REPOSITORY SUPPORT FOR VISUALIZATION IN RELATIONAL DATABASES

Gerrit Griebel

Submitted by Gerrit Griebel to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the department of computer science.

September 1996

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.

Gerrit Griebel

Abstract

Modern database management systems allow the specification of integrity constrains to ensure a proper database state at any point in time. According to the semantics of these constraints, integrity violating updates may be restricted or lead to further updates which eventually result in a consistent database state.

In this dissertation we target visualization of these processes to support data model design and evaluation processes. We consider a three layered Entity Relationship approach to database design. An Extended Entity Relationship Model is transformed into an abstract relational model which in turn can be used to generate a data schema to be used with a specific DBMS.

We develop the design of a repository database, consistent with current standards, for storing all layers of modeling together with the mapping between them. This repository forms the basis for a proposed visualization tool.

Contents

Chapter 1

Introduction

Designing *information systems* (IS) is a complex task where structuring is needed to cope with complexity. *Database management systems* (DBMS) are used to hide the complexity of low level storage details from the IS application developer. They feature a separation of intensional and extensional information. The intensional information becomes just another type of data, often referred to as *meta-data* or schema. One methodology of designing such meta-data, in particular for relational data models, involves the use of the *extended entity relationship* (EER) notation as a design artifact. This is then mapped, possibility by using some tool, to an abstract relational representation which in turn can be used to generate the data schema for a particular target DBMS. The concern of this work was threefold.

1. To investigate the mapping procedure with respect to preservation of EER model semantics. This will require a close study of relational integrity constraints and their implementation in current DBMS.

2. To design a repository suitable for storing all three model representations and mappings between them. The structure, especially of the relational layer in the repository, was to avoid, if possible, any loss of semantics. Existing repository standards were to be considered. The repository was to allow for convenient access from a host language.

3. To propose a visualization architecture based on the repository and the according target application database. This was to support visualization of the meta-data and the data at different levels by mapping the application database data to the abstract relational level and to the EER level.

The user of such a visualization tool may manipulate data through the tool and monitor active behavior due to integrity constraints, at the EER level as well as the relational level. Today's DBMS allow predefined behavior to maintain the integrity of the database. Updates to the database which violate its referential integrity may cause cascades of other operations which eventually result in a consistent database state. This can only be visualized through representing the actual data which is interactively modified.

Turning static graphical notations into animated simulation which includes data as well as active behavior *facilitates communication* between the software developer and other parties in the requirements specification phase. It supports "*creative data modeling*" [66, 18] and can be used within a *visual debugging tool* during the development of the data model and the programs. Another application could be to explain a data model by using it as an *explanation system* for other database developers, domain experts and possibly also end users. Limited visualization systems could even *enrich end user's interfaces*. Finally, such a tool could *inspire the community* to new developments in this area.

In this dissertation we emphasize the development of the repository design and explore visualization issues as a demonstration of its utility.

Chapter 2 sets the framework for this work. It briefly describes the requirements specification phase for database applications and introduces each layer of abstraction (EER, abstract relational and target DBMS) in the process of modeling. Its referential integrity aspects and their mapping from layer to layer will be described in more detail. New notations will be introduced where nothing appropriate is found in the literature. A specific *Universe of Discourse* (UoD) is introduced to exemplify theoretical concepts. The chapter concludes with a brief discussion of repositories and an overview of related work. In Chapter 3 the

visualization requirements are defined. They set the framework for Chapter 4 which details the development of a repository which captures all modeling layers and associated mappings. The design of a prototype visualization tool which accesses the repository is described separately in Chapter 5. Chapter 6 contains a summary, a critical review and a discussion of possible future work.

Chapter 2

Background and Foundation

2.1 Introduction

In this chapter the framework is set for the development of a prototype visualization architecture which supports design and maintenance of database schemas. Herein methods and artifacts are chosen and motivated. To structure the facts within a UoD a *data model* is needed. The term "data model" refers to a certain representation of data in a form influenced primarily by the needs of the UoD and also by design, functional and implementation issues. This chapter will be essentially about different chosen models and the way they are created, used, stored, and visualized during the design process.

This work takes into account data modeling; the process and behavior oriented views (see [57, p. 30]) are not dealt with. Behavior in the context of this work is only a means to ensure a consistent database state.

A software design process is likely to be divided into phases. This is especially true for the design of database-driven applications. The following sections are organized according to the design steps shown in Figure 1.

Section 2.2 serves in the context of this work only as a justification for visual methods within modeling and development. Sections 2.3 and 2.4 are of main interest in this chapter

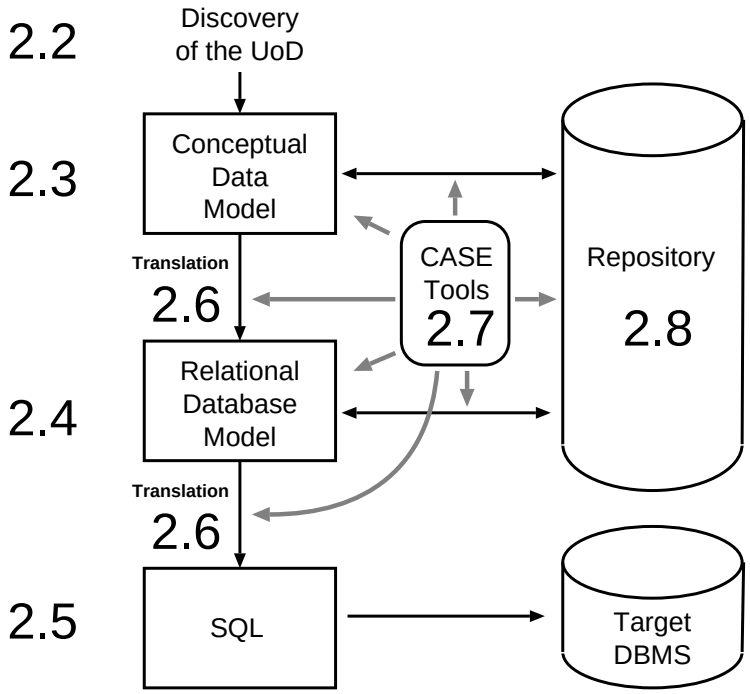


Figure 1: Graphical outline of chapter "Literature Survey"

as they introduce the different layers of modeling and how they maintain their integrity. Section 2.5 details the ways of implementation with current standards and current commercial DBMS implementations respectively. Section 2.6 is on the mapping of the introduced layers which can be realized by mapping tools (Section 2.7). The role of repositories is discussed in Section 2.8.

2.2 Discovery of the Universe of Discourse

In [39] (quoted in [66]) design is described as a "goal oriented, constrained, decision-making, exploration, and learning activity that operates within a context that depends on the designer's perception of the context."

There are always different ways of modeling database schemas as part of a IS [46] depending on the methodology used, the database designer's [32, p. 9] experience in modeling, her or his knowledge of the UoD, and the priorities of certain properties of the model. Since the data model determines the structure of the overall database application, it influences its software quality attributes like extensibility, reusability, performance, maintainability or understandability. A trial of different ways to model a database schema could help to avoid problems later in the design process. The worst case would be that the domain expert (system analyst [32, p. 10]) reveals design flaws when the development of programs has been finished. Therefore the involvement of domain experts or even end users [32, p. 10] at this early stage can be of great benefit as it helps to avoid costly mistakes [55, p. 49]. Against this it is sometimes stated that the end user should not know implementation details. This is often a view of system designers and not of end users and it can be questioned. "A manager [being an end user] does not have to know how an information system works, only how to use it." is revealed as a common misconception in [2]. The same author argues that ease-of-use criteria may lead to programs where the functionality is hidden behind a user friendly interface and that end users should be able to know also the functionality. This participation would facilitate the design process and increase the motivation of working with the system later on. Abstract schemas become clearer when they are populated with actual data. This is a strong argument for a visual prototyping system. Using sample data during the design process helps not only to find an appropriate data model it may also assist in finding the way the data is constrained by the real-world, i.e. to explicitly state which data configurations make sense and which not. As already mentioned in the introduction in Chapter 1 is this a

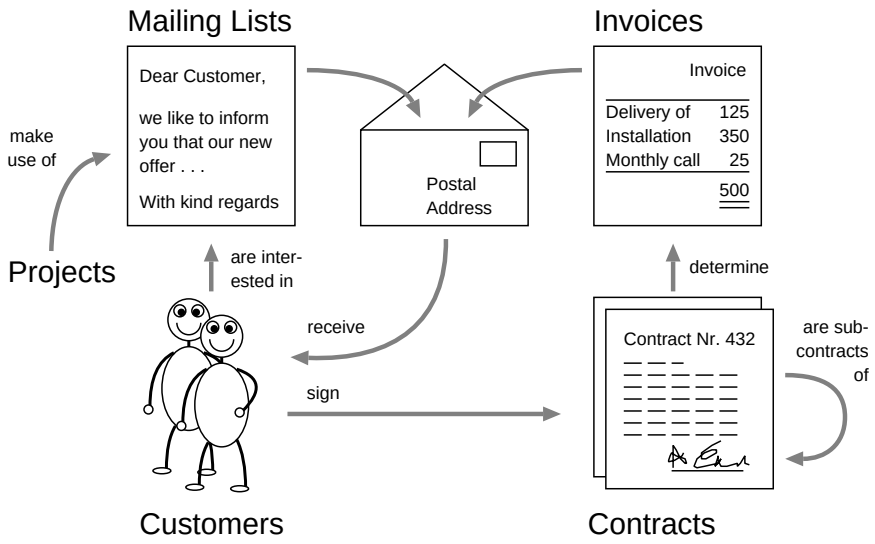


Figure 2: Universe of Discourse: Telephone Service Provider

major issue and the following sections will focus on this aspect.

Throughout this text references to an example UoD are made whenever appropriate to explain theoretical concepts. It is a simplified part of the postal mail dispatching for a telephone service provider. Customers sign contracts for telephone services and afterwards receive invoices and other information which is relevant to them. Figure 2 illustrates this UoD. The example is kept as simple as possible while allowing the demonstration of most of the data modeling concepts.

The next step will be to work out a conceptual data schema which is the topmost of the three mentioned layers of modeling.

2.3 Conceptual Database Schema Design

A model for so-called conceptual schema design should (1) be a means of communication about real-world objects between participants and (2) be a notation which can be used by database designers as a basis for the physical database design [32, p. 453].

In this phase as rich semantics as possible should be captured within a formal data model. This *semantic data model* should enable the user to "model the data in a manner similar to the human perception of the application" [60]. Central to this is the ability to specify *integrity constraints* (IC) within the model. These are often a matter of course to humans but have to be made explicit in the process of modeling. Semantic data models are a design artifact in that they allow to specify declaratively properties of data without being concerned if and how they can be ensured in a real database implementation.

The most used semantic data model is the *Entity-Relationship* (ER) model [14] introduced by Chen in 1976. Its graphical notation and semantics have been variously extended into *Extended Entity-Relationship* (EER) models of which there are now "over fifty versions" according to [43]. The EER model was chosen for this work, because it is well supported by the literature and by modeling tools (see Section 2.7). Some of the prominent variants of EER modeling are: Elmasri's [32], Information Engineering [54] and IDEF1X (FIPS standard) [34]. They differ in notation and supported features. Figure 3 gives an overview of included semantics and IC respectively. The only official standard [48] from ISO is limited to basic ER modeling.

