  UsesInfoFromRelation->notEmpty AND
  UsesInfoFromRelation.ChildAttribute->notEmpty AND
  UsesInfoFromRelation->selectClosure(ToFkKey.ToUnique.
  UniqueIsPartOfRelation)->includes(self)

- **Declaration:**

- **Action:**
  Text:='CREATE TABLE ' ;
  Text:=Text.concat(UsesInfoFromRelation.Name) ;
  Text:=Text.concat(' (\n' ) ;
  -- Collect and concatenate the attribute part
  UsesInfoFromRelation.ChildAttribute.
  AttributeProvidesInforFor->forAll( T : TextBlock |
                   Text:=Text.concat(T.Text) ) ;
  -- Collect and concatenate Unique indices.

87
UsesInfoFromRelation.ToUnique.
    UniqueProvidesInforFor->forAll( T : Textblock | Text:=Text.concat(T.Text) ) ;
-- Collect and concatenate Unique indices.
UsesInfoFromRelation.ToForeignKey->select(
    ToUnique.UniqueIsPartOfRelation.
    RelationProvidesInforFor->forAll( T : Textblock | T.Update=FALSE ) )->
    ( ForeignKeyProvidesInforFor->
        forAll( T : Textblock | Text:=Text.concat(T.Text) )
    ) ;
-- Remove the last ",\n"
Text:=Text.concat(
    Text.substr(1,Text.size-2);
    Text:=Text.concat( ’\n’ ) ;
-- Add result to end of the temp Model textblock.
UsesInfoFromRelation.PartOfModel.
    ModelProvidesInforFor->select(Type=’TEMP’).
    ModelProvidesInforFor->select(Type=’TEMP’).
    Text.concat(Text)
--
-- Collect and concatenate all foreignkeys
-- of this relation.
-- Add this to the ALTER model textblock.
-- Use the ALTER TABLE text in Type
-- and the constraint def. in Text.
UsesInfoFromRelation.ToForeignKey.
ForeignKeyProvidesInforFor->forAll( T : Textblock | UsesInfoFromRelation.PartOfModel.
    ModelProvidesInforFor->select(Type=’ALTER’).
    ModelProvidesInforFor->
        select(Type=’ALTER’).Text.
            concat(T.Type.concat(T.Text)) ;
    ) ;
-- Update complete.
Update:=FALSE ;
Chapter 5
Analysis

In this chapter the work is analysed and various details of the implemented system are discussed. In 5.1 the hierarchical structure of the resulting text is discussed. In 5.2 the benefits of the general transformation model are discussed together with various implementation details.

5.1 Hierarchical structure

The hierarchical model is advantageous since the result can be accessed on different layers. For example, a user can choose to extract either a single table definition from the repository, the whole database definition or just the name of the database (as shown in the example in figure 5.35). This, of course, requires that the different uses has been anticipated at the design time of the rules. A hierarchical approach will also make it easier to create rules as they can be designed in smaller steps. The rule designer does not have to consider building one complicated set of rules, eg. one that creates a whole database, instead he or she can first concentrate on smaller parts such as creating table field definitions and then aggregating them together to form a table definition.
Figure 5.35: Information can be extracted at different level of the hierarchical model

5.2 Discussion about transformation model

The general transformation model offers many features for a rule designer, and because of this, rules may be constructed in many different ways.

5.2.1 Propagating events

Why does an update to a source model trigger a rule that only propagates the event to a textblock? Could not the first rule build the textblock directly? The answer lies in the simplicity of rule writing. By making one rule responsible for building and updating a specific type of textblock, the logic of the transformation is concentrated in one place, thereby reducing the need to distribute duplicates of the same code to several rules.

For example, assume that two context classes exist, A and B, and that the information in these two builds a textblock adhering to A.

As shown in figure 5.36 there are six rules:

- Handling inserts in class A: creating a new textblock, build the textblock from information in A and B.

- Handling updates in class A: rebuilding the textblock from information in A and B.

- Handling deletes in class A: removing the textblock.
Figure 5.36: Two source model classes and a textblock class, rules builds textblock directly

- Handling inserts in class B: rebuild textblock T.
- Handling updates in class B: rebuild textblock T.
- Handling deletes in class B: rebuild textblock T.

As the last three rules are identical, they can be combined into one using "insert,update,delete" as an eventtype. However, this still means that there are four rules of which three contain the necessary code for rebuilding T.

The same configuration but using propagating rules is shown in figure 5.37. There are eight rules:

- Handling inserts in class A: creating a new textblock.
- Handling updates in class A: propagating the update event to the textblock.
- Handling deletes in class A: removing the textblock.
- Handling inserts in class B: propagating the event to the textblock.
- Handling updates in class B: propagating the event to the textblock.
Figure 5.37: Two source model classes and a textblock class, changes propagates

- Handling deletes in class B: propagating the event to the textblock.

- Handling inserts in class T: Build T from data in A and B.

- Handling update in class T: Build T from data in A and B.

There is no need to handle deletes in the T context in this example, therefore there is no rule for this. These eight rules can be simplified down to five rules by combining event types together. While this still is one rule more than the previous example all transformation logic can be found in one rule!

Not only must the transformation logic be duplicated in several rules in the first example, it must be rewritten to work as navigation paths are different.

So, the disadvantage of the propagating approach is that it needs more rules, the advantage is that transformation can be concentrated in one place. The latter makes it easier to maintain integrity between rules as the ruledesigner does not have to replicate changes in several places. But the rule designer has the last decision in this matter and depending on the type of transformation one method may be more suitable than the other.
5.2.2 Cardinality of the role <Class>ProvidesInfoFor

As shown in the excerpt in figure 5.38 the cardinality of the role <Class>ProvidesInfoFor permits multiple values. However, the example rulesets in appendices A, B, C and D only use a single cardinality. The reason for that is simple. A textblock object is only associated with a single source model object, and thus a rule executing in this context can navigate through the role UsesInfoFrom<Class> to either the source model objects, or any other object associated with it. Likewise, a rule executing either in the context of a source model class or in another textblock group, could navigate through the role <Class>ProvidesInfoFor and access the textblock object. However, this navigation is only possible if a single object is included in the association!

In theory an object could be used as an information source for more than one textblock, and thus associated through the role UsesInfoFrom<Class>/<Class>ProvidesInfoFor. Multiple textblocks could be generated if, for example, the information is to be presented in the final resulting text in two different sections. It is also possible that two different rulesets operate on the source model, and thus create textblocks for totally different uses.

For helping the rule designer circumvent this problem the attribute Type has been added to the textblock class. This can be set by a rule to indicate the type of information held in
the textblock. Assuming that multiple textblocks exist for a single source model object, a
rule can navigate to a specific instance by using an OCL select statement as shown in the
example below:

    self.RelationProvidesInfoFor->select(Type='TABLE') ->
        forall( T : Textblock | T.Update:=TRUE)

5.2.3 Includes/IsIncludedBy

Another feature available for the rule designer is the association Includes/IsIncludedBy on
the textblock class. Associations can be set up to reference other textblocks, and a rule can
then use them to construct a textblock. In, for example, the relational model, attribute
textblocks could be linked to the relational textblock, and then concatenated by a rule.
Since the association is ordered, this makes it possible for the execution order of rules to
decide in which order the result should be concatenated later.

In the example in appendix D this association is not used. This is because it is more
work to set an association than to navigate through the source model and find the textblock
this way. However, it could be used when the source model is very large and complicated,
resulting in long navigation paths. Shortcuts can then be set up between textblocks to
simplify rule writing.

5.2.4 Capabilities of the transformation model

Just as OMG avoids saying that MOF can store any model, as it is very difficult (or
even impossible) to prove that, this report will not state that the general transformation
can transform any model into any result. However, some conclusions can be drawn from
the experiences made while designing the rules in appendices D and F. A rather obvious
observation is that to be able to generate a certain result the source model must contain that
information. Information in a model includes data that can be derived from the model. For
example, a set of objects and their values is directly available to the transformation, and
their quantity can be derived. As derivations grows more and more complicated they pass
an invincible border and turns into calculations. An example of a problem which would require calculations is generating optimal SQL as outlined in section 4.6 and appendix F. Generating SQL from a relational model is relatively simple task, as all information is available in the model. However, generating optimal SQL, i.e. utilising a minimum of ALTER TABLE statements, is significantly more difficult as that information is not available in the model! It is not impossible as it can be calculated using some algorithm, but the transformation rules shifts from being merely conversion routines to problem solving algorithms.

5.3 The transformation rulesets

The rulesets presented in appendix D and F is capable of generating SQL from a given relational sourcemodel. A feature of both solutions are the deterministic result. Both rulesets will generate the same \(^1\) result from a given objectstate if executed two (or more) times. In the latter ruleset this is solved by generating ALTER TABLE statements for all foreign keys. This is not necessary in all cases, but if the user should be able to look at the result while working with the source model, he or she would certainly appreciate if the result stays the same with same input. For example, if the textual representation of a table from one execution uses CREATE TABLE constraints, and ALTER TABLE constraints the following, the user could be confused as the results would differ inexplicable. However, the general transformation model does not enforce a deterministic solution, this is up to the rule designer to implement.

\(^1\)I.e. a ruleset generates the same results if executed twice.
Chapter 6

Conclusions

In this chapter a summary of the work together with research contributions are given in section 6.1. In section 6.2 suggestions for future work in the area is given together with ideas that surfaced during the project.

6.1 Summary

A general transformation model have been specified, capable of storing rulesets with transformation targets of different kinds. The rulesets consist of rules written in OCL which use information from a sourcemodel stored in a MOF/UML based format. Changes to the source model are automatically transformed into a textual representation. Different textual output formats can be generated from the same source model, using different rulesets.

The transformation is described independently of the source model stored and the resulting text. Transformation rules are separated from the implementation of both the CASE-tool and the repository. Thus the transformation logics can be changed or extended more easily, in contrast with an application implemented transformation procedure written in, for example, C or Java.

As a proof of concept the transformation model has been shown capable of storing a ruleset for transforming a relational model to a textual representation in SQL. Two implementations are provided: one which use delta (incremental) changes to build a database
from a relational model, utilising SQL 'ALTER TABLE' statements; and one ruleset which
tries to minimise the use of ALTER TABLE statements as this makes the SQL-code more
difficult to read and understand. These two different strategies show the versatile capabili-
ties of the transformation model.

To be able to use OCL as an implementation language, extension was neccesary. The
most significant change is the ability to modify the state of the model, creating and removing
objects and associations. This is in line with the original proposal by Gustavsson and Lings

6.2 Future Work

In this section some ideas and issues not investigated above are presented.

• Investigate whether the sorting constructs provided in OCL are sufficient or whether
  extensions are neccesary.

• Look into whether the result of a method on an object can be used to create an atomic
  value as described in 4.3.1.

• Explore whether conversion routines can be specified in OCL, and then how. Perhaps
  a library of general conversion routines can be built in combination with libraries for
  specific domains.

• Examine whether the return value of a method in a class/object can be used as a
  source in a transformation.