Common to all ER model variants is the distinction of *entities*, their *attributes* and *relationships* between them. Some attributes in each entity form a primary key which identifies each entity instance. Entity instances themself are not considered. Relationships are of particular interest in this work, since they relate the entities to a conceptual schema and bear the referential integrity properties of the model. Each *role* of a relationship (see Figure 4) has a cardinality, i.e. a maximum and minimum number of relationship instances per entity

[1] IDEF1X calls the subtype concept "category", not related to Categories

	[14] Chen's ER	[32] Elm./Nav.	[54] Crow's foot	[34] IDEF1X
Attribute and entity integrity				
Attribute Domain				yes
Multi-valued Attribute		yes		
Composite Attribute		yes		
Derived Attribute		yes		
Weak Entity	yes	yes	yes	yes
Relationship integrity				
1:1 Relationship	yes	yes	yes	yes
1:N Relationship	yes	yes	yes	yes
N:M Relationship	yes	yes	yes	
Identifying Relationship	yes	yes	yes	yes
Participation Constraint	yes	yes	yes	yes
Ternary Relationship	yes	yes		
Relationship with Attribute	yes	yes		
Relationship between Relationship	yes	yes		
Extended concepts				
Type/Subtype relation[1]		yes	yes	yes
overlapping/mutually excl. subtypes		yes	yes	
partial/total subtypes		yes	yes	yes
Categories		yes		

Figure 3: Comparison of EER models and their features

instance. The cardinalities of both roles in a binary relationship are called *cardinality ratio* (see Figure 4 for examples[2]). The expressiveness and notation of the cardinality varies from notation to notation. Relationships may be weak (identifying) or may denote a supertype/subtype connection (also known as data abstraction, see Figure 4). The author of [1] concludes that "for the purposes of data modeling, however, it would be sufficient to define the EER model as: EER model = the ER model + data abstraction constructs"

[2] 1 optional ≡ 0–1; 1 mandatory ≡ 1; N optional ≡ 0–∞; N mandatory ≡ 1–∞

The usefulness of the semantic features of a model depends on different factors like expressiveness, ease of learning for unskilled personnel, understandability and also weather they can be mapped unambiguously to lower levels as discussed in Section 2.6. As can be seen in Figure 3, Chen's and Elmasri's notations allow *ternary relationships*, relationships between relationships and relationships with attributes. Therefore they are referred to as *semantically richer* than other variants. In [80] the author states that "there is a trend to binary relations with no attributes only between entities, because this simplifies the model". IDEF1X is a step in that direction. It combines EER modeling directly with relational concepts and therefore eliminates the need for a mapping from the EER conceptual level to an abstract relational level.

The use of a specific EER variant is often influenced by the availability of a schema design tool (see Section 2.7) which is most suitable for a project. But of course the choice of a tool is also influenced by its support for a methodology. Only if no tool is used may the developer freely choose a model. Within this work the notation used by *Information Engineering* (IE) is adopted. The diagrammatic notation of IE, also known as *crow's foot*, is depicted in Figure 4 [61]. Only the notation is used by the modeling tool S-Designor [74], which will be introduced in Section 2.7, and only the notation is of interest within this work. Information Engineering in general is a methodology which combines strategic planning and design of information systems [36]. A data schema of the telephone provider UoD using this notation is given without further reference in Figure 5.

The cardinality notation of IE is named *look across cardinality* in [35]. Figure 20 on page 20 shows an example case where each entity instance of "Customer" relates to at least one entity instance "Contract", denoted by the Mandatory symbol on *the other side* of the relationship. Other ER notations work exactly the opposite way around.

Using EER modeling the designer declares properties of entity instances and relationship instances without giving directions on how to ensure these properties. Nevertheless approaches exit to derive update behavior directly from the EER model. It is claimed in

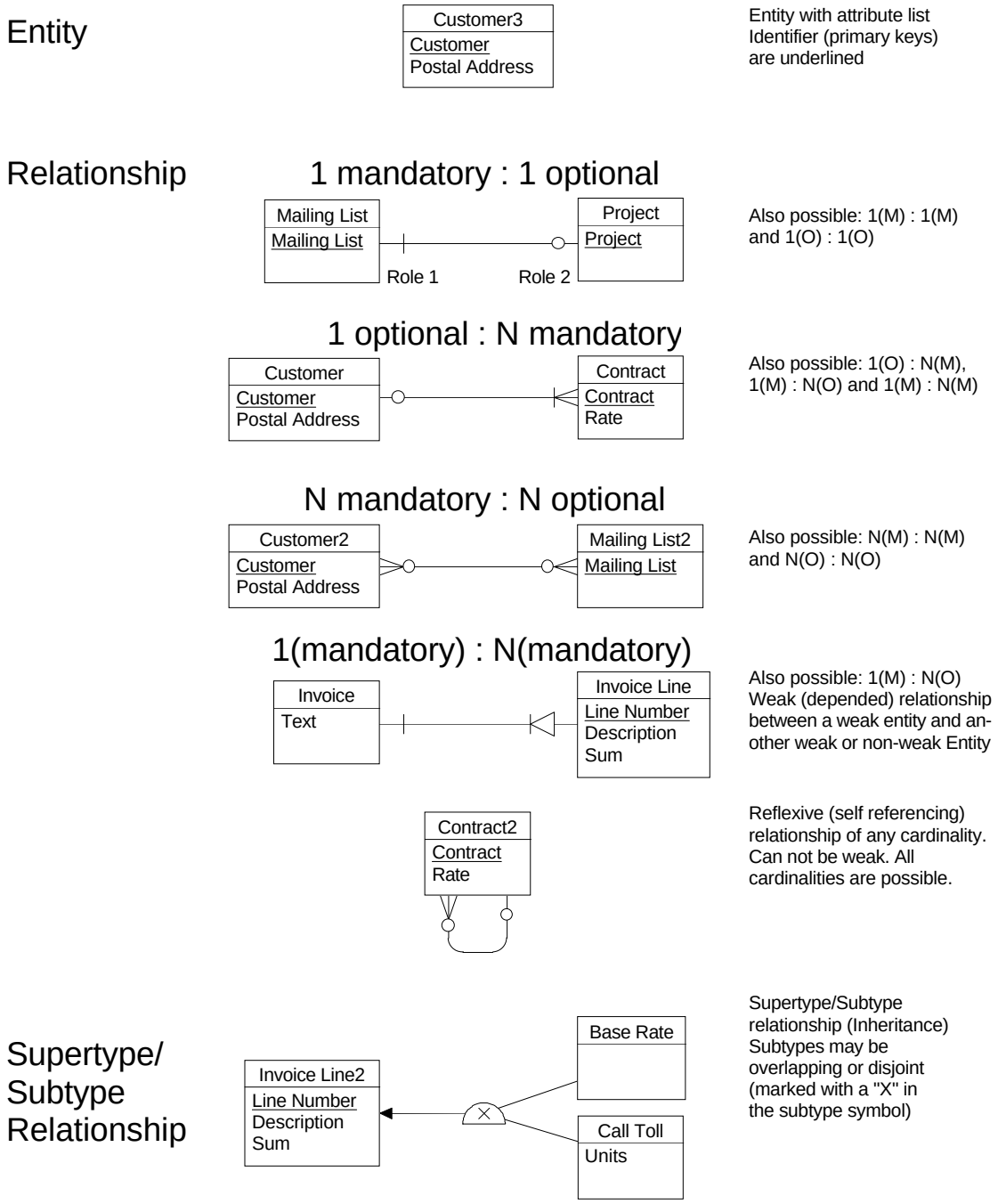


Figure 4: Notation used by Information Engineering (Crow's foot)

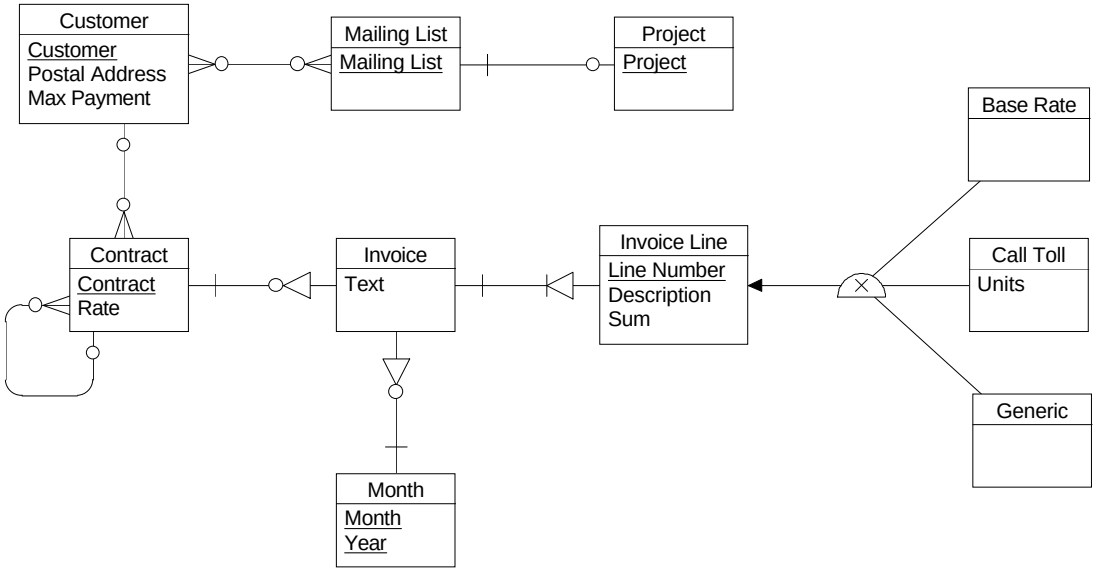


Figure 5: UoD (see Figure 2) schema using the notation of Figure 4

[29] that "Besides the obvious advantage of using the ER-model as the conceptual schema interface, there are additional advantages of the ER-model in enforcing integrity" and certain update behavior is proposed for "ordinary relationships", "total [mandatory] relationships" and "weak relationships". This schema is extended by "Generalization hierarchy" and "Subset hierarchy" in [1, pp. 65–67]. The schema suggests that for each update operation on the EER model a single appropriate integrity ensuring action can be identified. However, as shown later in Section 2.6, alternatives do exist, e.g. a deletion of a relationship instance may or may not include connected entity instances. Therefore the ultimate derivation of update propagation from the plain EER model is not possible, it rather may give some default behavior which might be refined manually.

The EER model is commonly seen as a conceptual model rather than a logical database model. It is used for modeling an application database schema that will be translated to a relational schema as shown in Figure 1. In this way not only a multilevel schema but also a multilevel representation of data is obtained. Data does not directly populate the conceptual schema, but data in the relational DBMS can be interpreted as data at the conceptual level

by using a stored mapping of the levels. Data resides physically in a relational database but it can be queried directly at this level or indirectly at the EER level by translating EER queries to relational queries. The effect of both kinds of queries should be made visible at all levels including the EER level. Update propagation will be modeled on the relational level using relational update propagation properties mostly derived from the EER semantics by a mapping procedure.

Even back in 1976 when Chen proposed the ER model the intention was to provide a model for database design which could be implemented by some of the data models which were competing at that time: the network, the entity set and the relational model. The last is discussed in the following section.

Formal Relational [23]	**SQL-92 [56]**	**Used in this text**
Relation	`TABLE`	Table
Tuple	Row	Row
Attribute	Column	Column
Domain	`DOMAIN`	Domain
Reference	Reference	Reference
Reference instance	Reference instance	Reference instance
Primary key	`PRIMARY KEY`	PK[3]
Alternate key	not available	-[4]
Candidate key	not available	-
Foreign key	`FOREIGN KEY`	FK
Primary key role	-	PK-role
Foreign key role	-	FK-role
Weak relation	-	Weak table
Cardinality	`COUNT(*)`	Cardinality[5]
Degree	Number of columns	Number of columns
Insert	`INSERT`	Insert
Delete	`DELETE`	Delete
Modify	`UPDATE`	Modify[6]

Figure 6: Relational terminology used in this text

2.4 Relational Database Model

The relational model of data is the prevailing data model for business applications. It was introduced by Codd [16] in 1970 and is well founded on its underlying relational algebra. It has a few well understood operators (*projection, selection, join*) on a single type of data (the *table*) and makes use of set theory (*union, intersection, difference*). The relational model overcomes the data dependencies of its predecessors (ordering, indexing and access path dependence [16]) which allows for declarative database languages and a clear logical layering of functionality. Todays commercial relational DBMS are powerful and mature processing systems using concepts like *query optimization*, *crash recovery*, *transactions* and *views*. An

[3] PK consists of a uniquely identifying column or column combination, as defined in [23, page 79]. If more than one uniquely identifying column combination is available a PK has to be chosen like with SQL-92.

[4] Not needed in the context of this work

[5] Almost only used in this text for the number of referenced rows

[6] The term *update* is used as a generic term for *insert, delete* or *modify*.

in depth description of the model and the implementation techniques are covered by various textbooks, e.g. [32] and [23]. Figure 6 introduces the terminology used within this work. The formal terms are not used in this text to avoid confusion with the terms of the EER model.

In the following, relational concepts are described which can be used to implement EER properties. The mapping itself is discussed later, in Section 2.6. In this section we will introduce terms and a classification scheme which allows the systematic mapping from EER relationships to relational references.

Referential Integrity

In the relational model "... relationship types [references] are not represented explicitly; instead, they are represented by having two attributes [columns] ... over the same domain included in two relations [tables]" [32, p. 174]. The *uniquely identifying* columns for a row of data in a table are called its *primary key* (PK). It is not allowed to be *NULL*[7], i.e. each row must be "identifiable" [23, p. 124]. This is referred to as the *entity integrity* constraint [23, 32]. The uniqueness property is ensured by the *key constraint* [32, p. 143]. These PK integrity rules are assumed in what follows, i.e. we do not mention them any more. Referencing a row of a foreign table is done by including columns into the table which contain foreign PK columns. This *reference* within a row is called a *foreign key* (FK). A FK with a value of NULL means that no row is referenced. If a FK belongs to the PK it is identified by a foreign table and is therefore called a *weak table*. The *referential integrity* constraint ensures that each instance of a FK contains a valid reference to an existing row in a foreign table or is NULL.

For explaining referential integrity two roles of a reference are introduced: the *PK-role* and the *FK-role*[8]. The PK-role is related to the referenced so called *Parent* table and the FK-role to the referencing so called *Child* table. An example which illustrates these terms is

[7] NULL is a marker in the database which means that no value is available. The term "NULL value" is therefore unfortunate

[8] The term "role" is borrowed from the EER model were each relationships has at least two roles

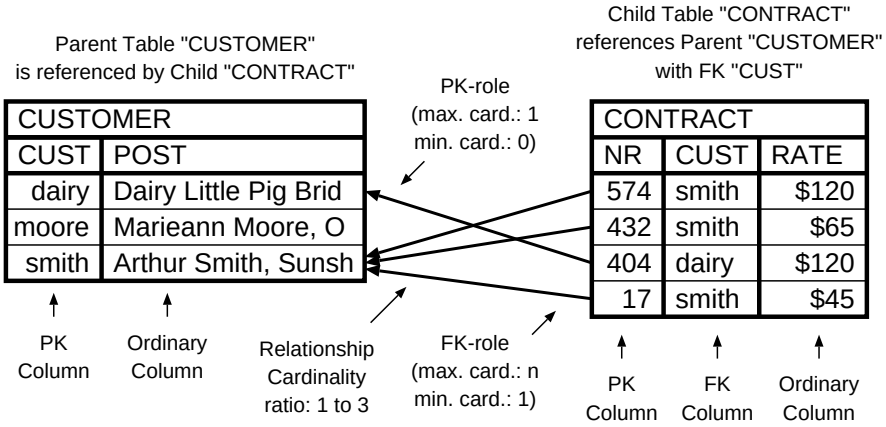


Figure 7: Example for references: A customer may have any number of contracts

PK-role	**Card.**	**How to achieve it?**
1 optional	0–1	is always ensured without IC
1 mandatory	1	each row in Parent must be referenced by at least one Child This is what [63] calls a pendant reference

FK-role	**Card.**	**How to achieve it?**
N optional	0–N	is always ensured without IC
N mandatory	1–N	is ensured by a mandatory FK (NULL not allowed)
1 optional	0–1	is ensured when each FK is unique or NULL
1 mandatory	1	through a combination of mandatory and unique FK

Figure 8: Possible cardinalities of a reference

given in Figure 7. Each table has a FK-role for each FK column[9] of the table and a PK-role for each foreign FK column which references the table. A table has rows as its instances. Likewise a reference has *reference instances*. For both roles of a reference there may be minimum and maximum cardinalities (number of reference instances) defined as depicted in Figure 7. These bounds are additional referential IC. The cardinalities of a reference are not arbitrary as in the EER model: A PK-role cannot have a cardinality greater than one, since a FK can only reference one PK[10]. *Cardinality ratios* are often simplified to *optional* and *mandatory*[11] as shown in Figure 8 where all possible simplified cardinalities for a PK-role and FK-role are listed[12]. This results in eight different cardinality ratios for a reference, even though one (*1 optional : 1 mandatory*) is redundant. Only the case *1 optional : N optional* is always ensured without IC. A cardinality 1 of the FK-role can be achieved by a unique FK, i.e. only one FK is allowed to point to a given row in a foreign table.

To summarize, one can say that referential integrity ensures that a specified cardinality ratio is not violated and that FK keys contain valid values, i.e. they must contain references to an existing foreign PK or be NULL at any point in time.

Update Propagation

Three update operations applied to a table may lead to a violation of referential integrity: the *insertion* of a row, the *modification* of a PK or FK, and *deletion* of a row. If such an update occurs which would lead to an inconsistent state of the database, action [23, p. 120ff] may be triggered to bring the database back into a consistent state:

Restrict: The update operation is aborted and the database state is restored.

[9] A PK and therefore a FK can consist of more than one column.

[10] If a PK consists of more than one column one could set a corresponding FK *partially* to NULL and hence reference a *set* of matching PK with a single reference. This is according to [59, p. 400] not allowed in "some textbooks and database products" and not considered within this work.

[11] This is also known as *existence* property

[12] Only the simplified cardinalities are used in the sequel even though the general case with arbitrary numerical cardinalities (e.g. $1 : 2\text{–}5$) would also be feasible in most cases

Cascade: The operation is executed and additional necessary updates to parents or children of the table of concern are triggered to reach a consistent state of the database. All actions are executed within one transaction, i.e. if any single operation is restricted the whole cascade is aborted.

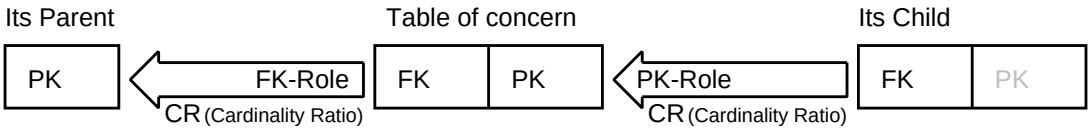
Set Null: (only applicable for PK roles) All FK of the Children are set to *NULL* when their referenced PK is removed or altered.

Set Default: (like *Set Null*, only applicable for PK roles) All FK of the Children are set to a specified *default value* when their referenced PK is removed or altered.

These actions enrich a relational reference with its real world semantics. The table in Figure 9 lists the possible combinations between PK/FK-roles, update operation and integrity ensuring action to take. It is an overview of the ways update propagation due to referential IC is possible at the relational layer. It does not consider entity integrity and the key constraint which apply to the PK-Role and where the behavior of most DBMS is to restrict integrity violating updates. It is the framework for any visualization of behavior within this work[13]. The table may also serve as a reference to classify the integrity features of a DBMS or a tool which derives relational IC from the conceptual level. It will therefore be referenced later in Sections 2.5 and 2.7. An appropriate action may be looked up for each PK/FK-role of a given table or for the PK-role and the FK-role of a given reference. These directions can then be directly executed as described in Section 2.6.

From Figure 9 it follows that cascades may not only occur in the case of *Cascade* update (annotation number 5) but also if a FK[14] of weak tables is being changed (annotation number 4) due to *Set Null* or *Set Default*. If during any cascade a propagated update is restricted then the whole cascade is aborted, i.e. undone. It is important to note that the proposed behavior is often just one way of reacting. Other ways or modifications are possible. It is

[13] additional behavior due to Supertype/Subtype cardinalities will be introduced in Section 2.6
[14] which serves at the same time as PK



Referential integrity due to FKs (FK:) and cardinality (C:)		Insert	Modify of PK or FK respectively, 2	Delete
PK[8]	(None)	FK: -	FK: Leads to inconsistency if row is referenced by child, 3	FK: Leads to inconsistency if row is referenced by child
		C: CR may be violated	C: CR may be violated	C: CR may be violated
	Restrict	FK: -	FK: Restrict updates if row is still referenced by child, 3	FK: Restrict deletion if row is still referenced by child
		C: Restrict insert if CR would be violated	C: Restrict modify if CR of new ref. would be violated, 9	C: -, 9
	Cascade	FK: -	FK: Change all FK in children which ref. row, 3, 4	FK: Delete all children which reference row, 5
		C: Insert a row in child if the reference is mandatory, 5	C: Insert a row in child if the ref. is mandatory, 5, 9	C: -, 9
	Set Null	FK: -	FK: Set all FK in children which ref. old PK of row to NULL, 3, 4	FK: Set all FK in children which ref. row to NULL, 4
		C: -	C: Restrict modify if the above is not allowed due to the CR	C: Restrict modify if the above is not allowed due to the CR
	Set Default	FK: -	FK: Set all FK in children which ref. row to their default, 3, 4	Set all FK in children which reference row to default, 4
		C: -	C: Restrict modify if the above is not allowed due to the CR	C: Restrict modify if the above is not allowed due to the CR
FK[8]	(None)	FK: Leads to inconsistency if row in parent does not exist	FK: Leads to inconsist. if new row in parent does not exist	FK: -
		C: CR may be violated	C: CR may be violated	C: CR may be violated
	Restrict	FK: Restrict insert if referenced row in parent does not exist	FK: Allow change only if new ref. row exists in parent, 7	FK: -
		C: Restrict insertion if CR would exceed maximum	C: Restrict modification if either the CR of the new or the old ref. would be violated	C: Restrict deletion if CR would fall below minimum
	Cascade	FK: Insert a row in parent, if row in parent does not exist, 1, 5	FK: Insert a row in parent, if new row in parent does not exist, 5	FK: -
		C: Only to restrict an insertion is appropriate if CR exceeds its maximum	C: Combination of (1) a deletion (see right) and (2) an insertion (see left) in one transaction	C: Delete row in parent if ref. is mandatory and this is the last FK reference to it, 5

Figure 9: "Four dimensional" update propagation chart, for footnotes see Figure 10

1. Cascaded insert is only applicable in special cases, when additional data can be prompted or all attributes are optional.
2. In the case of dependent tables attributes can serve as PK and FK at the same time, i.e. both rules are applicable.
3. It can be questioned if allowing a change of a PK is a good idea at all.
4. When FK is part of PK (dependent table) a change to a FK implies change of a PK, i.e. cascades may be necessary.
5. The changes to a parent or a child may cause further action, i.e. cascades are possible.
6. Unique PK are a property of most DBMS and don't need extra care, i.e. restrict is the default build-in action of the DBMS.
7. If reference is non-transferable (change of parent not allowed) each change of FK is restricted, even if new FK is valid.
8. PK or FK may encompass more than one attribute.
9. There is no cardinality problem with a deletion of a row connected to a PK-role, because the reference will disappear.

Figure 10: Footnotes for Figure "Update Propagation Chart"

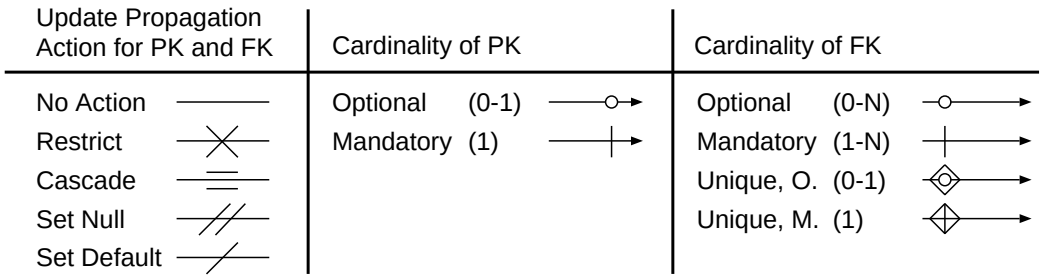


Figure 11: Graphical notation of update propagation action and cardinalities

mentioned in [32, p. 151] that "modifying a PK value is similar to deleting one tuple and inserting another in its place". The field *Cascade-Modify* in Figure 9 offers an alternative, namely to change the FK instances in the child table due to a PK modification.