• Could a 'wizard' be helpful in the creation of rules? Smaller construction units could
  be implemented by extending the general transformation model slightly. If a rule is
  built up from sub-rules, then each sub-rule has a specific purpose, eg. adding static
  text to the textblock or navigating to a certain source model object and eliciting infor-
  mation from it. The actual implementation details of expressing rules in OCL could
be hidden from the user, but still be available to the superuser who can understand the more complex details.

- Investigate whether a smaller alias for a concatenation operator could be useful in increasing the readability of the code. For example using ‘&’ as an operator, it could look like this:

  \[ \text{String:=String & String2 & String3 & \ldots & StringN} \]

How much change would this extension impose on the OCL language?

- Another thing that could ease the development of rules is macros. By declaring a template rule and naming it, it can later be used as:

  \[
  \text{GenericInsertRule Relation}
  \text{GenericInsertRule Attribute}
  \]

In this way a lot of similar looking code could be avoided, thereby minimising the amount of information a rule designer must grasp. Because, as can be seen in the example ruleset, the number of rules quickly increases with complexity of model.

- This dissertation has concentrated on what happens when objects are added, updated or removed from the model. But it is also possible that the number and state of objects remain the same and only associations are created or removed. The syntax for specifying rules must be extended to handle these events too, and the rules rewritten accordingly. As associations can be seen as properties of objects, modifying an association on an object should trigger an update event. It must, however, be handled slightly differently.

- Investigate if the transformation could be reversed to read information from a text source and then propagate this information to a model of some kind. The transformation should preferably be able to work in a bi-directional way. How does this
affect the transformation logics? Perhaps additional information must be added to the textual format to enable association between that information and the information in the source model. Could this information be stored in parallel to the transformation metadata, i.e. the text-blocks?
Bibliography


URL: http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.ps.gz.

Appendix A

Execution example of a insert

```plaintext
-- Rulebase ----------------------------------------

Rule1:Rule
  Context   Relation
  Type      insert
  Condition
  Declaration Textblock T
  Action    T.create();
            T.UsesInfoFromRelation:=self;
            T.Update:=TRUE

Rule2:Rule
  Context   Textblock
  Type      insert
  Condition UsesInfoFromRelation->notEmpty AND Update=TRUE
  Declaration
  Action    Text:='CREATE TABLE ';
            Text:=Text.concat(UsesInfoFromRelation.Name);
            Text:=Text.concat( '()' ) ;
            Update:='FALSE

-- Objectstate ----------------------------------

-- Eventqueue -----------------------------------

-- Action / Rule triggering ----------------------

-- External event -------------------------------
insert   Rel1:Relation # A new instance of the class
         # Relation is inserted.
```
Appendix A. Execution example of a insert

```plaintext
-- Object state -----------------------------
Relation
  Name 'Artist'

-- Event queue -----------------------------
insert

-- Action / Rule triggering ----------------
execute Rule1 on Relation yields T1

-- Object state -----------------------------
Relation
  Name 'Artist'
  RelationProvidesInfoFor {T1}  # An association between
  RelationProvidesInfoFor {T1}  # Relation1 and T1.
  # Since reflexivity,
  # T1 is associated with Relation1.

-- Event queue -----------------------------
insert

-- Action / Rule triggering -----------------
execute Rule2 on T1

-- Object state -----------------------------
Relation
  Name 'Artist'
  RelationProvidesInfoFor {T1}

Text
  CREATE TABLE Artist ( )
  Update FALSE
  UsesInfoFromRelation {Relation1}
```

104
Appendix B

Execution example of an update

```plaintext
-- Rulebase -----------------------------------
Rule3: Rule
  Context   Relation
  Type      update
  Condition RelationProvidesInfoFor->notEmpty
  Declaration RelationProvidesInfoFor->forall
     ( T : Textblock | T.Update:=TRUE )

Rule2b: Rule
  Context   Textblock
  Type      insert, update
  Condition UsesInfoFromRelation->notEmpty AND
              Update:=TRUE
  Declaration
    Action    Text:='CREATE TABLE ' ;
              Text:=Text.concat(UsesInfoFromRelation.Name);
              Text:=Text.concat( ' )' );
             Update:=FALSE

-- Objectstate --------------------------------
Rel1: Relation
  RelationProvidesInfoFor {T1}
     Name  'Artist'

T1: Textblock
  Text             'CREATE TABLE Artist ( )'
  Update           FALSE
  UsesInfoFromRelation {Rel1}
```
Appendix B. Execution example of an update

-- Eventqueue -----------------------------

-- Action / Rule triggering -------------

-- External event -----------------------
update Rel1.Name:= 'Group'

-- Objectstate --------------------------
Rel1:Relation
    RelationProvidesInfoFor {T1}
    Name    'Group';

T1:Text-block
    Text    'CREATE TABLE Artist ( )'
    Update   FALSE
    UsesInfoFromRelation {Rel1}

-- Eventqueue -----------------------------
Rel1    update 'Group'   Name

-- Action / Rule triggering -------------
execute Rule3 on Rel1

-- Objectstate --------------------------
Rel1:Relation
    RelationProvidesInfoFor {T1}
    Name    'Group';

T1:Text-block
    Text    'CREATE TABLE Artist ( )'
    Update   TRUE
    UsesInfoFromRelation {Rel1}

-- Eventqueue -----------------------------
T1     update TRUE   Update

-- Action / Rule triggering -------------
execute Rule2b on T1

-- Objectstate --------------------------
Appendix B. Execution example of an update

Rel1: Relation
  RelationProvidesInfoFor {T1}
  Name 'Group'

T1: Text-block
  Text 'CREATE TABLE Group ( )'
  Update FALSE
  UsesInfoFromRelation {Rel1}
Appendix C

Execution example of a delete

```
-- Rulebase -------------------------------------
Rule4:Rule
  Context  Relation
  Type     delete
  Condition RelationProvidesInfoFor->notEmpty
  Declaration
    Action RelationProvidesInfoFor->forAll( T : Textblock | T.Update:=TRUE )
             RelationProvidesInfoFor:={} 

Rule5:Rule
  Context  Textblock
  Type     update
  Condition Update=TRUE AND UsesInfoFromRelation->empty
  Declaration
    Action self.delete()

-- Objectstate -----------------------------
Rel1:Relation
  RelationProvidesInfoFor {T1}
  Name     'Group'

T1:Text-block
  Text     'CREATE TABLE Group ( )'
  Update   FALSE
  UsesInfoFromRelation {Rel1}

-- Eventqueue -----------------------------

-- Action / Rule triggering -------------------
```
Appendix C. Execution example of a delete

```plaintext
-- External event ----------------------------
delete Rel1

-- Objectstate -----------------------------
Rule4:Rule
Rule5:Rule

Rel1:Relation  # Object is deleted but still exists for
    RelationProvidesInfoFor {T1}  # an eventhandler that executes as a
    Name 'Group'  # consequence of the delete event.

T1:Text-block
    Text 'CREATE TABLE Group ( )'
    Update FALSE
    UsesInfoFromRelation {Rel1}

-- Eventqueue -----------------------------
Rel1 delete

-- Action / Rule triggering ---------------
execute Rule4 on Rel1

-- Objectstate -----------------------------
Rule4:Rule
Rule5:Rule

T1:Text-block
    Text 'CREATE TABLE Group ( )'
    Update TRUE
    UsesInfoFromRelation {}

-- Eventqueue -----------------------------
T1 update TRUE Update

-- Action / Rule triggering ---------------
execute Rule5 on T1

-- Objectstate -----------------------------
Rule4:Rule
Rule5:Rule

-- Eventqueue -----------------------------
T1 delete
```
Appendix D

Generation of SQL from a relational model utilising ALTER TABLE

D.1 A relational to SQL transformation ruleset utilising ALTER TABLE

Example of a rulesystem that builds a textual representation in SQL from a relational model. This is done using ALTER TABLE statements for creating foreign key constraints.

D.1.1 Rules for handling source model objects

# Handles new Model objects.
Rule_Model_ins:Rule
  Context Model
  Type insert
  Condition
  Declaration Textblock T
  Action T.create();
     T.UsesInfoFromModel:=self ;
     T.Update:=TRUE

# Handles updates to Model objects.
Rule_Model_Upd:Rule
  Context Model
  Type update
  Condition ModelProvidesInfoFor->notEmpty
  Declaration
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Action ModelProvidesInfoFor->forAll( T : Textblock | T.Update:=TRUE )

# Handles deletes to Model objects.
Rule_Model_Del:Rule
   Context Model
   Type delete
   Condition ModelProvidesInfoFor->notEmpty
   Declaration
   Action ModelProvidesInfoFor->forAll( T : Textblock |
      T.delete() ) ;
      ModelProvidesInfoFor:=

# Handles new Relation objects.
Rule_Relation_Ins:Rule
   Context Relation
   Type insert
   Condition
   Declaration Textblock T
   Action T.create() ;
      TUsesInfoFromRelation:=self ;
      T.Update:=TRUE

# Handles updates to Relation objects.
Rule_Relation_Upd:Rule
   Context Relation
   Type update
   Condition RelationProvidesInfoFor->notEmpty
   Declaration
   Action RelationProvidesInfoFor->forAll( T : Textblock |
      T.Update:=TRUE )

# Handles deletes to Relation objects.
Rule_Relation_Del:Rule
   Context Relation
   Type delete
   Condition RelationProvidesInfoFor->notEmpty
   Declaration
   Action RelationProvidesInfoFor->forAll( T : Textblock |
      T.delete() ) ;
      RelationProvidesInfoFor:=
      -- Propagate the change to the Model textblock.
      PartOfModel.ModelProvidesInfoFor.Update:=TRUE ;
# Handles new Attribute objects.
Rule Attribute Ins:Rule
   Context   Attribute
   Type      insert
   Condition
   Declaration Textblock T
   Action    T.create();
               T.UsesInfoFromAttribute:=self;
               T.Update:=TRUE

# Handles updates to Attribute objects.
Rule_Attribute_Upd:Rule
   Context   Attribute
   Type      update
   Condition AttributeProvidesInfoFor->notEmpty
   Declaration
   Action    AttributeProvidesInfoFor->forAll( T : Textblock |
               T.Update:=TRUE )

# Handles deletes to Attribute objects.
Rule_Attribute_Del:Rule
   Context   Attribute
   Type      delete
   Condition AttributeProvidesInfoFor->notEmpty
   Declaration
   Action    AttributeProvidesInfoFor->forAll( T : Textblock |
               T.delete() );
               AttributeProvidesInfoFor:=[];
               -- Propagate the change to the Relation textblock.
               ParentRelation.RelationProvidesInfoFor.Update:=TRUE;

# Handles new Unique objects.
Rule_Unique_Ins:Rule
   Context    Unique
   Type       insert
   Condition
   Declaration Textblock T
   Action     T.create();
               T.UsesInfoFromUnique:=self;
               T.Update:=TRUE
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

# Handles updates to Unique objects.
Rule_Unique_Upd:Rule
  Context Unique
  Type update
  Condition UniqueProvidesInfoFor->notEmpty
  Declaration
  Action UniqueProvidesInfoFor->forAll( T : Textblock |
                                    T.Update:=TRUE )

# Handles deletes to Unique objects.
Rule_Unique_Del:Rule
  Context Unique
  Type delete
  Condition UniqueProvidesInfoFor->notEmpty
  Declaration
  Action UniqueProvidesInfoFor->forAll( T : Textblock |
                                    T.delete() ) ;
          UniqueProvidesInfoFor:={} ;
          -- Propagate the change to the Relation textblock.
          UniquesPartOfRelation.RelationProvidesInfoFor.
          Update:=TRUE ;

# Handles new ForeignKey objects.
Rule_ForeignKey_Ins:Rule
  Context ForeignKey
  Type insert
  Condition Textblock T
  Declaration
  Action T.create() ;
          T.UsesInfoFromForeignKey:=self ;
          T.Update:=TRUE

# Handles updates to ForeignKey objects.
Rule_ForeignKey_Upd:Rule
  Context ForeignKey
  Type update
  Condition ForeignKeyProvidesInfoFor->notEmpty
  Declaration
  Action ForeignKeyProvidesInfoFor->forAll( T : Textblock |
                                    T.Update:=TRUE )
# Handles deletes to ForeignKey objects.