A generally agreed abstract notation for the relational model does not exist. Each tool which allows modeling or reports at the relational level has its own textual or graphical notation, e.g. [53, 61]. A concise graphical notation of the abstract relational data model which can be used to visualize the schema is needed for this work. The update propagation properties (restrict, cascade, set null and set default) should be visible within the schema. This requires the definition of a new notation which is depicted in Figure 11. The symbols for *Cascade* and *Set Null* are borrowed from a notation proposed in [5]. The cardinality symbols are the same as in the EER crow's foot notation [36]. Additional is the diamond shaped *Unique* symbol[15] introduced as a property of FK roles. The update propagation symbols are only valid for one type of update (either insert, modify or delete) at a time. A

[15] Mnemonic: unique like a diamond

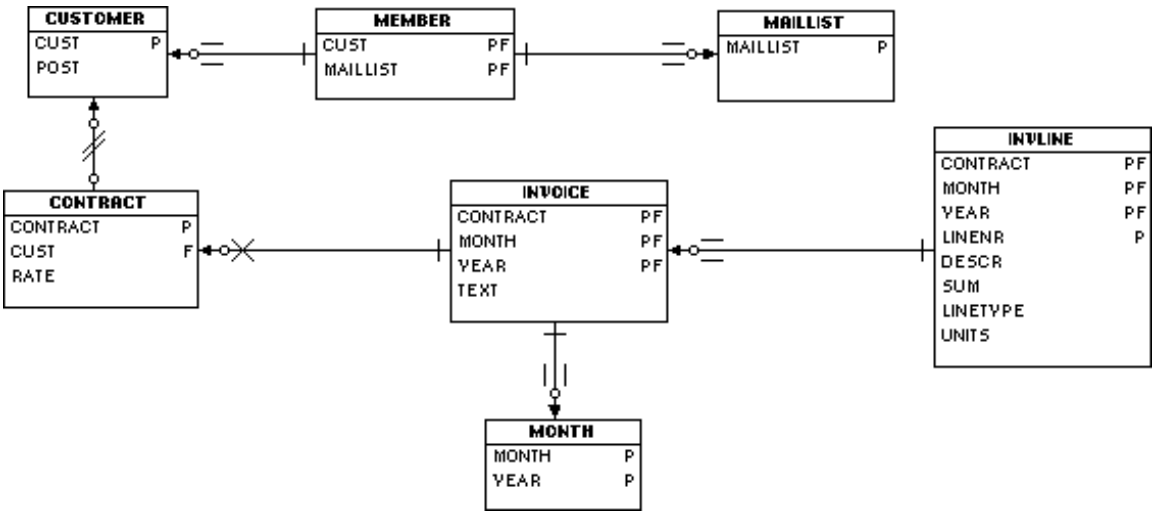


Figure 12: Relational model with symbols for propagated *Deletions*

PK-role may, for example, restrict an update but cascade a deletion. They may be specified for FK-roles or PK-roles. On the other hand cardinality ratios are properties which are independent of database operations. Figure 11 may be compared with Figure 8. Finally, the complete relational schema of the telephone company example is shown in Figure 12. The update propagation symbols depicted are for propagated deletions. How to implement the mentioned IC will be discussed in Section 2.5 on SQL and Section 2.6 on the mapping of the layers.

2.5 Database Language SQL

This section will discuss the implementation of integrity constraints in today's relational DBMS. Meaningful use of these features is the aim of the preceding design steps. It should be noted here that explicit visualization of processes within a DBMS require that the behavior is modeled without using the mechanisms provided. Otherwise one would only see the effects, but not the step by step behavior. As mentioned the abstract relational layer will be used for simulation of active model behavior.

The most widespread database language is SQL[16], the "intergalactic dataspeak" [68]. SQL is being standardized by ISO[17] and a subset of the language is implemented in nearly all commercial relational DBMS products[18]. It includes a language for data definition (DDL) and data manipulation (DML). Early SQL standards (SQL-86 and SQL-89) were trying to capture what was available in commercial implementations of that time. The current standard, SQL-92 [49], goes beyond that and defines features which are not implemented yet. This is to avoid divergence of the concurrent development of the various DBMS vendors. Due to this standardization effort a database system can be changed during the lifetime of a database application without major changes to the application code. The decision for a certain SQL-server may therefore be made for economic and technical (e.g. scalability, performance, security) reasons. SQL is a declarative language which can be used interactively statement by statement or embedded in a conventional procedural *host language*. The latter is the most common application of SQL and some SQL features can only be used in conjunction with a host language. This causes an "*impedance mismatch* between a procedural programming language and an embedded DML with declarative semantics" ([67], quoted in [23, p. 670]).

This has a consequence with respect to the focus of this work. It is common practice to model a database at the conceptual level (see Section 2.3) using some design tool (see

[16] Its full name is *Database Language SQL*. "SQL" is not an abbreviation for "Structured Query Language" and not pronounced "sequel"

[17] Joint technical committee ISO/IEC JTC1/SC21/WG3

[18] Except for some small PC-based systems

domain	`CREATE DOMAIN`
column	`NOT NULL`
	`NOT NULL UNIQUE`
	`DEFAULT NULL`
table	`PRIMARY KEY`
	`FOREIGN KEY REFERENCES`
	`UNIQUE`
	`CHECK`
database	`CREATE ASSERTION`
	`CREATE TRIGGER`

Figure 13: Classification of SQL-92 integrity enforcement statements

Section 2.7). The tool will probably be able to generate SQL table definitions. For an applications programmer these table definitions are of primary interest because this is the level of abstraction for embedded queries in the programs. Code generation in general is often a one way process where an executable representation is computed. This is fundamentally different with database design where all levels (EER, abstract relational and SQL) are of interest. Therefore the mapping in both directions between the different levels as described in Section 2.6 is of importance when data residing in SQL tables should be interpreted and displayed at the conceptual level.

After having discussed how SQL is being used, an overview of its available integrity features is given. They may be used to implement the concepts of the previous section (see classification scheme in Figure 9).

Integrity Constraints in SQL-92

The aim of enriching a database language like SQL with IC features is "capturing more of the meaning of the data, while preserving the independence of implementation", [17]. The predecessors of the relational model had inherent support for *referential integrity* due to their built-in access paths, which was the reason for their data dependence[19]. How to ensure

[19] See [16] for a discussion of the predecessors of the relational model and the problems with data dependency.

```
# If row in referenced CUSTOMER is deleted or updated this operation
# will be propagated to all referencing CONTRACTs. All updates to
# column RATE which would violate the CHECK statement are restricted.

CREATE TABLE CONTRACT
(
        CONTRACT INTEGER   NOT NULL,
        CUST     CHAR(7)   NOT NULL,
        RATE     MONEY     NOT NULL,

        PRIMARY KEY ( CONTRACT ),
        FOREIGN KEY ( CUST ) REFERENCES CUSTOMER
                             ON DELETE CASCADE
                             ON UPDATE CASCADE,
        CHECK ( RATE > 0 AND RATE < 450 )
)


# The total amount of payments for all CONTRACTs of a CUSTOMER may not
# exceed the defined limit in CUSTOMER.MAXPAYMENT

CREATE ASSERTION
(
        NOT EXISTS
        (
                SELECT SUM ( CONTRACT.RATE ) AS PAYMENT
                FROM CONTRACT, CUSTOMER
                GROUP BY CONTRACT.CUST
                WHERE CUSTOMER.CUST = CONTRACT.CUST
                  AND PAYMENT > CUSTOMER.MAXPAYMENT
        )
)
```

Figure 14: Example for the usage of SQL-92 IC statements; taken from the scenario in Figure 7

```
FOREIGN KEY [ ( column-name {, column-name } ) ]
REFERENCES [ PENDANT[21] ] table-name [ ( [ column-name {, column-name } ][22] ) ]
[ MATCH ( FULL | PARTIAL ) ]
[ ON UPDATE ( NO ACTION | RESTRICT[23] | CASCADE | SET NULL | SET DEFAULT )
[ ON DELETE ( NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT )
```

Figure 15: SQL-92 / SQL3 / SQL4 *referential constraint definition* in EBNF

these semantics on the relational level, and which additional constructs are needed, was discussed in the research community during the seventies and eighties and influenced the SQL standardization process. The IC of SQL-92 will be introduced below, but the discussion of their application to implement integrity features of the foregoing section will be explained later in the section on mapping (see Section 2.6).

A classification scheme for IC in SQL is to distinguish domain-, column-, table- and database constraints as in [23, p. 444]. Figure 13 lists the features available in SQL and Figure 14 shows the usage of some features in an example related to the telephone provider UoD.

The SQL-92 data types serve primarily as *domain* constraints. The user can add new domains to the system[20]. *Column* constraints are `NOT NULL` and `UNIQUE`.

Of interest for this work are the *table* constraints which can be specified over a set of columns to implement *referential integrity*. The `FOREIGN KEY` construct declares columns used for referencing rows of other tables through their primary keys and is therefore the primary means for referential integrity in SQL. As shown in Figure 15 can be declared what action to take if referential integrity is violated. A clause for a propagated deletion (`ON DELETE CASCADE`) is shown exemplary in Figure 14.

`CHECK` allows the specification of table invariants which must hold at any time, otherwise

[20] See [23, p. 220] for a discussion on the limitations of SQL-92 domains
[21] `PENDANT` introduced in SQL4
[22] Only optional with SQL4
[23] `RESTRICT` introduced in SQL3 (and SQL4)

an update operation will be restricted. A *database* constraint would be the checking of arbitrary invariants of the database state including any number of tables. These might go even beyond entity and referential integrity. Figure 14 shows a `CREATE ASSERTION` as an example for the SQL-92 support. Each of the above mentioned constraints have to be implemented separately by a DBMS. This is not a trivial task and it turns out that, e.g. assertions are not yet implemented by any DBMS vendor even though they are regarded as an important feature. Figure 16 is derived from Figure 9 and shows the declarative IC features of SQL which may be specified with the `FOREIGN KEY` constraint. The table was split into valid foreign key and cardinality issues. The pendant concept appeared in the ANSI working papers [64] and [63] from 1986. It describes how to handle mandatory PK-roles. This has been taken up by the ISO standardization committee and appears in the current SQL3/SQL4 draft [50] as an extension of the `FOREIGN KEY` statement as shown in Figure 15.

The reader may note the limitations of declarative SQL integrity support. The lower table shows just the statements to *restrict* violation updates. The field for a optional one FK-role cannot be implemented with the SQL `unique` statement. And the pendant concept will probably not available within the next ten years. If these integrity features should be implemented today and if the empty fields of the table should be filled, a more generic approach is needed.

SQL Triggers

"Specifying the test for the non-procedural constraint is the simplest part of the job. The hard part is specifying the course to take if the test fails; clearly a *procedural* approach is needed for this", [59, p. 412]. The use of *procedural table constraints* allows the programmer to specify the semantics of the checks and the control flow alternatives in case of an integrity violation. The conventional way to do this is to write procedural code in the host language that either polls the database at a certain frequency or does checks before or after each embedded update statement. Both ways have their drawbacks. Frequent polling puts unnecessary load on the

Referential integrity using `FOREIGN KEY`		Insert	Modify of PK or FK respectively	Delete
PK-Role	Restrict	-	`ON UPDATE NO ACTION`	`ON UPDATE NO ACTION`
	Cascade	-	`ON UPDATE CASCADE`	`ON DELETE CASCADE`
	Set Null	-	`ON UPDATE SET NULL`	`ON DELETE SET NULL`
	Set Default	-	`ON UPDATE SET DEFAULT`	`ON DELETE SET DEFAULT`
FK-Role	Restrict	`MATCH FULL/PARTIAL`	`MATCH FULL/PARTIAL`	-
	Cascade			-

SQL statements to restrict the violation of Cardinality		1	N
PK-Role	Optional	no IC needed	-
	Mandatory	`PENDANT` (SQL4)	-
FK-Role	Optional	UNIQUE with any number of NULLs allowed	no IC needed
	Mandatory	`NOT NULL UNIQUE`	`NOT NULL`

Figure 16: Classification of declarative SQL IC support using Figure 9

```
CREATE TRIGGER trigger-name BEFORE | AFTER
INSERT | DELETE | UPDATE [ OF column-name {, column-name } ]
ON table-name
[ REFERENCING old-or-new-values-alias-list ]
[ FOR ( EACH ROW | FOR EACH STATEMENT ) ]
[ WHEN ( search-condition ) ]
statement | BEGIN ATOMIC statement ; { statement ; } END )
```

Figure 17: SQL3 trigger syntax in EBNF

DBMS and fails to ensure the correct state of the database at each point in time. Checking at each update leads to redundant code scattered in different parts of the software which makes maintenance, extension and changes difficult. *Event-Condition-Action* (ECA) rules were introduced by the HiPAC project [11] to overcome these problems. They have been adopted by many research projects. These rules are stored in the database and "fire" when a certain *event* like an update to a specific table occurs. Then the *action* part of the rule is executed, if an arbitrary *condition* of the database state holds. Allowing ECA rules without further restrictions may lead to non-determinism and since rules may cause other rules to "fire", termination cannot be guaranteed [25]. For these reasons the SQL standardization group hesitated to incorporate ECA rules in the current standard SQL-92. The *trigger* construct is a proposal for the next standard (SQL3, currently under development) which is, according to [47], expected for 1998.

> The standards committees simply guessed wrong: they didn't realize how rapidly the demand for triggers and the implementations of triggers would come along [...] However, most vendors have simply gone beyond SQL-92 and used the SQL3 trigger specification as the basis for their implementation. [56, p.388][24]

The actual implementations of triggers in commercial database servers differ, because there is still no standard available. We will use in the following the semantics of the SQL3

[24] This quote also sheds some light on the way SQL standardization works: the "final vote" on features is cast by the members. Some are vendors, whose customers "vote" with their dollars.

draft standard [50], which differs considerably from current implementations as depicted in a comparison table in [79, p. 258].

> The issue of interactions between triggers and referential integrity is an important one, dealt with in the proposed SQL3 standard by prohibiting triggers on tables with referential integrity constraints. However; most commercial systems do not (yet) have such a strict restriction [79, p. 244]

The SQL trigger as proposed by the SQL3 standardization group (see Figure 17) is more limited than ECA rules as proposed in [11] to avoid the above mentioned problems. For each triggering event (*insert, delete, modify*) and each table only one trigger before and one after the update is allowed to avoid non-determinism due to arbitrary ordering of trigger invocations. It is also not permitted to alter a table more than once during one triggering sequence also known as self-triggering. In this way the termination problem is solved. These restrictions are conservative and will in some cases prevent the use of triggers, but they will be no problem for the intended purposes within this work. The trigger definition includes procedural statements for the action part or a call to a SQL *stored procedure*. The trigger granularity (`FOR EACH ROW` or `FOR EACH STATEMENT`, see Figure 17) must always be `FOR EACH ROW` for update propagation purposes. If the commit statement of a transaction could also be a triggering event *transactional consistency* could be also ensured with triggers, "but at present it is unclear if a new standard will offer this capability" [59]. This would be nice, since some conceptual constraints can only be enforced if the checks could be deferred to the end of a transaction as shown in Section 2.6.

Since triggers are stored in the database independent of applications, they may be automatically generated. This is a main reason why triggers became a popular means of IC (see Section 2.6), but they are not limited to IC. So-called *business rules* are derived from the application domain but are supplementary to the semantics of the conceptual database schema. To efficiently maintain a large number of these rules is a well known problem [25, 37, 27, 6, 3]. Debugging and visualization of these rules is not a problem of concern in this work but is an

active research field in the context of active databases and workflow modeling.

The maintenance of integrity with procedural triggers is, on the other hand, very central since it allows the implementation of advanced referential integrity which was discussed in Section 2.4. In this context trigger events (see Figure 17) include insertions, modifications and deletions of rows. The condition part of a trigger checks if integrity was violated by the event. Integrity maintaining actions (*Restrict*, *Cascade*, *Set Null* or *Set Default*) can be specified in the procedural action part of a trigger statement.

An example of a triggered procedure which was generated to ensure a valid FK after an insertion is shown in Figure 18.

At this point the three layers of modeling are introduced:

- the conceptual layer using EER modeling
- the abstract relational layer using our own notation and
- the physical layer using the database language SQL.

We need mappings between these layers with respect to structure and semantics. This will be discussed in the following section.

```
--  Insert procedure "pi_contract" for table "CONTRACT"
create procedure pi_contract(new_cust char(10))
    define  errno    integer;
    define  errmsg   char(255);
    define  numrows  integer;


    --  Parent "CUSTOMER" must exist when inserting a child in "CONTRACT"
    if new_cust is not null then
       select count(*)
       into   numrows
       from   CUSTOMER
       where  CUST = new_cust;
       if (numrows = 0) then
          let errno  = -1002;
          let errmsg = "Parent does not exist in ""CUSTOMER"".
                       Cannot create child in ""CONTRACT"".";
          raise exception -746, 0, errmsg;
       end if;
    end if;

end procedure;

-- Insert trigger "ti_contract" for table "CONTRACT"
create trigger ti_contract insert on CONTRACT
referencing new as new_ins
   for each row (execute procedure pi_contract(new_ins.CUST));
```

The trigger prevents the insertion of rows with an invalid FK, i.e. where no row in the foreign table matches the FK. The example is related to Figure 7 and was generated by S-Designor for an Informix database server. The DDL code creates a stored procedure and a trigger which calls the stored procedure on each insertion to table `CONTRACT`.

Figure 18: Example for an SQL trigger

EER-Model	**Relational Model**
Entity	Table
Weak entity	Table with PK that incl. a FK to identifying table
1:1 Relationship	Merge of ent. into one table if rel. is tot., otherwise FK
1:N Relationship	FK in table on the N-side
M:N Relationship	Relationship table with two foreign keys
n-ary Relationship	Relationship table with n foreign keys
Simple attribute	Column
Composite attribute	More than one column
Multi-valued attribute	Table and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key
Specialization	Four options A, B, C and D as described in [32, page 629]
Generalization	Same as specialization
Category	super-classes reference category by a surrogate key [32, page 632]

Figure 19: Correspondence between EER and relational models

2.6 Mapping Between Layers

This section discusses the mapping of the conceptual EER model to an abstract relational model and in turn to the database language SQL. The EER features as depicted in Figure 4 are implemented at the relational level using the constructs as listed in Figure 19. A central point of this work is to allow not only a mapping from EER to relational, but also its inverse. Commonly only the forward mapping from the EER to the relational model is considered. Appropriate structures for storing this information will be developed within this work (see Chapter 4).

The mapping can be split into two parts: (1) the structural mapping from entities in the EER model to tables in the relational model and (2) the enhancement of the relational model with IC to reflect the inherent constrains in the EER model. The latter is often disregarded in the database literature, e.g. in [32].

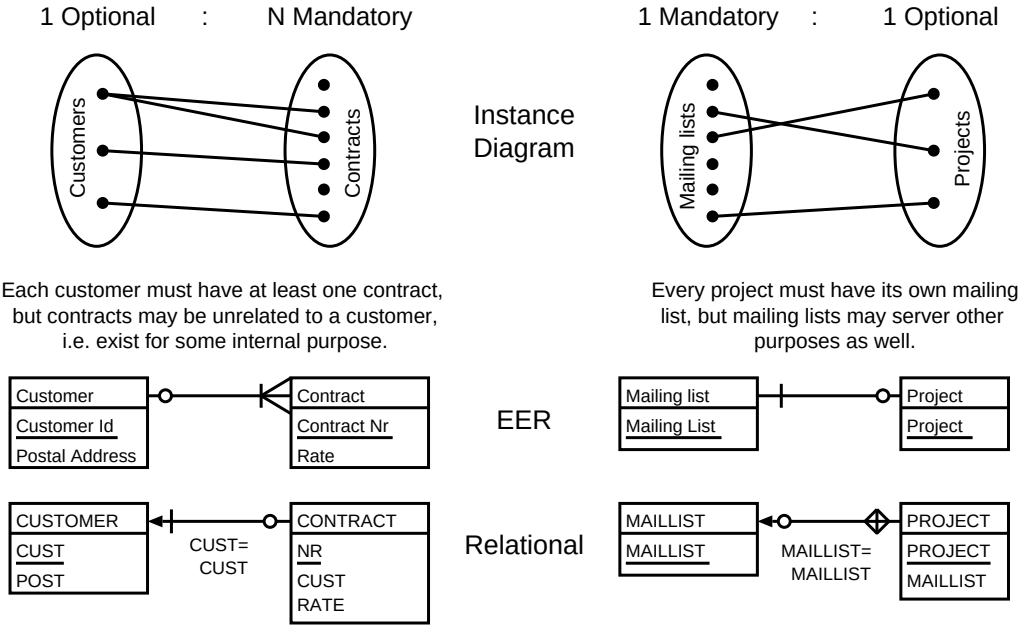


Figure 20: Examples to illustrate the mapping of EER relations to FK references.

Structural Mapping

The structural mapping of entities and basic ER relations is described, for example, in steps one to seven in [32, pp. 172–177]. Other published mapping formalisms exist. All entities become tables and relations are modeled as relational references based on FK. The two examples in Figure 20 are included for better understanding of the mapping of relations. Note how the cardinality ratio is related to the EER relationship and how it is transformed to mandatory, optional or unique reference roles. The cardinality symbols change the side due to the look-across cardinalities (see Section 2.4) of the IE notation. The mapping of N:M relations requires a new associative table with FK to the participating tables. 1:1 relations are discussed later in this chapter. An overview is given in Figure 19, compare [32, p. 177]. The "normal forms (NF[25]) based on primary keys" [32, page 407] have to be considered when designing the relational level. Fortunately "using the entity-relationship model we can arrange data in a form similar to 3NF relations" [14], i.e. no additional restructuring of

[25]In the following NF stands for *normal form*, e.g. 3NF denotes the third normal form

Generated Tables		Cardinality of Subtypes	Parent and Children A	Children B	Parent C	Parent D
disjoint Only one subtype instance per supertype instance is allowed.	**total**	**1**	√ Integrity constraints for ensuring the relationship cardinality are needed	√ PK of (1) is only allowed in either (1,2) or (1,3)	√ if (t) is not allowed to be NULL	
	partial	**0-1**	√	If a row in (1) has no subtype in (2) or (3), it is lost	√ if (t) is allowed to be NULL	
overlapping Each supertype instance may have more than one subtype instance.	**total**	**1-n**	√	(√) Causes redundancy: Attributes of (1) are in (1,2) and (1,3)		√ If at least one boolean (t) is always true
	partial	**0-n**	√ is always ensured without IC	If a row in (1) has no subtype in (2) or (3), it is lost		√ is always ensured without IC
Note			only the PK of (1) is inherited and used as FK.	Equi-Join operation build into the Schema	(t) is called specifying attribute. It might be replaced by an attribute of (1).	(t) are booleans and state whether (2) or (3) contain data
Usage			Storage efficient, no NULL values due to missing subtypes. Integrity constrains are necessary to ensure total or disjoint subtypes.	Only for disjoint and total relationships. A view is necessary to access all entries of (1) without join.	More runtime efficient and less storage efficient (NULL values), since the outer join is build into the schema.	
Results of outer join						
Results of equi-join						

Figure 21: Mapping of Supertype-Subtype relationship to a relational representation

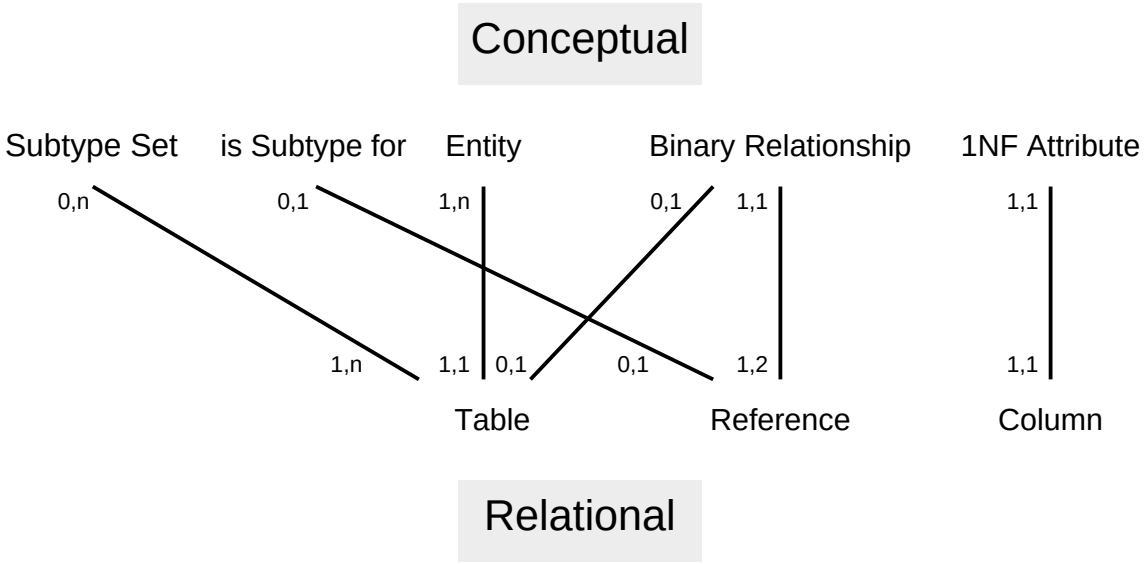


Figure 22: Mapping relations

tables is necessary to ensure 3NF. The mapping of the EER construct *Supertype/Subtype relationship*[26] requires choices to be made by the designer (step eight; cases A to D, [32, page 629]). Four ways of mapping a Supertype/Subtype relationship are shown in Figure 21. The mapping choice will depend on the properties, i.e. *disjoint* or *total* and on efficiency criteria as annotated in Figure 21. If the EER notation allows the specification of domains, they must be mapped to the available set of types in the target DBMS. The mapping relationships are depicted in Figure 22 and an example for a mapping of the Telephone Provider UoD is given in Figure 23.