**RuleForeignKey_Del:**

- **Context:** ForeignKey
- **Type:** delete
- **Condition:** ForeignKeyProvidesInfoFor->notEmpty
- **Declaration:**
  - **Action:** ForeignKeyProvidesInfoFor->forAll(T : Textblock |
    T.delete() );
  - ForeignKeyProvidesInfoFor:={} ;
  - -- Propagate the change to the Relation textblock.
  - ForeignKeyIsPartOfRelation.RelationProvidesInfoFor.
    Update:=TRUE ;

## D.1.2 Rules for handling textblock objects

The following rules all executes in the context of the textblock class. Most transformation logics can be found in these rules.

# Handles models, ie. puts the whole database together.

**Rule TM Model:**

- **Context:** Textblock
- **Type:** insert, update
- **Condition:** UsesInfoFromModel->notEmpty AND Update=TRUE
- **Declaration:**
  - **Action:** Text:='' ;
    UsesInfoFromModel.ConsistsOfRelations.
    RelationProvidesInfoFor->forAll(T : Textblock |
      Text:=Text.concat(T.Text) ;
      Text:=Text.concat('
') ) ;
  - -- Add foreignkey (ALTER TABLES) if they exist.
  - UsesInfoFromModel.ConsistsOfRelations.ToForeignKey.
    ForeignKeyProvidesInfoFor->forAll( T : Textblock |
      Text:=Text.concat(T.Text) ;
      Text:=Text.concat('
') ) ;
  - Update:=FALSE
# Handles fields of non-string type.
Rule_TB_NonStringField:Rule
  Context  Textblock
  Type     insert, update
  Condition UsesInfoFromAttribute->notEmpty AND UsesInfoFromAttribute.Length=0 AND Update=TRUE
  Declaration
  Action   Text:='';
           Text:=Text.concat(UsesInfoFromAttribute.Name);
           Text:=Text.concat( ' ' );
           Text:=Text.concat(UsesInfoFromAttribute.UsesTypeName);
           Text:=Text.concat( ',\n' );
           -- Propagate the change to the Relation textblock.
           UsesInfoFromAttribute.ParentRelation.
           RelationProvidesInfoFor.Update:=TRUE;
           Update:=FALSE

# Handles fields of string type.
Rule_TB_StringField:Rule
  Context  Textblock
  Type     insert, update
  Condition UsesInfoFromAttribute->notEmpty AND UsesInfoFromAttribute.Length>0 AND Update=TRUE
  Declaration
  Action   Text:='';
           Text:=Text.concat(UsesInfoFromAttribute.Name);
           Text:=Text.concat( ' ' );
           Text:=Text.concat(UsesInfoFromAttribute.UsesTypeName);
           Text:=Text.concat( ' (' );
           Text:=Text.concat(UsesInfoFromAttribute.Length.asString);
           Text:=Text.concat( '),\n' );
           -- Propagate the change to the Relation textblock.
           UsesInfoFromAttribute.ParentRelation.
           RelationProvidesInfoFor.Update:=TRUE;
           Update:=FALSE

# Handles valid relations.
Rule_TB_Relation:Rule
  Context  Textblock
  Type     insert, update

115
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Condition UsesInfoFromRelation->notEmpty AND UsesInfoFromRelation.ChildAttribute->notEmpty AND Update=TRUE

Declaration Textblock Waist

Action -- Create and initialize temporary object.
   Waist.create() ; Waist.Text:='' ;
-- Collect and build the attribute part
   UsesInfoFromRelation.ChildAttribute.
   AttributeProvidesInforFor->forAll( T : Textblock |
      Waist.Text:=Waist.Text.concat(T.Text) ) ;
-- Collect and build together Unique indices.
   UsesInfoFromRelation.ToUnique.
   UniqueProvidesInforFor->forAll( T : Textblock |
      Waist.Text:=Waist.Text.concat(T.Text) ) ;
-- Put all parts together
   Text:='CREATE TABLE ' ;
   Text:=Text.concat(UsesInfoFromRelation.Name) ;
   Text:=Text.concat( '(
' ) ;
      -- Add the waist part, except the last ",
   Text:=Text.concat( Waist.Text.substr(1,Waist.Text.size()-2) ;
   Text:=Text.concat( ')
' ) ;
-- Propagate the change to the Model textblock.
   UsesInfoFromRelation.PartOfModel.
   ModelProvidesInforFor.Update:=TRUE
-- Remove temporary objects.
   Waist.delete();
-- Update complete.
   Update:=FALSE ;

# Handles relations with no attributes.
Rule_TB_EmptyRelation:Rule

Context Textblock
Type insert, update
Condition UsesInfoFromRelation->notEmpty AND UsesInfoFromRelation.ChildAttribute->empty AND Update=TRUE

Declaration

Action Text:=''' ;
-- Propagate the change to the Model textblock.
   UsesInfoFromRelation.PartOfModel.
   ModelProvidesInforFor.Update:=TRUE
# Handles unique keys, primary and unique indices.

## Rule_TB_Unique:Rule

**Context**

Textblock

**Type**

insert, update

**Condition**

UsesInfoFromUnique->notEmpty AND 
UsesInfoFromUnique.UniqueConsistsOfAttr->notEmpty AND 
Update=TRUE

**Declaration**

if UsesInfoFromUnique.PrimaryKey then 
    Text:=' PRIMARY KEY (' :
else
    Text:=' UNIQUE (' :
endif

-- Collect all attributes that are in the key
UsesInfoFromUnique.UniqueConsistsOfAttr->
    forAll( A : Attribute |
        Text:=Text.concat(A.Name) ;
        Text:=Text.concat(',') ) ;

-- Remove the last 
Text:=Text.substr(1,Text.size-1) ;
-- Finish the line.
Text:=Text.concat('),
') ;
-- Propagate the change to the Relation textblock.
UsesInfoFromUnique.UniqueIsPartOfRelation.
    RelationProvidesInfoFor.Update:=TRUE ;
Update:=#FALSE

# Handles unique keys with no attributes.

## Rule_TB_UniqueNoAttributes:Rule

**Context**

Textblock

**Type**

insert, update

**Condition**

UsesInfoFromUnique->notEmpty AND 
UsesInfoFromUnique.UniqueConsistsOfAttr->empty AND 
Update=TRUE

**Declaration**

Text:=''

-- Propagate the change to the Relation textblock.
UsesInfoFromUnique.UniqueIsPartOfRelation.
    RelationProvidesInfoFor.Update:=TRUE ;
Update:=#FALSE

117
# Handles valid foreign keys.

**Rule_TB_ForeignKey:Rule**

**Context** Textblock

**Type** insert, update

**Condition**

- UsesInfoFromForeignKey->notEmpty AND
- UsesInfoFromForeignKey.ToUnique->notEmpty AND
- UsesInfoFromForeignKey.ToUnique. UniqueConsistsOfAttr->notEmpty AND
- UsesInfoFromForeignKey.FOREIGNKEYConsistsOfAttr->notEmpty AND
- UsesInfoFromForeignKey.FOREIGNKEYIsPartOfRelation->notEmpty AND
- UsesInfoFromForeignKey->Update=TRUE

**Declaration**

**Action**

```
Text:='ALTER TABLE ';
Text:=Text.concat(UsesInfoFromForeignKey. ForeignKeyIsPartOfRelation.Name);
Text:=Text.concat( 'ADD CONSTRAINT ' ) ;
Text:=Text.concat(UsesInfoFromForeignKey.Name);
Text:=Text.concat( '
FOREIGN KEY ( ' ) ;
-- Collect all attributes that are in the foreign key
Text:=Text.concat(UsesInfoFromForeignKey.FOREIGNKEYConsistsOfAttr->
    forall( A : Attribute |
        Text:=Text.concat(A.Name);
        Text:=Text.concat(',,') ) ;
-- Remove the last ","
Text:=Text.substr(1,Text.size-1) ;
--
Text:=Text.concat( ' REFERENCES ' ) ;
Text:=Text.concat(UsesInfoFromForeignKey.ToUnique. UniqueIsPartOfRelation.Name);
Text:=Text.concat( ' ) ;
-- Collect all attributes that are in the unique key
Text:=Text.concat(UsesInfoFromForeignKey.ToUnique. UniqueConsistsOfAttr->forall( A : Attribute |
    Text:=Text.concat(A.Name);
    Text:=Text.concat(',,') ) ;
-- Remove the last ","
Text:=Text.substr(1,Text.size-1) ;
-- Finish the line.
```
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```plaintext
Text:=Text.concat('')\n'';
-- Propagate the change to the Model textblock.
UsesInfoFromForeignKey.ForeignKeyIsPartOfRelation.
    PartOfModel.ModelProvidesInfoFor.Update:=TRUE;
Update:=FALSE

# Handles invalid foreign keys.
Rule_TB_InvalidForeignKey:Rule
    Context Textblock
    Type      insert, update
    Condition UsesInfoFromForeignKey->notEmpty AND
               (UsesInfoFromForeignKey.ToUnique->empty OR
                UsesInfoFromForeignKey.ToUnique.
                UniqueConsistsOfAttr->notEmpty OR
                UsesInfoFromForeignKey.
                ReferenceConsistsOfAttr->empty OR
                UsesInfoFromForeignKey.
                ForeignKeyIsPartOfRelation->empty) AND
                Update=TRUE
    Declaration
    Action   Text:='';
-- Propagate the change to the Model textblock.
UsesInfoFromForeignKey.ForeignKeyIsPartOfRelation.
    PartOfModel.ModelProvidesInfoFor.Update:=TRUE;
Update:=FALSE
```

D.2 Execution log of a relational model transformed to SQL

The execution log has been divided into several sections to be easier to read.

D.2.1 A relation with three attributes are inserted

```plaintext
-- Objectstate -----------------------------------------

TypeInt:Type
    Name   'INTEGER' 

TypeChar:Type
    Name   'CHAR' 