Behavioral Mapping

The derivation of relational IC from the inherent structural properties of the EER model should ensure that most of the semantics at the conceptual level are preserved at the relational level. A helpful categorization[27] in this context identifies three groups of how an IC is

[26] The often found terms *Specialization* and *Generalization* refer to ways of defining a Supertype/Subtype relationship according to [32, page 613]

[27] orthogonal to the categorization mentioned in the previous section: domain-, column-, table- and database constraints

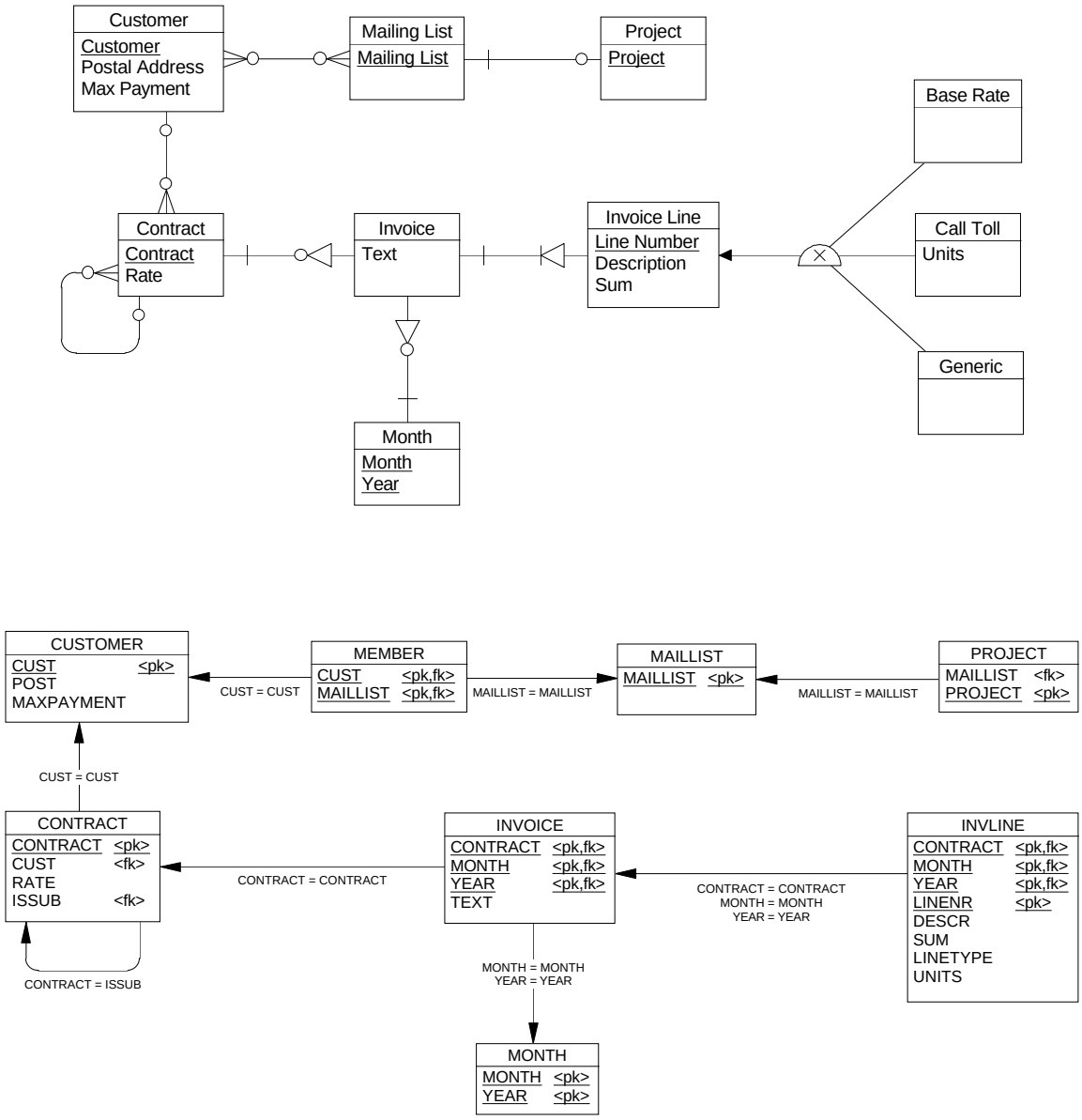


Figure 23: Telephone Provider UoD mapped from EER to relational

supported by a data model: *inherently*, *implicitly* and *explicitly* [32] as depicted in Figure 24. "The distinction between inherent and explicit constraints depends on the structures provided by the data model.", [28].

"*Inherent* constraints are those rules that are implied by the basic constructs of the model and reflect the nature and the semantics of the data model itself" [28]. These constraints need not be specified using a DDL. The distinction between relations and entities and therefore referential integrity is inherent to the EER model. The notion of a key and its uniqueness, i.e. entity integrity and key constraint can be seen as an inherent property of the relational and the EER model. Additional semantics which can be specified using a DDL are called, according to [32, page 642], "implicit constraints" like declaratively specified referential integrity (SQL `FOREIGN KEY`, Figure 24).

"*Explicit* constraints are those rules that are not embedded in the basic structures of the data model and explicitly stated to handle application dependent properties of data", [28]. They can be implemented either using a procedural host language within database transactions, by triggers or "via an assertion specification language" [32, page 642].

Inherent IC of the EER model (abstract properties of the model like for example cardinalities), have to be mapped to abstract relational concepts which are concerned with the procedural assurance of integrity as shown in Figure 9 on page 9. These abstract relational concepts have to be in turn mapped to implicit or explicit relational IC using a given set of DDL statements. It is often not easy to judge which kind of relational DDL IC to use to model a given abstract constraint and it is often questionable if it should be modeled as inherent, implicit or explicit. Much has been written on this topic [28, 23, 17]. Some considerations are marked with numbered gray arrows in Figure 24. The numbers in the following paragraphs refer to these numbers.

1. Codd [17] proposed that referential integrity should be an inherent rather than an implicit part of the relational model. This is not considered a good idea in [22] because it does not allow for specification of what should be done in case of a violation.

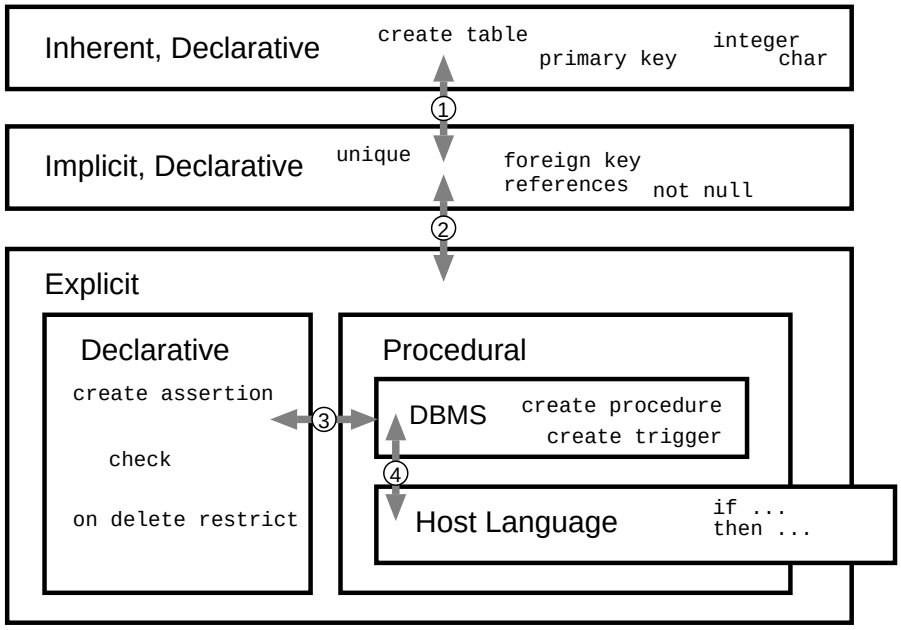


Figure 24: Classification of IC with SQL examples

2. Explicit directions on how to ensure IC are necessary when the action to take on a violation of an IC should do more than restrict an update.

3. Of more importance to this work is whether relational constraints are modeled with a set of declarative constraints or in a procedural fashion using SQL triggers. Triggers could be the only mechanism for integrity in a database system All previously discussed IC can be modeled by triggers. There is support for the use of rules (triggers) as a single simple means to implement constraints. A limited number of implemented database concepts simplifies the design of DBMS, especially the design of query optimizers. Although this is more of a theoretical DBMS design issue it could be also a good strategy for a case tool to use only certain kinds of integrity constraints for a particular DBMS. This issue is addressed in Section 2.7. Of course, if advanced referential integrity should be implemented (see Figure 9) the use of triggers is in some cases inevitable. Since procedural triggers may become quite lengthy the use of tools for automatic generation of triggers might be necessary (see Section 2.7). The performance of the resulting

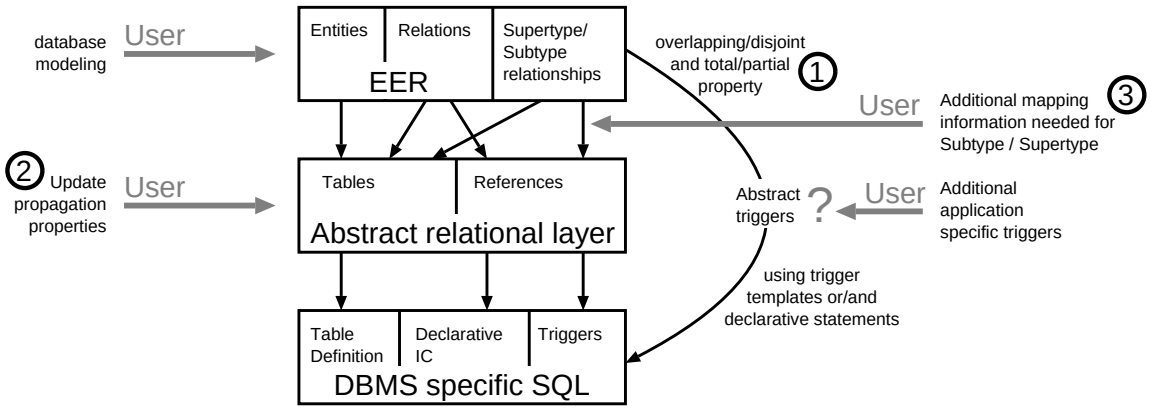


Figure 25: Three layer mapping problems and possible solutions

database application is also an important issue [27] in this discussion but is not treated here.

4. If constraints are "buried" inside the host language of application programs they can't be automatically generated and even worse in this context they can't be explicitly shown with a visualization tool. Other drawbacks were mentioned in Section 2.4. For these reasons integrity maintaining code in the application programs should be avoided in favor of IC in the DBMS whenever possible.

Problems with the three layered approach

Our way of mapping is to convert the EER schema to an abstract relational and in turn to a target DDL like SQL. The ideal would be that an automatic mapping procedure generates executable SQL statements which reflect all the semantics of the EER schema. Unfortunately (1) some semantics may be lost during the mapping, (2) not all properties of the relational model can be derived from the EER schema semantics and (3) mapping can't be done unambiguously. These points are depicted in Figure 25 and will be explained in the following:

1. Some EER notations make extensive use of the specification of mutually exclusive (disjoint) relations and overlapping relations[28]. They are at least used for Subtype/Supertype relations (see Figures 4 and 21). These role constraints (AND, OR, XOR) pose a problem to the strictly three layered approach, since they cannot be expressed using only tables and references when the mapping is done according to case (A) in Figure 21, since the cardinality restrictions include more than one relational reference unlike normal cardinality ratios. A solution to this problem would either be to generate IC DDL statements directly from the EER level, bypassing the abstract relational level, or to extend the abstract relational level with means of storing this information. An abstract trigger specification which makes use of trigger templates to generate triggers might be feasible, as sketched in Figure 25. An entire M.Sc. thesis [1] is devoted to this issue.

   Another issue is that the mandatory-many role as depicted on the left hand side of Figure 20 cannot be ensured even by triggers because it requires a temporal violation of IC when (1) a line in "CUSTOMER" is inserted and (2) the corresponding referencing row in "CONTRACT" is inserted. If no transaction triggering event is available[29] this has to be programmed into an application transaction, resulting in the above mentioned drawbacks. The declarative pendant concept [63] (see Figures 15 and 16) appears to be a partial solution to the problem, since it is limited to restrict violating updates.

2. EER notations usually[30] do not allow the specification of update propagation properties. This information has to be added to the relational model. Meaningful defaults may be derived from the EER relationship semantics as proposed in [29]. Some update propagation actions which could be automatically derived from Subtype/Supertype relations are listed in [32, p. 619] and [1, p. 66].

[28] An example using the notation of Information Engineering is described in [36, p. 69–81]
[29] not available in SQL-92, see Section 2.5
[30] See [53] for a tool which allows this

3. Ambiguous mapping can be a problem when naming conflicts occur while FK attributes are migrated during mapping. Most tools provide an automatic renaming scheme of some kind to tackle the problem. Another problem for ambiguous mapping is the case of a Supertype/Subtype relationship. The designer has to choose one of the mappings depicted in Figure 21 as already mentioned. An often overlooked problem is the mapping of 1:1 relationships, described in detail in [24]. The mapping depends on one of the three possible cases[31] and on the cardinality[32] of the tables involved. In order to avoid NULLs one should place the FK which constitutes the 1:1 relation in the smaller table, i.e. in the table with less rows. Another choice is to "merge the two entity types and the relationship into a single relation [table]" [32, p. 173]. This information has to be provided by the designer as well. Other aspects and ways of mapping 1:1 relationships are discussed in [24].