-- Eventqueue ----------------------------------------
```

119
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

-- Action / Rule triggering ---------------------

-- External event -----------------------------
insert M:Model                                 # A model is created.

-- Objectstate --------------------------------

TypeInt:Type
TypeChar:Type

M:Model
  ConsistOfRelations {}                        

-- Eventqueue ---------------------------------
M  insert                                       

-- Action / Rule triggering -------------------
execute Rule_Model_Ins on M yields T1

-- Objectstate --------------------------------

TypeInt:Type
TypeChar:Type

M:Model
  ConsistOfRelations {}
  ModelProvidesInfoFor {T1}

T1:TextBlock                                 
  Text                                       
  Update          TRUE                        
  UsesInfoFromModel {M}

-- Eventqueue ---------------------------------
T1  insert                                     

-- Action / Rule triggering -------------------
execute Rule_TB_Model on T1

-- Objectstate --------------------------------

TypeInt:Type
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

TypeChar:Type

M:Model
  ConsistOfRelations {}
  ModelProvidesInfoFor {T1}

T1:Textblock
  Text ' ',
  Update FALSE
  UsesInfoFromModel {M}

-- Eventqueue ------------------------

-- Action / Rule triggering ----------

-- External event -------------------
# A relation and its attributes are committed in one atomic event.
insert Attr1:Attribute
insert Attr2:Attribute
insert Attr3:Attribute
insert Rel1:Relation

add association Rel1.PartOfModel={M}
add association M.ConsistOfRelations={Rel1}
add association Rel1.ChildAttribute={Attr1,Attr2,Attr3}
add association Attr1.ParentRelation={Rel1}
add association Attr2.ParentRelation={Rel1}
add association Attr3.ParentRelation={Rel1}

add association Attr1.UsesType={TypeInt}
add association Attr2.UsesType={TypeChar}
add association Attr3.UsesType={TypeChar}

-- Uobjectstate ----------------------

TypeInt:Type
TypeChar:Type

M:Model
  ConsistOfRelations {Rel1}
  ModelProvidesInfoFor {T1}

T1:Textblock
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```
Text
Update FALSE
UsesInfoFromModel {M}

Attr1:Attribute
  Name 'RecId'
  Length 0
  ParentRelation {Rel1}
  UsesType {TypeInt}

Attr2:Attribute
  Name 'Artist'
  Length 50
  ParentRelation {Rel1}
  UsesType {TypeChar}

Attr3:Attribute
  Name 'Year'
  Length 4
  ParentRelation {Rel1}
  UsesType {TypeChar}

Rel1:Relation
  Name 'Record'
  ChildAttribute {Attr1,Attr2,Attr3}
  PartOfModel {M}

-- Eventqueue ------------------------------
Attr1 insert
Attr2 insert
Attr3 insert
Rel1 insert

-- Action / Rule triggering ----------------
execute Rule_Attribute_ins on Attr1 yields T2
execute Rule_Attribute_ins on Attr2 yields T3
execute Rule_Attribute_ins on Attr3 yields T4
execute Rule_Relation_ins on Rel1 yields T5

-- Objectstate --------------------------

TypeInt:Type
TypeChar:Type
```
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

M: Model
  ConsistOfRelations {Rel1}
  ModelProvidesInfoFor {T1}

T1: Textblock
  Text
  Update FALSE
  UsesInfoFromModel {M}

Attr1: Attribute
  Name 'RecId'
  Length 0
  ParentRelation {Rel1}
  UsesType {TypeInt}
  AttributeProvidesInfoFor {T2}

Attr2: Attribute
  Name 'Artist'
  Length 50
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3: Attribute
  Name 'Year'
  Length 4
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T4}

Rel1: Relation
  Name 'Record'
  ChildAttribute {Attr1, Attr2, Attr3}
  RelationProvidesInfoFor {T5}
  PartOfModel {M}

T2: Textblock
  Text
  Update TRUE
  UsesInfoFromAttribute {Attr1}

T3: Textblock
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Text
Update TRUE
UsesInfoFromAttribute {Attr2}

T4:Textblock
Text
Update TRUE
UsesInfoFromAttribute {Attr3}

Tb:Textblock
Text
Update TRUE
UsesInfoFromRelation {Rel1}

-- Eventqueue -------------------------------
T2 insert
T3 insert
T4 insert
T5 insert

-- Action / Rule triggering -----------------
execute Rule_TB_NonStringField on T2
execute Rule_TB_StringField on T3
execute Rule_TB_StringField on T4
execute Rule_TB_Relation on T5

-- Objectstate -------------------------------
TypeInt:Type
TypeChar:Type

M:Model
ConsistOfRelations {Rel1}
ModelProvidesInfoFor {T1}

T1:Textblock
Text
Update TRUE
UsesInfoFromModel {M}

Attr1:Attribute
Name 'RecId'
Length 0
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

ParentRelation {Rel1}
UsesType {TypeInt}
AttributeProvidesInfoFor {T2}

Attr2: Attribute
  Name 'Artist'
  Length 50
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3: Attribute
  Name 'Year'
  Length 4
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T4}

Rel1: Relation
  Name 'Record'
  ChildAttribute {Attr1, Attr2, Attr3}
  RelationProvidesInfoFor {T5}
  PartOfModel {M}

T2: Textblock
  Text , RecId INTEGER,
    Update FALSE
    UsesInfoFromAttribute {Attr1}

T3: Textblock
  Text , Artist CHAR(50),
    Update FALSE
    UsesInfoFromAttribute {Attr2}

T4: Textblock
  Text , Year CHAR(4),
    Update FALSE
    UsesInfoFromAttribute {Attr3}

T5: Textblock
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```
Text 'CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4)
);

Update FALSE
UsesInfoFromRelation {Rel1}

-- Eventqueue -------------------------------
T1   update 'Update' TRUE

-- Action / Rule triggering -------------------
execute Rule_TB_Model on T1

-- Objectstate -------------------------------

TypeInt:Type
TypeChar:Type

M:Model
    ConsistOfRelations {Rel1}
    ModelProvidesInfoFor {T1}

T1:Textblock
    Text 'CREATE TABLE Record(
        RecId INTEGER,
        Artist CHAR(50),
        Year CHAR(4)
    );

    Update FALSE
    UsesInfoFromModel {M}

Attr1:Attribute
    Name 'RecId'
    Length 0
    ParentRelation {Rel1}
    UsesType {TypeInt}
    AttributeProvidesInfoFor {T2}

Attr2:Attribute
```

126
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Name 'Artist'
Length 50
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T3}

Attr3: Attribute
Name 'Year'
Length 4
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T4}

Rel1: Relation
Name 'Record'
ChildAttribute {Attr1, Attr2, Attr3}
RelationProvidesInfoFor {T5}
PartOfModel {M}

T2: Textblock
Text ' RecId INTEGER,'
; Update FALSE
UsesInfoFromAttribute {Attr1}

T3: Textblock
Text ' Artist CHAR(50),'
; Update FALSE
UsesInfoFromAttribute {Attr2}

T4: Textblock
Text ' Year CHAR(4),'
; Update FALSE
UsesInfoFromAttribute {Attr3}

T5: Textblock
Text 'CREATE TABLE Record(
   RecId INTEGER,
   Artist CHAR(50),
   Year CHAR(4)
)
D.2.2 A primary key is added

-- Eventqueue -----------------------------

-- Action / Rule triggering ---------------

-- External event ------------------------
# A unique (primary) key is inserted.
insert Uniql::Unique

add association Uniql.UniqueConsistsOfAttr={Attr1}
add association Attr1.IncludedInUnique={Uniql}

add association Uniql.UniqueIsPartOfRelation={Rel1}
add association Rel1.ToUnique={Uniql}

-- Objectstate ----------------------------

TypeInt::Type
TypeChar::Type

M::Model
ConsistOfRelations {Rel1}
ModelProvidesInfoFor {T1}

T1::Textblock
Text CREATE TABLE Record(
      RecId INTEGER,
      Artist CHAR(50),
      Year CHAR(4)
    )

, Update FALSE
UsesInfoFromModel {M}

Attr1::Attribute
Name 'RecId':
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Length 0
ParentRelation {Rel1}
UsesType {TypeInt}
AttributeProvidesInfoFor {T2}
IncludedInUnique {Uniq1}

Attr2:Attribute
  Name 'Artist'
  Length 50
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3:Attribute
  Name 'Year'
  Length 4
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T4}

Rel1:Relation
  Name 'Record'
  ChildAttribute {Attr1,Attr2,Attr3}
  RelationProvidesInfoFor {T5}
  PartOfModel {M}
  ToUnique {Uniq1}

T2:Textblock
  Text 'ReclId INTEGER,'
  Update FALSE
  UsesInfoFromAttribute {Attr1}

T3:Textblock
  Text 'Artist CHAR(50),'
  Update FALSE
  UsesInfoFromAttribute {Attr2}

T4:Textblock
  Text 'Year CHAR(4),'
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Update FALSE
UsesInfoFromAttribute {Attr3}

Th: Textblock
Text 'CREATE TABLE Record(
   RecId   INTEGER,
   Artist  CHAR(50),
   Year    CHAR(4)
)'

Type 'TABLE';
Update FALSE
UsesInfoFromRelation {Rel1}

Uniq1: Unique
   PrimaryKey TRUE
   UniqueConsistsOfAttr {Attr1}
   UniqueIsPartOfRelation {Rel1}

-- Eventqueue ----------------------------------
Uniq1 insert

-- Action / Rule triggering -------------------
execute Rule_Uniq1_ins on Uniq1 yields T6

-- Objectstate --------------------------------

TypeInt: Type
TypeChar: Type

M: Model
   ConsistOfRelations {Rel1}
   ModelProvidesInfoFor {T1}

T1: Textblock
Text 'CREATE TABLE Record(
   RecId   INTEGER,
   Artist  CHAR(50),
   Year    CHAR(4)
)'
}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Update TRUE
UsesInfoFromModel {M}

Attr1:Attribute
   Name 'RecId'
   Length 0
   ParentRelation {Rel1}
   UsesType {TypeInt}
   AttributeProvidesInfoFor {T2}
   IncludedInUnique {Uniq1}

Attr2:Attribute
   Name 'Artist'
   Length 50
   ParentRelation {Rel1}
   UsesType {TypeChar}
   AttributeProvidesInfoFor {T3}

Attr3:Attribute
   Name 'Year'
   Length 4
   ParentRelation {Rel1}
   UsesType {TypeChar}
   AttributeProvidesInfoFor {T4}

Rel1:Relation
   Name 'Record'
   ChildAttribute {Attr1,Attr2,Attr3}
   RelationProvidesInfoFor {T5}
   PartOfModel {M}
   ToUnique {Uniq1}

T2:Textblock
   Text , RecId INTEGER,
   Update TRUE
   UsesInfoFromAttribute {Attr1}

T3:Textblock
   Text , Artist CHAR(50),
   Update TRUE
   UsesInfoFromAttribute {Attr2}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

T4: Textblock
   Text , Year CHAR(4),
   Update FALSE
   UsesInfoFromAttribute {Attr3}

Tb: Textblock
   Text 'CREATE TABLE Record(
      Recid INTEGER,
      Artist CHAR(50),
      Year CHAR(4)
   )
   ,
   Update FALSE
   UsesInfoFromRelation {Rel1}

Uniq1: Unique
   PrimaryKey TRUE
   UniqueConsistsOfAttr {Attr1}
   UniqueProvidesInfoFor {T6}
   UniqueIsPartOfRelation {Rel1}

T6: Textblock
   Text ,
   Update TRUE
   UsesInfoFromUnique {Uniq1}

-- Eventqueue ----------------------------------------
T6 insert

-- Action / Rule triggering ------------------------
execute Rule_TB_Unique on T6

-- Objectstate --------------------------------------

TypeInt: Type
TypeChar: Type

M: Model
   ConsistOfRelations {Rel1}
   ModelProvidesInfoFor {T1}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

T1:Textblock
Text 'CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4)
);
Update FALSE
UsesInfoFromModel {M}

Attr1:Attribute
Name 'RecId'
Length 0
ParentRelation {Rel1}
UsesType {TypeInt}
AttributeProvidesInfoFor {T2}
IncludedInUnique {Uniq1}

Attr2:Attribute
Name 'Artist'
Length 50
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T3}

Attr3:Attribute
Name 'Year'
Length 4
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T4}

Rel1:Relation
Name 'Record'
ChildAttribute {Attr1,Attr2,Attr3}
RelationProvidesInfoFor {T5}
PartOfModel {M}
ToUnique {Uniq1}

T2:Textblock
Text ' RecId INTEGER,'
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

,  
Update  FALSE
UsesInfoFromAttribute {Attr1}

T3: Textblock
Text  ,  Artist  CHAR(50),
,  
Update  FALSE
UsesInfoFromAttribute {Attr2}

T4: Textblock
Text  ,  Year  CHAR(4),
,  
Update  FALSE
UsesInfoFromAttribute {Attr3}

T5: Textblock
Text  'CREATE TABLE Record(
    RecId  INTEGER,
    Artist  CHAR(50),
    Year  CHAR(4)
)

Update  TRUE
UsesInfoFromRelation {Rel1}

uniq1: Unique
  PrimaryKey  TRUE
  UniqueConsistsOfAttr {Attr1}
  UniqueProvidesInfoFor {T6}
  UniqueIsPartOfRelation {Rel1}

T6: Textblock
Text  ,  PRIMARY KEY (RecId),
,  
Update  FALSE
UsesInfoFromUnique {uniq1}

-- Eventqueue ----------------------------
T5  update 'Update'  TRUE

-- Action / Rule triggering ----------------
execute Rule_TB_Relation on T5
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```sql
-- Objectstate -----------------------------

TypeInt:Type
TypeChar:Type

M:Model
  ConsistOfRelations {Rel1}
  ModelProvidesInfoFor {T1}

T1:Textblock
  Text
    'CREATE TABLE Record(
      RecId    INTEGER,
      Artist   CHAR(50),
      Year     CHAR(4)
    );
  
  Update    TRUE
  UsesInfoFromModel {M}

Attr1:Attribute
  Name    'RecId'
  Length  0
  ParentRelation {Rel1}
  UsesType  {TypeInt}
  AttributeProvidesInfoFor {T2}
  IncludedInUnique {Uniq1}

Attr2:Attribute
  Name    'Artist'
  Length  50
  ParentRelation {Rel1}
  UsesType  {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3:Attribute
  Name    'Year'
  Length  4
  ParentRelation {Rel1}
  UsesType  {TypeChar}
  AttributeProvidesInfoFor {T4}
```
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Rel1:Relation
   Name 'Record'
   ChildAttribute {Attr1, Attr2, Attr3}
   RelationProvidesInfoFor {T5}
   PartOfModel {M}
   ToUnique {Uniq1}