Despite these problems all but the above can be carried out step by step as briefly described and depicted at the beginning of this chapter. Therefore automation of this process is highly desirable.

[31] (*1 optional : 1 optional*) or (*1 mandatory : 1 optional*) or (*1 mandatory : 1 mandatory*)
[32] Estimated number of rows in a table

2.7 Modeling Tools

To efficiently handle the mapping of layers software tools can be employed. If a design task can be done step by step, i.e. an algorithm can be found, then it should be done with the aid of design programs. This saves time and money because it frees the developer from routine tasks and reduces the probability of faults. Consistency checks are also necessary and feasible to detect, e.g. circular dependencies, isolated entities or incomplete naming [61, p. 126]. Automated mapping is valuable, if the EER level should be used in the ongoing modeling process, i.e. when the schema is modified during the lifetime of a database application.

It is therefore not surprising that such tools are available commercially and as research prototypes. They allow the user to graphically assemble a conceptual data model in some notation which often borrows from one of the main EER notations as described in Section 2.3. A comparison of some available tools can be found in [69]. The article evaluates the tools with respect to their integrity constraint support. Other reviews and comparisons appear frequently in periodicals [26, 21, 20].

A general problem with modeling tools is that they may limit the freedom of the designer if they do not allow the generation of desired output. Tools are of limited value, if generated code has to be altered manually. The tools should also reflect and use all relevant features of a target DBMS. This often requires a close cooperation between DBMS vendors and design tool developers[33].

S-Designor modeling tool

In this work the tool S-Designor [74] is used. It is based on the notation used in Information Engineering (see Section 2.3). It allows the user to model not only in the EER, called *Conceptual Data Model* (CDM), but also in the relational model which is called *Physical Data Model* (PDM). Detailed documentation, including graphical schemas, at this level is of

[33] In fact Powersoft (a design tool vendor) is owned by Sybase (DBMS), and Oracle (DBMS) sells its own modeling tool

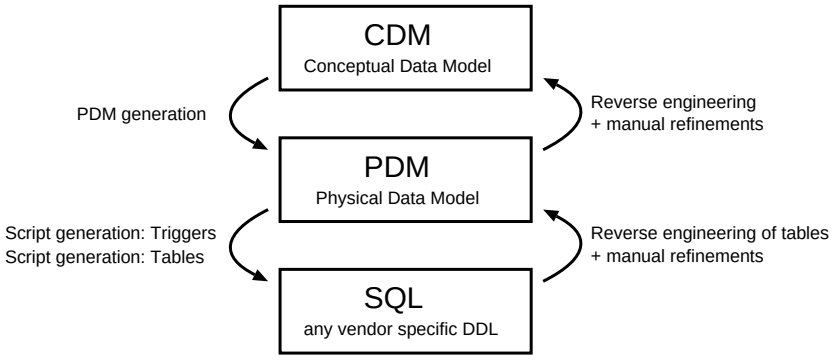


Figure 26: Layers and terms with S-Designor

great value for application programmers. S-Designor tries to keep the layers synchronized, i.e. to preserve changes in either layer. The mapping procedures must be started explicitly and work in both directions (see Figure 26). They perform, consistency checks, detect circular dependencies and rename attributes as necessary. The relational level is called the *Physical Data Model* (PDM) and is already database dependent, because it includes the data types of the target DBMS and restricts design choices, e.g. when certain IC features are not available in the target DBMS. Changing the target database at the PDM layer is, however, possible. The PDM and CDM are stored separately in the file system or in a repository which may reside in "any external RDBMS [Relational DBMS] which is accessible via ODBC [Open Database Connectivity]". This strict separation between conceptual design and physical implementation was a major reason for choosing the tool for this work.

It should be mentioned that another major reason for choosing S-Designor was the transparent way the tool saves its models to disk as plain text in a relational fashion. Also the complete definition of target DBMS is stored as plain text documents which can be modified or written from scratch as needed.

Integrity features of S-Designor

S-Designor lets the user choose whether referential IC should be ensured by declarative table constraints (see Section 2.5) or whether the same functionality should be implemented by

Referential integrity support of S-Designor		Insert	Modify of PK or FK respectively	Delete
PK	Restrict		Trigger or declaratively: `ON UPDATE NO ACTION`	Trigger or declaratively: `ON DELETE NO ACTION`
	Cascade		Trigger or declaratively: `ON UPDATE CASCADE`	Trigger or declaratively: `ON DELETE CASCADE`
	Set Null		Trigger or declaratively: `ON UPDATE NULLIFY`	Trigger or declarativley: `ON DELETE NULLIFY`
	Set Default		Trigger or declaratively: `ON UPDATE DEFAULT`	Trigger or declaratively: `ON DELETE DEFAULT`
FK	Restrict	Trigger	Trigger	
	Cascade			
	Additional Integrity		Trigger restricts updates to "non-modifiable" columns FK-Restrict: Trigger restricts a change of FK (Parent) when "change of Parent" disallowed	

Figure 27: Classification of S-Designor's build-in referential IC support due to valid FK values, c.f. Figure 9

triggers. Triggers are needed anyhow if all available IC features of S-Designor are used. Figure 27 gives an overview of what kinds of IC are supported. S-Designor is limited to the assurance of valid FK values. It fails to guarantee the cardinality ratios which were specified at the conceptual level. This information is already partially lost at the PDM level[34], since all that is left is the structure of tables and their FK plus information concerning whether a FK is mandatory (SQL: `NOT NULL`), i.e. cardinality ratios are only used to build the structure of the tables, but not for additional referential integrity. Due to the mapping problems mentioned in Section 2.6 the overlapping/disjoint property of a Supertype/Subtype "is for documentation only and has no impact in generating the PDM", [61, p. 111]. The mapping decision for Supertype/Subtype relationship is a property of the CDM layer with S-Designor, unlike the mapping of 1:1 relationships. The way S-Designor handles this particular problem, mentioned in Section 2.6, is to generate two references in both directions [61, p. 142] and leaves it to

[34] No distinction is made between N Mandatory and N optional

the user to delete one of the references in the PDM[35]. Further, S-Designor does not permit the definition of *1 Mandatory : 1 Mandatory* relations. The motivation for this modeling restriction is not apparent. Update propagation properties have to be specified at the PDM layer. This impairs the strict logical layering and could have been avoided by supplying the information at the conceptual level as shown and implemented in [53]. Anyhow, the update propagation properties in the PDM layer are preserved during subsequent mappings. A major feature of the tool is *trigger templates* for generating triggers using a script language [61, pp. 217–251]. Integrity maintaining action can be customized and extended in that way.

[35]This is our suggestion and not mentioned in the tool documentation

2.8 Repositories

Software Engineering [46] is the strand of computer science which covers different approaches to the systematic development and maintenance of software. When developing IS the requirements are often subject to constant changes as corporations adapt to the marketplace to stay competitive. "An enterprise should think of an ongoing continuum of information resource building" [55]. The term *software life cycle* [46] is connected to this idea. As this suggests not only is the data itself vital to a corporation's health but also adapting meta-data. *Computer aided software engineering* (CASE) [46] is the generic term for methodologies and tools to assist the software life cycle. According to [51] CASE encompasses even "enterprise strategic planning, information systems strategic planning, project planning, systems development, documentation and maintenance". "Central to any CASE architecture is the concept of the *data dictionary* or *repository*" [51]. A *repository* "is usually a database or file system in which all the information about the current status of the development process is held" [51]. When a DBMS is used for storing the information its features like locking, permissions and data integrity rules may support the interaction in distributed development performed by a team of designers and programmers. CASE tools, like S-Designor or the rapid prototype developed in this project, may access the repository directly and in [51] it is remarked that "CASE started as nothing more than an attempt to automate paper-and-pen software methods, which in turn provided the much-needed standardization of documents and models". This standardization is facilitated and forced by the use of a central repository and serves the communication between developers and is a basis for interaction of different CASE tools. Especially for the latter generally accepted standards on repositories are needed. They are unfortunately presently not available for two reasons: (1) There is no agreement on the scope and contents of a repository structure and (2) corporations already use proprietary or tailored repositories and changing a structure once established is very costly, hence it is in most cases not feasible.

According to [76] "the four major standards efforts that surround repositories" are:

- CDIF (Case Data Interchange Format) [30]
- IRDS (Information Resource Dictionary System) [42]
- PCTE (Portable Tool Environment) [19]
- ATIS (A Tools Integration Standard)

In the context of this work the use of a repository is limited to the storage of information on data models, also known as *meta-data*. Figure 28 relates the repository database to the target application database and the levels of visualization within this project.

The multilevel repository is a transparent means for storing the of the data model and should include also the two way mapping between the layers. This will be the basis of any visualization. CDIF is used for structuring the multilevel repository at the conceptual and abstract relational level. The purpose of CDIF, namely to transfer CASE data[36], is not specifically the intent of this work. However, in the following we adopt the proposed meta-model structures. So CDIF is briefly introduced at the end of this section.

Exact meta-data representations at the SQL level are given by the SQL-92 standard [49] and SQL3 draft standard [50]. The INFORMATION_SCHEMA tables [56, pp. 371–385] are set up to cover all language features of the particular standard. They should (1) be used by DBMS vendors to overcome the proprietary ways of accessing meta-data through their SQL interface and may (2) serve as a structure for storing SQL in a separate repository database [56, p. 371].

CDIF overview

CDIF stands for *CASE Data Interchange Format*. As the name suggests it is "a single architecture for exchanging information between CASE tools, and between repositories" [33]