T2:Textblock
   Text , RecId INTEGER,
   ,
   Update FALSE
   UsesInfoFromAttribute {Attr1}

T3:Textblock
   Text , Artist CHAR(50),
   ,
   Update FALSE
   UsesInfoFromAttribute {Attr2}

T4:Textblock
   Text , Year CHAR(4),
   ,
   Update FALSE
   UsesInfoFromAttribute {Attr3}

T5:Textblock
   Text 'CREATE TABLE Record(
      RecId INTEGER,
      Artist CHAR(50),
      Year CHAR(4),
      PRIMARY KEY (RecId)
   )',

   Update FALSE
   UsesInfoFromAttribute {Attr1}

Uniq1:Unique
   PrimaryKey TRUE
   UniqueConsistsOfAttr {Attr1}
   UniqueProvidesInfoFor {T6}
   UniqueIsPartOfRelation {Rel1}

1.36
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

T6:Textblock
   Text , PRIMARY KEY (RecId),
   :
   Update   FALSE
   UsesInfoFromUnique {Uniq1}

-- Eventqueue -------------------------------------
T1     update 'Update'   TRUE

-- Action / Rule triggering ----------------------
execute Rule_TB_Model on T1

-- Uobjectstate -------------------------------

TypeInt:Type
TypeChar:Type

M:Model
   ConsistOfRelations {Rel1}
   ModelProvidesInfoFor {T1}

T1:Textblock
   Text 'CREATE TABLE Record(
       RecId INTEGER,
       Artist CHAR(50),
       Year CHAR(4),
       PRIMARY KEY (RecId)
   )
   :
   Update   FALSE
   UsesInfoFromModel {M}

Attr1:Attribute
   Name 'RecId'
   Length 0
   ParentRelation {Rel1}
   UsesType {TypeInt}
   AttributeProvidesInfoFor {T2}
   IncludedInUnique {Uniq1}

Attr2:Attribute
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Name 'Artist'
Length 50
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T3}

Attr3: Attribute
Name 'Year'
Length 4
ParentRelation {Rel1}
UsesType {TypeChar}
AttributeProvidesInfoFor {T4}

Rel1: Relation
Name 'Record'
ChildAttribute {Attr1,Attr2,Attr3}
RelationProvidesInfoFor {T5}
PartOfModel {M}
ToUnique {Uni1}

T2: Textblock
Text 'RecId INTEGER,';
Update FALSE
UsesInfoFromAttribute {Attr1}

T3: Textblock
Text 'Artist CHAR(50),';
Update FALSE
UsesInfoFromAttribute {Attr2}

T4: Textblock
Text 'Year CHAR(4),';
Update FALSE
UsesInfoFromAttribute {Attr3}

Tb: Textblock
Text 'CREATE TABLE Record(
  RecId INTEGER,
  Artist CHAR(50),
  Year CHAR(4).'}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```
PRIMARY KEY (RecId)
}

Update FALSE
UsesInfoFromRelation {Rel1}

Uniq1:Unique
  PrimaryKey TRUE
  UniqueConsistsOfAttr {Attr1}
  UniqueProvidesInfoFor {T6}
  UniqueIsPartOfRelation {Rel1}

T6:Textblock
  Text PRIMARY KEY (RecId),
  
  Update FALSE
  UsesInfoFromUnique {Uniq1}
```

D.2.3 A relation with a foreign key is added

```
-- Eventqueue ------------------------------

-- Action / Rule triggering -----------------

-- External event --------------------------

# A new relation is inserted which consists of two
# attributes, the last which is a foreign key.
insert Attr4:Attribute
  insert Attr5:Attribute
insert FKey1:ForeignKey
insert Rel2:Relation

add association Rel2.ChildAttribute={Attr4,Attr5}
add association Attr4.ParentRelation={Rel2}
add association Attr5.ParentRelation={Rel2}
add association Attr4.UsesType={TypeInt}
add association Attr5.UsesType={TypeInt}

add association Rel2.ToForeignKey={FKey1}
add association FKey1.ForeignKeyIsPartOfRelation={Rel2}
add association FKey1.ToUnique={Uniq1}
add association FKey1ForeignKeyConsistsOfAttr={Attr5}
```
add association Uniq1.IsUsedByForeignKey={FKey1}
add association Rel2.PartOfModel={M}
add association M.ConsistsOfRelations={Rel2}

-- Objectstate -----------------------------

 McInt:Type
 TypeChar:Type

 M:Model
   ConsistsOfRelations {Rel1,Rel2}
   ModelProvidesInfoFor {T1}

 T1:Textblock
   Text
     CREATE TABLE Record(
       RecId INTEGER,
       Artist CHAR(50),
       Year CHAR(4),
       PRIMARY KEY (RecId)
     )
   ;
   Update FALSE
   UsesInfoFromModel {M}

 Attr1:Attribute
   Name 'RecId'
   Length 0
   ParentRelation {Rel1}
   UsesType {TypeInt}
   AttributeProvidesInfoFor {T2}
   IncludedInUnique {Uniq1}

 Attr2:Attribute
   Name 'Artist'
   Length 50
   ParentRelation {Rel1}
   UsesType {TypeChar}
   AttributeProvidesInfoFor {T3}

 Attr3:Attribute
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Name 'Year'  
Length 4  
ParentRelation {Rel1}  
UsesType {TypeChar}  
AttributeProvidesInfoFor {T4}

Rel1: Relation  
Name 'Record'  
ChildAttribute {Attr1, Attr2, Attr3}  
RelationProvidesInfoFor {T5}  
PartOfModel {M}  
ToUnique {Uniq1}

T2: Textblock  
Text , RecId INTEGER,  
Update FALSE  
UsesInfoFromAttribute {Attr1}

T3: Textblock  
Text , Artist CHAR(50),  
Update FALSE  
UsesInfoFromAttribute {Attr2}

T4: Textblock  
Text , Year CHAR(4),  
Update FALSE  
UsesInfoFromAttribute {Attr3}

T5: Textblock  
Text 'CREATE TABLE Record(  
RecId INTEGER,  
Artist CHAR(50),  
Year CHAR(4),  
PRIMARY KEY (RecId)  
)  
,'  
Update FALSE  
UsesInfoFromRelation {Rel1}

Uniq1: Unique
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

PrimaryKey TRUE
UniqueConsistsOfAttr {Attr1}
UniqueProvidesInfoFor {T6}
UniqueIsPartOfRelation {Rel1}
IsUsedByForeignKey={FKey1}

T6:Texthlock
   Text PRIMARY KEY (RecId),
   Update FALSE
   UsesInfoFromUnique {Uniq1}

Attr4:Attribute
   Name 'SaleId'
   Length 0
   ParentRelation {Rel2}
   UseType {TypeInt}
   AttributeProvidesInfoFor {}

Attr5:Attribute
   Name 'RecId'
   Length 0
   ParentRelation {Rel2}
   UseType {TypeInt}
   AttributeProvidesInfoFor {}

FKey1:ForeignKey
   Name 'fk_RecId_Record'
   ForeignKeyIsPartOfRelation {Rel2}
   ToUnique {Uniq1}
   ForeignKeyConsistsOfAttr {Attr5}

Rel2:Relation
   Name 'Sale'
   ChildAttribute {Attr4,Attr5}
   RelationProvidesInfoFor {}
   PartOfModel {}
   ToForeignKey {FKey1}

-- Eventqueue -------------------------------
Attr4 insert
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Attr5 insert
FKey1 insert
Rel2 insert

-- Action / Rule triggering -----------------
execute Rule_Attribute_ins on Attr4 yields T7
execute Rule_Attribute_ins on Attrb yields T8
execute Rule_ForeignKey_ins on FKey1 yields T9
execute Rule_Relation_ins on Rel1 yields T10

-- Objectstate -----------------------------

TypeInt:Type
TypeChar:Type

M:Model
ConsistOfRelations {Rel1, Rel2}
ModelProvidesInfoFor {T1}

T1:TextTable
Text 'CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4),
    PRIMARY KEY (RecId)
);
,
Update FALSE
UsesInfoFromModel {M}

Attr1:Attribute
Name 'RecId':
Length 0
ParentRelation {Rel1}
UsesType {TypeInt}
AttributProvidesInfoFor {T2}
IncludedInUnique {Uniq1}

Attr2:Attribute
Name 'Artist':
Length 50
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

    ParentRelation {Rel1}
    UsesType {TypeChar}
    AttributeProvidesInfoFor {T3}

Attr3:Attribute
    Name 'Year'
    Length 4
    ParentRelation {Rel1}
    UsesType {TypeChar}
    AttributeProvidesInfoFor {T4}

Rel1:Relation
    Name 'Record'
    ChildAttribute {Attr1,Attr2,Attr3}
    RelationProvidesInfoFor {T5}
    PartOfModel {N}
    ToUnique {Uniql}

T2:Textblock
    Text  , RecId INTEGER,
    Update FALSE
    UsesInfoFromAttribute {Attr1}

T3:Textblock
    Text  , Artist CHAR(50),
    Update FALSE
    UsesInfoFromAttribute {Attr2}

T4:Textblock
    Text  , Year CHAR(4),
    Update FALSE
    UsesInfoFromAttribute {Attr3}

Tb:Textblock
    Text  'CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4),
    PRIMARY KEY (RecId)
    )

144
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Update FALSE
UsesInfoFromRelation {Rel1}

Uniq1:Unique
  PrimaryKey TRUE
  UniqueConsistsOfAttr {Attr1}
  UniqueProvidesInfoFor {T6}
  UniqueIsPartOfRelation {Rel1}
  IsUsedByForeignKey={FKey1}

T6:Textblock
  Text , PRIMARY KEY (RecId),
  Update FALSE
  UsesInfoFromUnique {Uniq1}

Attr4:Attribute
  Name 'SaleId'
  Length 0
  ParentRelation {Rel2}
  UseType {TypeInt}
  AttributeProvidesInfoFor {T7}

T7:Textblock
  Text ,
  Update TRUE
  UsesInfoFromAttribute {Attr4}

Attr5:Attribute
  Name 'RecId'
  Length 0
  ParentRelation {Rel2}
  UseType {TypeInt}
  AttributeProvidesInfoFor {T8}

T8:Textblock
  Text ,
  Update TRUE
  UsesInfoFromAttribute {Attr5}

FKey1:ForeignKey
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Name 'fk_RecID_Record'
ForeignKeyIsPartOfRelation {Rel2}
ToUnique {Uniq1}
ForeignKeyConsistsOfAttr {Attr5}
ForeignKeyProvidesInfoFor {T9}

T9:Textblock
Text ' ':
Update TRUE UsesInfoFromForeignKey {FKey1}

Rel2:Relation
Name 'Sale'
ChildAttribute {Attr4,Attr5}
RelationProvidesInfoFor {T10}
PartOfModel {M}
ToForeignKey {FKey1}

T10:Textblock
Text ' ':
Update TRUE UsesInfoFromRelation {Rel2}

-- Eventqueue -----------------------------
T7 insert T8 insert T9 insert T10 insert

-- Action / Rule triggering -----------------
execute Rule_TB_NonStringField on T7
execute Rule_TB_NonStringField on T8
execute Rule_TB_ForeignKey on T9
execute Rule_TB_Relation on T10

-- Objectstate -----------------------------

TypeInt:Type
TypeChar:Type

M:Model
ConsistOfRelations {Rel1,Rel2}
ModelProvidesInfoFor {T1}

T1:Textblock
  Text 'CREATE TABLE Record(
    RecId INTEGER,
    Artist CHAR(50),
    Year CHAR(4),
    PRIMARY KEY (RecId)
  )

  ,

  Update TRUE
  UsesInfoFromModel {M}

Attr1:Attribute
  Name 'RecId'
  Length 0
  ParentRelation {Rel1}
  UsesType {TypeInt}
  AttributeProvidesInfoFor {T2}
  IncludedInUnique {Uniq1}

Attr2:Attribute
  Name 'Artist'
  Length 50
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3:Attribute
  Name 'Year'
  Length 4
  ParentRelation {Rel1}
  UsesType {TypeChar}
  AttributeProvidesInfoFor {T4}

Rel1:Relation
  Name 'Record'
  ChildAttribute {Attr1,Attr2,Attr3}
  RelationProvidesInfoFor {T5}
  PartOfModel {M}
  ToUnique {Uniq1}
T2: Textblock
Text 'CREATE TABLE Record(
    RecId  INTEGER,
    Artist  CHAR(50),
    Year  CHAR(4),
    PRIMARY KEY (RecId)
)

Update  FALSE
UsesInfoFromRelation {Rel1}

T3: Textblock
Text 'CREATE TABLE Record(
    RecId  INTEGER,
    Artist  CHAR(50),
    Year  CHAR(4),
    PRIMARY KEY (RecId)
)

Update  FALSE
UsesInfoFromRelation {Rel1}

T4: Textblock
Text 'CREATE TABLE Record(
    RecId  INTEGER,
    Artist  CHAR(50),
    Year  CHAR(4),
    PRIMARY KEY (RecId)
)

Update  FALSE
UsesInfoFromRelation {Rel1}

T5: Textblock
Text 'CREATE TABLE Record(
    RecId  INTEGER,
    Artist  CHAR(50),
    Year  CHAR(4),
    PRIMARY KEY (RecId)
)

Update  FALSE
UsesInfoFromRelation {Rel1}

Uniq1: Unique
PrimaryKey  TRUE
UniqueConsistsOfAttr {Attr1}
UniqueProvidesInfoFor {T6}
UniqueIsPartOfRelation {Rel1}
IsUsedByForeignKey={FKey1}

T6: Textblock
Text 'CREATE TABLE Record(
    RecId  INTEGER,
    PRIMARY KEY (RecId).

Update  FALSE
UsesInfoFromUnique {Uniq1}

Attr4: Attribute
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Name 'SaleId'
Length 0
ParentRelation {Rel2}
UsesType {TypeInt}
AttributeProvidesInfoFor {T7}

T7:Textblock
Text 'SaleId INTEGER,',
Update FALSE
UsesInfoFromAttribute {Attr4}

Attr5:Attribute
Name 'RecId'
Length 0
ParentRelation {Rel2}
UsesType {TypeInt}
AttributeProvidesInfoFor {T8}

T8:Textblock
Text 'RecId INTEGER,',
Update FALSE
UsesInfoFromAttribute {Attr5}

FKey1:ForeignKey
Name 'fk_RecId_Record'
ForeignkeyIsPartOfRelation {Rel2}
ToUnique {Uniq1}
ForeignkeyConsistsOfAttr {Attr5}
ForeignkeyProvidesInfoFor {T9}

T9:Textblock
Text 'ALTER TABLE Sale ADD CONSTRAINT fk_RecId_Record
FOREIGN KEY (RecId) REFERENCES Record(RecId)
,
Update FALSE
UsesInfoFromForeignKey {FKey1}

Rel2:Relation
Name 'Sale'
ChildAttribute {Attr4,Attr5}
RelationProvidesInfoFor {T10}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

```
PartOfModel {M}
ToFoerignKey {FKKey1}

T10:Textblock
Text
'CREATE TABLE Sale ( 
    SaleId  INTEGER,
    RecId   INTEGER 
);',
Update     FALSE
UsesInfoFromRelation {Rel2}

-- Eventqueue ---------------------------------------
T1  update  'Update'  TRUE

-- Action / Rule triggering ------------------------
execute Rule_TR_Model on T1

-- Ubjectstate -------------------------------------

TypeInt:Type
TypeChar:Type

M:Model
ConsistOfRelations {Rel1,Rel2}
ModelProvidesInfoFor {T1}

T1:Textblock
Text
'CREATE TABLE Record( 
    RecId   INTEGER,
    Artist  CHAR(50),
    Year    CHAR(4),
    PRIMARY KEY (RecId)
); 

CREATE TABLE Sale ( 
    SaleId  INTEGER,
    RecId   INTEGER 
); 

ALTER TABLE Sale ADD CONSTRAINT fk_RecID_Record 
    FOREIGN KEY (RecId) REFERENCES Record(RecId)
, 

1511
```
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Update    FALSE
UsesInfoFromFile {M}

Attr1:Attribute
  Name     'RecId'
  Length   0
  ParentRelation {Rel1}
  UsesType  {TypeInt}
  AttributeProvidesInfoFor {T2}
  IncludedInUnique {Uniq1}

Attr2:Attribute
  Name     'Artist'
  Length   50
  ParentRelation {Rel1}
  UsesType  {TypeChar}
  AttributeProvidesInfoFor {T3}

Attr3:Attribute
  Name     'Year'
  Length   4
  ParentRelation {Rel1}
  UsesType  {TypeChar}
  AttributeProvidesInfoFor {T4}

Rel1:Relation
  Name     'Record'
  ChildAttribute {Attr1,Attr2,Attr3}
  RelationProvidesInfoFor {T5}
  PartOfModel {M}
  ToUnique   {Uniq1}

T2:Textblock
  Text    ,    RecId INTEGER,

    Update    FALSE
    UsesInfoFromAttribute {Attr1}

T3:Textblock
  Text    ,    Artist CHAR(50),

    Update    FALSE
    UsesInfoFromAttribute {Attr2}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

T4: Textblock
    Text
    Year CHAR(4),
    ;
    Update FALSE
    UsesInfoFromAttribute {Attr3}

Tb: Textblock
    Text
    'CREATE TABLE Record(
        RecId INTEGER,
        Artist CHAR(50),
        Year CHAR(4),
        PRIMARY KEY (RecId)
    );
    Update FALSE
    UsesInfoFromRelation {Rel1}

Uniq1: Unique
    PrimaryKey TRUE
    UniqueConsistsOfAttr {Attr1}
    UniqueProvidesInfoFor {T6}
    UniqueIsPartOfRelation {Rel1}
    IsUsedByForeignKey={FKey1}

T6: Textblock
    Text
    PRIMARY KEY (RecId),
    ;
    Update FALSE
    UsesInfoFromUnique {Uniq1}

Attr4: Attribute
    Name 'SaleId'
    Length 0
    ParentRelation {Rel2}
    UsesType {TypeInt}
    AttributeProvidesInfoFor {T7}

T7: Textblock
    Text
    SaleId INTEGER,
    ;
    Update FALSE
    UsesInfoFromAttribute {Attr4}
Appendix D. Generation of SQL from a relational model utilising ALTER TABLE

Attr5:Attribute
   Name 'RecId'
   Length 0
   ParentRelation {Rel2}
   UseType {TypeInt}
   AttributeProvidesInfoFor {T8}

T8:Textblock
   Text 'RecId INTEGER,'
   Update FALSE
   UsesInfoFromAttribute {Attr5}

FKKey1:ForeignKey
   Name 'fk_RecID_Record'
   ForeignKeyIsPartOfRelation {Rel2}
   ToUnique {Uniq1}
   ForeignKeyConsistsOfAttr {Attr5}
   ForeignKeyProvidesInfoFor {T9}

T9:Textblock
   Text 'ALTER TABLE Sale ADD CONSTRAINT fk_RecID_Record
           FOREIGN KEY (RecId) REFERENCES Record(RecId)
       ;
   Update FALSE
   UsesInfoFromForeignKey {FKKey1}

Rel12:Relation
   Name 'Sale'
   ChildAttribute {Attr4,Attr5}
   RelationProvidesInfoFor {T10}
   PartOfModel {M}
   ToForeignKey {FKKey1}

T10:Textblock
   Text 'CREATE TABLE Sale ( SaleId INTEGER,
                           RecId INTEGER
                     );
   Update FALSE
   UsesInfoFromRelation {Rel12}
Appendix E

Output from PostgreSQL interactive terminal

test=# CREATE TABLE Record(  
test(#   RecId  INTEGER,  
test(#   Artist  CHAR(50),  
test(#   Year  CHAR(4),  
test(#   PRIMARY KEY (RecId)  
test(# ):  
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit  
   index 'record_pkey' for table 'record'
CREATE  
test=# CREATE TABLE Sale (  
test(#   SaleId  INTEGER,  
test(#   RecId  INTEGER  
test(# ):  
CREATE  
test=# ALTER TABLE Sale ADD CONSTRAINT fk RecId Record  
test=# FOREIGN KEY (RecId) REFERENCES Record(RecId);  
NOTICE: ALTER TABLE ... ADD CONSTRAINT will create implicit  
   trigger(s) for FOREIGN KEY check(s)
CREATE  
test=# \d  
     List of relations  
        Name | Type | Owner  
--------------+-------+-------  
record | table | postgres  
sale | table | postgres  
(2 rows)
Appendix E. Output from PostgreSQL interactive terminal

```sql
test=# \
record
Table "record"
 Attribute | Type | Modifier
-----------|------|--------
 recid     | integer | not null
 artist    | character(50)
 year      | character(4)
index: record_pkey

test=# \
sale
Table "sale"
 Attribute | Type | Modifier
-----------|------|--------
 saleid    | integer |
 recid     | integer |
```

```sql
test=# 
```
Appendix F

A more intelligent ruleset for generating SQL

F.1 A relational to SQL transformation ruleset

This appendix contains a ruleset which is capable of generating SQL from a relational model in a more intelligent way, thus utilising less ALTER TABLE statements than previous examples.

F.1.1 Rules for handling source model objects

# Handles new Model objects.
Rule_LessAlter_Model_ins:Rule
    Context Model
    Type   insert
    Condition
        Declaration Textblock TT, Textblock TA, Textblock TL
        Action
            -- TT = Textblock Temporary, holds partial result
            -- while transformation is running.
            -- TA = Textblock Alter table, holds all ALTER
            -- TABLEs until final concatenation.
            -- TL = Textblock Live, holds final result of
            -- transformation. Valid at all times.
            TT.create();
            TT.Type:='TEMP';
            TT.UsesInfoFromModel:=self;
            TT.Update:=TRUE;
            --
            TA.create();
Appendix F. A more intelligent ruleset for generating SQL

```latex
TA.Type='ALTER'; TA.UsesInfoFromModel:=self; TA.Update:=FALSE;

-- TL.create(); TL.Type='LIVE'; TL.UsesInfoFromModel:=self; TL.Update:=FALSE

# Handles updates to Model objects.
Rule_LessAlter_Model_Upd:Rule
  Context Model
  Type update
  Condition ModelProvidesInfoFor->notEmpty
  Declaration
  Action -- Force an update on the whole model.
          ModelProvidesInfoFor->select(Type='TEMP')->{
            (Update:=TRUE)
          }

# Handles deletes to Model objects.
Rule_LessAlter_Model_Del:Rule
  Context Model
  Type delete
  Condition ModelProvidesInfoFor->notEmpty
  Declaration
  Action -- Delete all textblocks.
          ModelProvidesInfoFor->forAll( T : Textblock |
                                       T.delete() ) ;
          ModelProvidesInfoFor:={}  

# -------------------

# Handles new Attribute objects.
Rule_LessAlter_Attribute_Ins:Rule
  Context Attribute
  Type insert
  Condition
  Declaration Textblock T
  Action T.create();
            T.UsesInfoFromAttribute:=self ;
            T.Update:=TRUE
```

157
Appendix F. A more intelligent ruleset for generating SQL

# Handles updates to Attribute objects.
Rule_LessAlter_Attribute_Upd:Rule
  Context    Attribute
  Type       update
  Condition  AttributeProvidesInfoFor->notEmpty
  Declaration
  Action     AttributeProvidesInfoFor->forAll( T : Textblock | T.Update:=TRUE )

# Handles deletes to Attribute objects.
Rule_LessAlter_Attribute_Del:Rule
  Context    Attribute
  Type       delete
  Condition  AttributeProvidesInfoFor->notEmpty
  Declaration
  Action     AttributeProvidesInfoFor->forAll( T : Textblock | T.delete() );
             AttributeProvidesInfoFor:={};
             -- Propagate the change to the temp Model textblock.
             ParentRelation.PartOfModel.ModelProvidesInfoFor->
             select(Type='TEMP').Update:=TRUE;

# -------------------

# Handles new Unique objects.
Rule_Unique_Ins:Rule
  Context    Unique
  Type       insert
  Condition
  Declaration Textblock T
  Action     T.create();
             T.UsesInfoFromUnique:=self;
             T.Update:=TRUE

# Handles updates to Unique objects.
Rule_Unique_Upd:Rule
  Context    Unique
  Type       update
  Condition  UniqueProvidesInfoFor->notEmpty
  Declaration
  Action     UniqueProvidesInfoFor->forAll( T : Textblock | T.Update:=TRUE )
# Handles deletes to Unique objects.
Rule_Unique_Del:Rule
  Context Unique
  Type delete
  Condition UniqueProvidesInfoFor->notEmpty
  Declaration
  Action UniqueProvidesInfoFor->forall( T : Textblock |
                     T.delete() ) ;
                    UniqueProvidesInfoFor:={} ;
                    -- Propagate the change to the temp Model textblock.
                    UniquesPartUfRelation . PartUfModel .
                    ModelProvidesInfoFor->
                    select(Type='TEMP').Update:=TRUE ;

# ---------------------

# Handles new ForeignKey objects.
Rule_ForeignKey_ins:Rule
  Context ForeignKey
  Type insert
  Condition
  Declaration Textblock T
  Action T.create() ;
     T . UsesInfoFromForeignKey:=self ;
     T . Update:=TRUE ;

# Handles updates to ForeignKey objects.
Rule_ForeignKey_UPd:Rule
  Context ForeignKey
  Type update
  Condition ForeignKeyProvidesInfoFor->notEmpty
  Declaration
  Action ForeignKeyProvidesInfoFor->forall( T : Textblock |
                     T . Update:=TRUE )

# Handles deletes to ForeignKey objects.
Rule_ForeignKey_Del:Rule
  Context ForeignKey
  Type delete
  Condition ForeignKeyProvidesInfoFor->notEmpty
  Declaration
  Action ForeignKeyProvidesInfoFor->forall( T : Textblock |
                     T . Delete:=TRUE )
T.delete() ;
ForeignKeyProvidesInfoFor:={} ;
-- Propagate the change to the temp Model textblock.
ForeignKeyIsPartOfRelation.PartialModel.
    ModelProvidesInfoFor->
select(Type='TEMP').Update:=TRUE ;

# ---------------------

# Handles new Relation objects.
Rule_Relation_Ins:Rule
    Context     Relation
    Type        insert
    Condition
    Declaration Textblock T
    Action      T.create() ;
                T.UsesInfoFromRelation:=self ;
                -- Propagate the change to the temp Model textblock.
                PartialModel.ModelProvidesInfoFor->
select(Type='TEMP').Update:=TRUE ;

# Handles updates to Relation objects.
Rule_Relation_Upd:Rule
    Context     Relation
    Type        update
    Condition   RelationProvidesInfoFor->notEmpty
    Declaration
    Action      -- Propagate the change to the temp Model textblock.
                PartialModel.ModelProvidesInfoFor->
select(Type='TEMP').Update:=TRUE ;

# Handles deletes to Relation objects.
Rule_Relation_Del:Rule
    Context     Relation
    Type        delete
    Condition   RelationProvidesInfoFor->notEmpty
    Declaration
    Action      RelationProvidesInfoFor->forEach(T: Textblock |
                T.delete() ) ;
                RelationProvidesInfoFor:={} ;
                -- Propagate the change to the temp Model textblock.
                PartialModel.ModelProvidesInfoFor->
select(Type='TEMP').Update:=TRUE ;
### F.1.2 Rules for handling source model objects

# This rule will make all relations build this textblock
# from the information in the relation textblocks (and other).
# This rule will not trigger (again) if the live textblock
# still not is updated, i.e. there is a transformation running.

**Rule_LessAlter_TB_Model_TEMP:Rule**

**Context** Textblock

**Type** insert, update

**Condition**

- Update=TRUE AND
- Type='TEMP' AND
- UsesInfoFromModel->notEmpty AND
- UsesInfoFromModel.ModelProvidesInfoFor->
  select(Type='LIVE').Update=FALSE

**Declaration**

**Action**

-- Clear TEMP och ALTER textblocks.

```text
Text=''
```

-- Make all relations first rebuild themself and
-- then the temporary textblock.

```text
UsesInfoFromModel.ModelProvidesInfoFor->
select(Type='ALTER').Text=''
```

-- Live textblock should update when
-- transformation is finished.

```text
UsesInfoFromModel.ModelProvidesInfoFor->
select(Type='LIVE').Update=TRUE
```

-- Ok, we are done.

Update:=FALSE

# Update the live textblock from the temporary
# if no transformation is running.

**Rule_LessAlter_TR_Model_LIVE:Rule**

**Context** Textblock

**Type** insert, update

**Condition**

- Update=TRUE AND
- Type='LIVE' AND
- UsesInfoFromModel->notEmpty AND
- UsesInfoFromModel.ConsistsOfRelations.

```text
RelationProvidesInfoFor->forAll(Update:=TRUE)
```

# if no transformation is running.

Update:=FALSE

161
Appendix F. A more intelligent ruleset for generating SQL

Action
Text:=UsesInfoFromModel.ModelProvidesInfoFor->
   select(Type='TEMP').Text;
Text:=Text.concat(UsesInfoFromModel.
   ModelProvidesInfoFor->select(Type='ALTER').Text);
-- Ok, we are done.
Update:=#FALSE

# ---------------------

# Handles fields of any type.
Rule_LessAlter_TH_Attribute:Rule

Context Textblock
Type insert, update
Condition Update=TRUE AND
UsesInfoFromAttribute->notEmpty

Declaration
Action
Text:=
   Text.concat(UsesInfoFromAttribute.Name);
Text:=Text.concat(' ',
   Text.concat(UsesInfoFromAttribute.UsesType.Name);
if UsesInfoFromAttribute.Length=0 then
   Text:=Text.concat('(',
   Text.concat(UsesInfoFromAttribute.
   Length.asString);
   Text:=Text.concat(' '),
   Text.concat('
')
else
   Text:=Text.concat(' ',
   Text.concat(' ',
   Text.concat('
')
endif
-- Propagate the change to the temp Model textblock.
UsesInfoFromAttribute.ParentRelation.
   PartOfModel.ModelProvidesInfoFor->
   select(Type='TEMP').Update:=TRUE;
-- Ok, we are done.
Update:=#FALSE

# ---------------------

# Handles unique keys, primary and unique indicies.
Rule_LessAlter_TB_Unique:Rule

Context Textblock
Type insert, update
Condition Update=TRUE AND
UsesInfoFromUnique->notEmpty AND 
UsesInfoFromUnique.UniqueConsistsUfAttr->notEmpty

Declaration

Text:='CONSTRAINT' 
Text:=Text.concat(UsesInfoFromUnique.Name) ;
if UsesInfoFromUnique.PrimaryKey=TRUE then 
   Text:=Text.concat( ' PRIMARY KEY ' ) ;
else 
   Text:=Text.concat( ' UNIQUE ' ) ;
endif

-- Collect all attributes that are in the key
UsesInfoFromUnique.UniqueConsistsUfAttr->

forAll( A : Attribute | 
   Text:=Text.concat(A.Name) ; 
   Text:=Text.concat(',') ) ;
-- Remove the last ","
Text:=Text.substr(1,Text.size-1) ;
-- Finish the line.
Text:=Text.concat( ',
' ) ;
-- Propagate the change to the temp Model textblock.
UsesInfoFromUnique.UniquelsPartUfRelation.
   PartUfModel.ModelProvidesInfoFor->
      select(Type='TEMP').Update:=TRUE ;
-- Update complete.
Update:=FALSE

# Handles unique keys with no attributes.
Rule_LessAlter_TB_UniqueNoAttributes:Rule

Context Textblock
Type insert, update
Condition Update=TRUE AND 
UsesInfoFromUnique->notEmpty AND 
UsesInfoFromUnique.UniqueConsistsUfAttr->empty

Declaration

Text:=' ' ;
-- Propagate the change to the temp Model textblock.
UsesInfoFromUnique.UniquelsPartUfRelation.
   PartUfModel.ModelProvidesInfoFor->
      select(Type='TEMP').Update:=TRUE ;
-- Update complete.
Update:=FALSE

#------------------

163
# Handles valid foreign keys. The generated textblock will
# contain the beginning of an ALTER TABLE statement (in Type)
# and a CREATE TABLE foreign key constraint (in Text).
# Concatenated they form a complete ALTERTABLE constraint.

Rule_LessAlter_TB_ForeignKey:Rule

Context Textblock
Type insert, update
Condition Update=TRUE AND
 UsesInfoFromForeignKey->notEmpty AND
 UsesInfoFromForeignKey.ToUnique->notEmpty AND
 UsesInfoFromForeignKey.ToUnique. UniqueConsistsUfAttr->notEmpty AND
 UsesInfoFromForeignKey. ForeignKeyConsistsUfAttr->notEmpty AND
 UsesInfoFromForeignKey. ForeignKeyIsPartOfRelation->notEmpty

Declaration Action -- Store some text in the Type variable (unused).

Type:='ALTER TABLE ';
Type:=Type.concat(UsesInfoFromForeignKey.
 ForeignKeyIsPartOfRelation.Name);
Type:=Type.concat( ' ADD ');
-- Build the constraint.
Text:='CONSTRAINT ');
Text:=Text.concat(UsesInfoFromForeignKey.Name);
Text:=Text.concat( '
 FOREIGN KEY (');
-- Collect all attributes that are in the foreign key
UsesInfoFromForeignKey.ForeignKeyConsistsUfAttr->
forall( A : Attribute |
    Text:=Text.concat(A.Name);
    Text:=Text.concat(' ,') );
-- Remove the last " ,"
Text:=Text.substr(1,Text.size-1);
--
Text:=Text.concat( ') REFERENCES ');
Text:=Text.concat(UsesInfoFromForeignKey.ToUnique.
 ForeignKeyIsPartOfRelation.Name);
Text:=Text.concat( ')' );
-- Collect all attributes that are in the unique key
UsesInfoFromForeignKey.ToUnique. UniqueConsistsOfAttr->forall( A : Attribute |
    Text:=Text.concat(A.Name);
\textbf{Text:} = \texttt{Text.concat(’,’,)} ;
-- Remove the last ","
\textbf{Text:} = \texttt{Text.substr(1, Text.size - 1)} ;
-- Finish the line.
\textbf{Text:} = \texttt{Text.concat( ’\n’ ) ;}
-- Propagate the change to the temp Model textblock.
\textbf{UsesInfoFromForeignKey.ForeignKeyIsPartOfRelation.}
\textbf{PartUfModel.ModelProvidesInfoFor->}
\textbf{select(Type='TEMP').Update:=TRUE ;}
-- Update complete.
\textbf{Update:}=FALSE

\# Handles invalid foreign keys.
\textbf{Rule_LessAlter_TB_InvalidForeignKey:Rule}
\textbf{Context} \hspace{1cm} \textbf{Textblock}
\textbf{Type} \hspace{1cm} insert, update
\textbf{Condition} \hspace{1cm} \texttt{UsesInfoFromForeignKey->notEmpty AND}
\hspace{1cm} \texttt{(UsesInfoFromForeignKey.ToUnique->empty OR}
\hspace{1cm} \texttt{UsesInfoFromForeignKey.ToUnique.}
\hspace{1cm} \texttt{UniqueConsistsUfAttr->notEmpty OR}
\hspace{1cm} \texttt{UsesInfoFromForeignKey.}
\hspace{1cm} \texttt{ReferenceConsistsUfAttr->empty OR}
\hspace{1cm} \texttt{UsesInfoFromForeignKey.}
\hspace{1cm} \texttt{ForeignKeyIsPartOfRelation->empty)} AND
\texttt{Update=TRUE}

\textbf{Declaration}
\textbf{Action} \hspace{1cm} \textbf{Text:=’’ :}
-- Propagate the change to the temp Model textblock.
\textbf{UsesInfoFromForeignKey.ForeignKeyIsPartOfRelation.}
\textbf{PartUfModel.ModelProvidesInfoFor->}
\textbf{select(Type='TEMP').Update:=TRUE ;}
-- Update complete.
\textbf{Update:}=FALSE

\# -------------------------------

\# Handles valid relations, there the reference target of any foreign
\# key exists. All relations that is part of a cyclic reference chain
\# are suppressed. Thus only Relations which are possible to generate
\# without using ALTER TABLE are processed.
\textbf{Rule_LessAlter TH NormalRelation:Rule}
\textbf{Context} \hspace{1cm} \textbf{Textblock}
# Appendix F: A more intelligent ruleset for generating SQL

<table>
<thead>
<tr>
<th>Type</th>
<th>insert, update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>Update=TRUE AND</td>
</tr>
<tr>
<td></td>
<td>UsesInfoFromRelation-&gt;notEmpty AND</td>
</tr>
<tr>
<td></td>
<td>UsesInfoFromRelation.ChildAttribute-&gt;notEmpty AND</td>
</tr>
<tr>
<td></td>
<td>UsesInfoFromRelation.ToForeignKey.ToUnique.</td>
</tr>
<tr>
<td></td>
<td>UniquesPartOfRelation.RelationProvidesInfoFor-&gt;forAll( T : Textblock</td>
</tr>
<tr>
<td></td>
<td>UsesInfoFromRelation-&gt;selectClosure(ToForeignKey.</td>
</tr>
<tr>
<td></td>
<td>ToUnique,UniqueIsPartOfRelation)-&gt;excludes(self)</td>
</tr>
</tbody>
</table>

## Declaration

## Action

    Text:='CREATE TABLE ' ;
    Text:=Text.concat(UsesInfoFromRelation.Name) ;
    Text:=Text.concat( '(
    -- Collect and build the attribute part
    UsesInfoFromRelation.ChildAttribute.
    AttributeProvidesInforFor->forAll( T : Textblock |
    Text:=Text.concat(T.Text) ) ;
    -- Collect and concatenate Unique indices.
    UsesInfoFromRelation.ToUnique.
    UniqueProvidesInforFor->forAll( T : Textblock |
    Text:=Text.concat(T.Text) ) ;
    -- Collect and concatenate foreignkeys.
    UsesInfoFromRelation.ToForeignKey.
    ForeignKeyProvidesInforFor->forAll( T : Textblock |
    Text:=Text.concat(T.Text) ;
    Text:=Text.concat(’,’ ) ) ;
    -- Remove the last ",\n"
    Text:=Text.concat(Text.substr(1,Text.size-2)) ;
    Text:=Text.concat(‘
)’
    -- Add result to end of the temp Model textblock.
    UsesInfoFromRelation.PartOfModel.
    ModelProvidesInfoFor->select(Type='TEMP').
    Text.concat(Text)
    -- Update complete.
    Update:=FALSE ;

# Handle valid Relations. One or more of the foreignkey constraints
# transitively reference this object. Thus there is a cyclic
# dependency which must be broken up. This is done by creating
# all foreign keys as ALTER TABLE statements.

Rule_LessAlter_TB_CyclicRelation:Rule

Context Textblock

166
Type insert, update
Condition Update=TRUE AND
  UsesInfoFromRelation->notEmpty AND
  UsesInfoFromRelation.ChildAttribute->notEmpty AND
  UsesInfoFromRelation->selectClosure(ToForeignKey.ToUnique.
    UniqueIsPartOfRelation)->includes(self)
Declaration
Action
  Text:='CREATE TABLE ';
  Text:=Text.concat(UsesInfoFromRelation.Name);
  Text:=Text.concat(' (\n') ;
  -- Collect and concatenate the attribute part
  UsesInfoFromRelation.ChildAttribute.
    AttributeProvidesInforFor->forall( T : Textblock |
      Text:=Text.concat(T.Text) ) ;
  -- Collect and concatenate Unique indices.
  UsesInfoFromRelation.ToUnique.
    UniqueProvidesInforFor->forall( T : Textblock |
      Text:=Text.concat(T.Text) ) ;
  -- Collect and concatenate Unique indices.
  UsesInfoFromRelation.ToForeignKey->select( 
    ToUnique.UniqueIsPartOfRelation.
      RelationProvidesInforFor->forall( 
        T : Textblock | T.Update=FALSE ) )->
    ( ForeignKeyProvidesInforFor->
      forall( T : Textblock |
        Text:=Text.concat(T.Text) ) ) ;
  -- Remove the last ",\n"
  Text:=Text.concat( 
    Text.substr(1,Text.size-2) ;
  Text:=Text.concat(' \n') ;
  -- Add result to end of the temp Model textblock.
  UsesInfoFromRelation.PartUfModel.
    ModelProvidesInfoFor->select(Type='TEMP') .
    Text:=UsesInfoFromRelation.PartUfModel .
    ModelProvidesInfoFor->select(Type='TEMP').
    Text.concat(Text)
  --
  -- Collect and concatenate all foreignkeys
  -- of this relation.
  -- Add this to the ALTER model textblock.
  -- Use the ALTER TABLE text in Type
  -- and the constraint def. in Text.
UsesInfoFromRelation.ToForeignKey.
ForeignKeyProvidesInfoFor->forall(T : Textblock |
   UsesInfoFromRelation.PartOfModel.
   ModelProvidesInfoFor->select(Type='ALTER').
   ModelProvidesInfoFor->
   select(Type='ALTER').Text.
   concat(T.Type.concat(T.Text))
); -- Update complete.
Update:=FALSE ;