[36] and thereby tackling the above mentioned problem of CASE tool collaboration

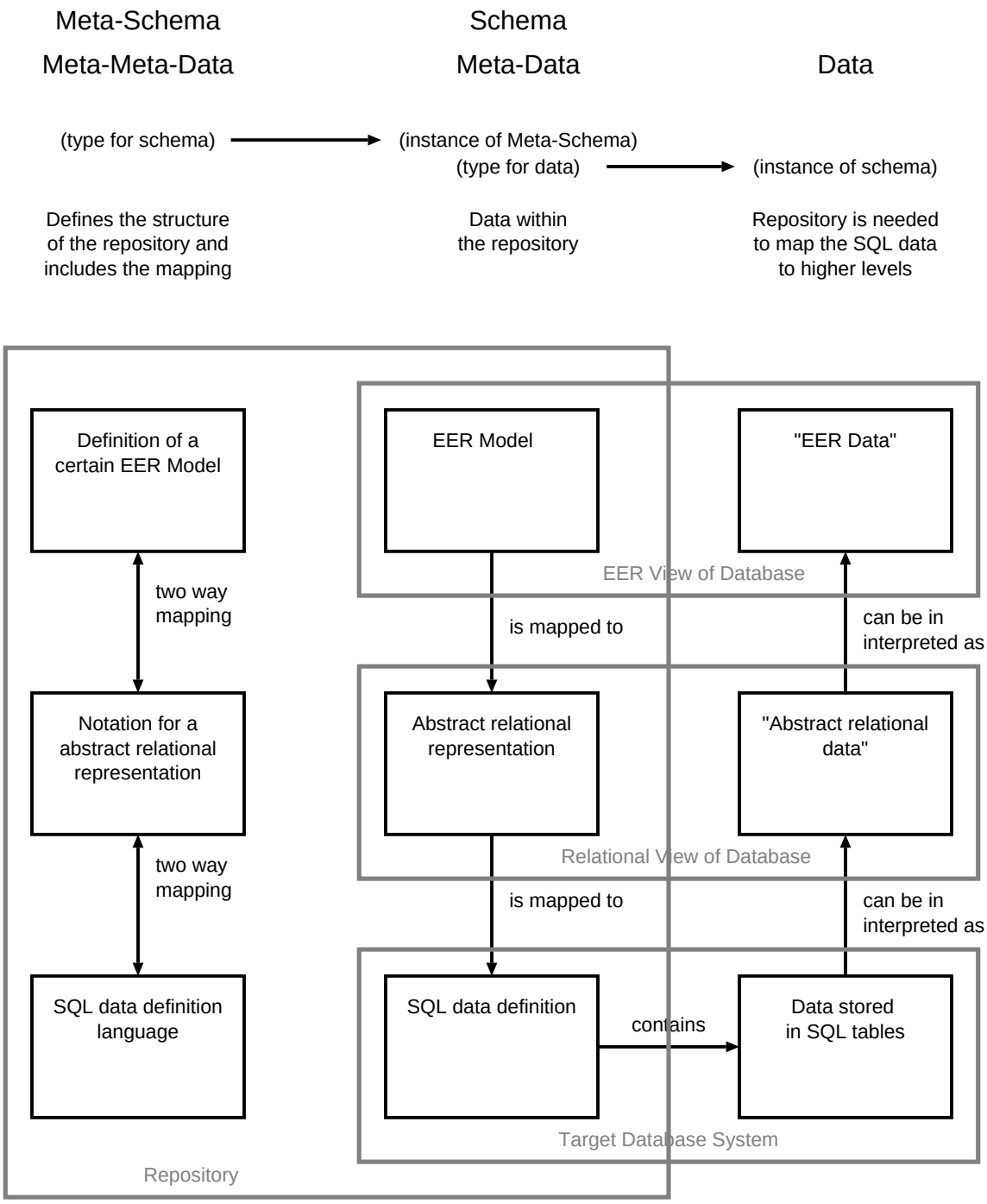


Figure 28: Meta-Schema, Schema and Data

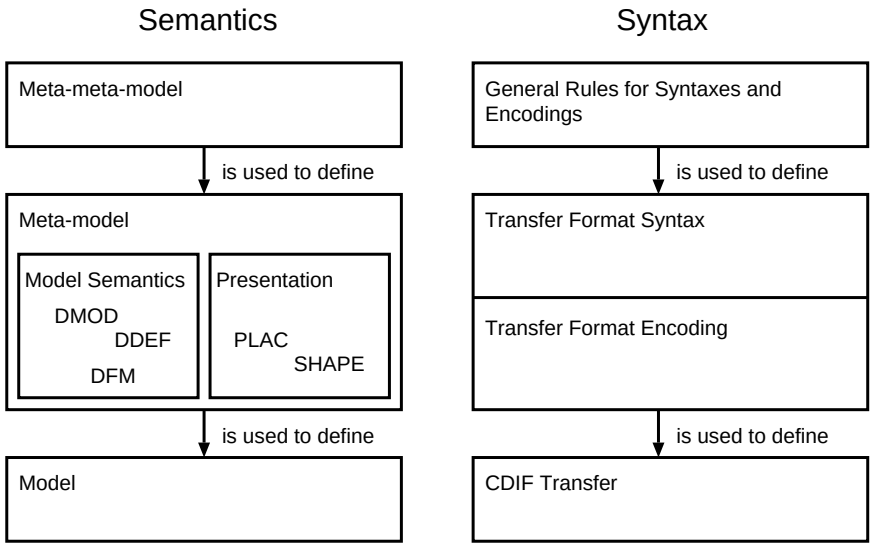


Figure 29: CDIF: Separation between semantics and syntax

which is "vendor-independent" and "method-independent" [33]. It should overcome the problem that no commonly agreed standard repository exists by defining a way of transferring data on the same *subject area* from one tool or repository to another. CDIF defines a meta-meta-model for the specification of arbitrary subject areas. Figure 29 illustrates the layering (see "Semantics" on left hand side).

Figure 30 lists the currently defined subject areas[37] to give an idea of the broad scope of CDIF. By making use of the meta-meta-model new subject areas may be added or extended if needed. The subject areas listed in Figure 30 try to be as general as possible for their particular domain, e.g. DMOD should allow the storage of all available EER notations and relational representations in the same meta-model. In this way method-independence is achieved. CASE software which imports or exports models may drop unsupported parts of a model. For transferring a model a *syntax* (see Figure 29, right hand side) and an *encoding* is defined. More than one encoding may exist for a given syntax. For now only one syntax and one plain text encoding exists[38].

[37] The subject areas relevant for this work are in italic boldface

[38] An OMG/CORBA-compliant way of exchanging CASE data is planned as part of CDIF [33].

Abbreviation	**Full title of Subject Area**
DMOD	*Data Modeling*
DFM	Data Flow Modeling
DDEF	*Data Definition (Data Types)*
STEV	State and Event Modeling
OOAD	Object Oriented Analysis and Design
PRDB	Physical Relational Database
PMSE	Project Management: Sizing & Estimating
PMPS	Project Management: Planning & Scheduling
PMTR	Project Management: Tracking
PLAC	*Presentation, Location & Connectivity*
GPAR	Presentation: Global Parameters
SHAPE	Presentation: Shape

Figure 30: List of CDIF Subject Areas

2.9 Related Work

Literature concerning the previous sections has already been cited at appropriate places. Some examples of related efforts in the area of database visualization are now given. This section tries to give an idea of the broad scope of the term "visualization" in connection with databases. The focus will then be narrowed to systems and proposals which are directly related to the intent of this work. Thereby the design is motivated. It is described directly after this section, in Chapter 4.

An area only remotely related to this work is graphical *data* visualization, where spatial data for computer aided design (CAD) or geographical data [78] is stored in a DBMS. Often numerical data (e.g. time series) is displayed using graphical methods like bar graphs and diagrams. Future applications of DBMS may include multimedia data. Data will be visualized in this project in a conventional tabular style. Scalability aspects of browsing tabular data are discussed in [77]. This issue is raised in Section 3.4.

Tools exist which merely show the structure of the database without including data at all (for an example on the World Wide Web (WWW) see [40]). Most modeling tools including S-Designor [61] are limited in this way. A novel approach in the context of ER modeling was taken in [18]. It is argued there that "there is a dearth of software offering modelers an environment in which they can create a model or partial model and then interact with it to better understand its 'behaviour' " [18]. A tool called TOTEM is introduced to (1) allow ER data modeling and (2) to populate the ER model with data. The emphasis is put on the cognitive aspects ("creative data modeling") of using a tool compared with "pen and paper" modeling. Even though update propagation is not considered in [18] the work has certain similarities to the efforts of this work. A comparison is included in Chapter 6.

Our focus is set on the visualization of mapping between the layers. We try to preserve the semantics from the conceptual layer to lower layers. If, as described in Section 2.4 a cascade is the desired reaction to integrity violating updates a complete visualization has to

include data to make the effects of an IC visible. In this sense the work is related to other work on active databases and visualization of arbitrary rule sets and triggering sequences. In what follows, some ways of implementing such visualization are listed. They can all be seen in this narrow perspective of active database behavior.

- A stand alone application without connection to a DBMS may visualize triggering cascades based on input files which contain sample data, meta-data and sample operations on the data. This approach was taken in the research prototype Adela [37]. Data is only regarded at an instance, i.e. row level, not in its data model context. The focus is set on the visualization of triggering cascades due to arbitrary ECA rules in a relational framework. "An event/rule trace tree is introduced to display: i) the context on which rules are fired, and ii) the execution states of rules" [37].

- It is desirable for debugging purposes to visualize triggering sequences and actions of real applications during their normal operation without interfering with their behavior. This may happen while the system is running or off line. Both cases are likely to be based on DBMS log file evaluation. An ECA rule debugger [13] for the object oriented active DBMS prototype Sentinel [12] is working in this way and allows the visualization of rule graphs and event graphs[39].

- Schema-related data visualization could also be part of a database application. However, to our knowledge all of these systems are only for data retrieval, e.g. the MORE prototype [52] only allows queries to be built, using an EER notation.

Our approach is to a implement a general database query and manipulation tool which operates on real application data. The necessary meta-data for accessing the application database is retrieved from a repository database. The repository also stores information about the EER layer, the mapping between the layers and the graphical notation (e.g. lines

[39]Sentinel allows composite events; their initiation is made visible

and boxes). The data in the repository is gathered as a side effect of the development process and does not require extra effort.

Chapter 3

Visualization Requirements

3.1 Introduction and cognitive aspects

The use of a repository based on a DBMS in this project results in a clear separation of programs and data structures. Nevertheless the requirements of both parts are interwoven. The requirements for visualization govern the requirements for the repository structure. Once the latter requirements are defined the development can proceed with the design, implementation and partial testing of the repository (Chapter 4) and can go on to the design, implementation and testing of the visualization programs (Chapter 5), i.e. the overall resulting system. Therefore visualization requirements are introduced first in this chapter.

Graphical design methods for designing the data schema are widely used and accepted since they allow relatively simple communication between technical and non-technical users; the latter know the real-world problems but are often unfamiliar with database technology. The validation process would be better supported if the graphical notations of the structure could also be used to incorporate dynamic functionality by populating the model with real-world *data*. This could be regarded as a way of rapid prototyping where no design effort besides the definition of the data model is needed. A means of exploring the database schema, in the introduced layers of abstraction to gain a better understanding of its structure and

dynamics should be looked for within this thesis. The following concepts which are known to improve understanding and learning should be considered.

Illustration: Graphical notations are easier and faster to understand than textual notations. "Solving a problem simply means representing it so as to make the solution transparent" [65, p. 153].

Interaction: Static notations like printed documentation have the drawback that all information has to share a single page. Interactive interfaces allow the user to focus on the information of interest and they seem also to be more appealing than the printed information (see [58, p. 38] on "multimedia" and "surface representations").

Examples: Examples bring life to theoretical concepts. Examples (data) and their logical structure (meta-data) are interdependent. If the schema is populated with data the user can see, for example, the update propagation properties by deleting or inserting rows into the schema. "A person is much better at solving a problem entailing familiar material — material that enables one to place oneself inside the situation" [38, p. 362].

Views: There is often more than one way of explaining the same matter ("Fitting representation to the task" [58, p. 63]). Different views of the same matter may give new insights especially if the relationships between those views are explicitly shown. This should not be confused with a database view which "may be a subset of the database or it may contain virtual data that is derived from the database" [32, p. 8].

Beside the above issues we make no further claims about the cognitive properties and the possible usefulness of the design in practice.

Initially it was planned in this project to have a single view for the complete visualization, including data and schema. This was not feasible because the relations between objects should be shown on the instance level to make update cascades visible. This means, for example, that at the conceptual level the visualization includes entity instances as well as relationship

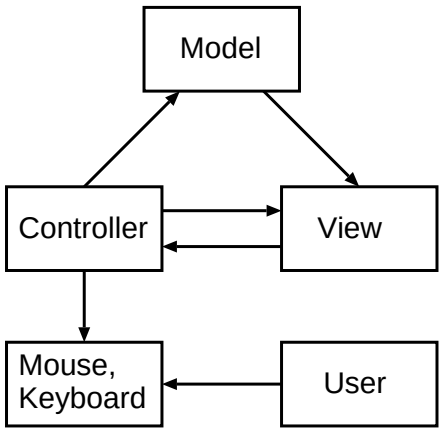


Figure 31: Model-View-Controller paradigm

instances. It might be possible to include this information into a schema for very limited examples but not for realistic database schemas. Scalability is often a problem with such visualization. It is not only a technical or geometric issue of placing graphical elements on a limited two dimensional space but also of comprehensibility for users. This problem led to the overall visualization architecture as layed out in this chapter.

3.2 Model-View-Controller paradigm

Graphical User Interfaces (GUI) allow graphical drawings and various ways of interaction with the user. The above mentioned properties are typical for GUI interfaces. An application may, for example, spawn multiple windows and the user may be able to point and click on objects of interest. An early example of a windowing system is the *Smalltalk* "graphical, interactive programming environment" [41]. Smalltalk is known as the purest *object oriented* programming language. For describing the design of the visualization the *Model-View-Controller* (MVC) paradigm is borrowed from the Smalltalk terminology. Figure 31 illustrates the concept. "The *model*[1] is used to represent the data or knowledge about the

[1] "model" is used in the following in this sense. It will be distinguishable from "data model" or "relational model".

application constructed" [9] and the *view* is the output interface, while the *controller* takes inputs from the user and passes them to the *model*. "A major idea in the MVC is the isolation of the model from the view and controller. Because the model normally represents a set of data or knowledge, it has no need to know how the controller or the view operates" [9]. For communication between the components a protocol must be defined.

MVC applied to database visualization

The MVC concept is especially suitable for database visualization, since it allows any number of views on the same model. The model in this case would be a structure representing a data schema and include the data needed for visualization, e.g. coordinates and lines. The views[2] may show the conceptual, the relational or even the DDL representations of the schema. They may include application data or may just show a static graphical schema notation. The input via mouse operations is interpreted by the controller and sent to the model. The model will react according to semantics of the operation. The model has to map the data model objects involved to their representation in the different layers of modeling thus make visual the relationships between the different views. This will result in screen update messages which are propagated to all views which are currently active, including the view that sent the actuating message originally through its controller. The views will then update the objects of concern within their displayed graphic. The granularity of highlighting and displaying should be as small as possible, i.e. if some attribute should be shown, then only the attribute should be highlighted and not the whole entity or table. On the other hand, if a view does not support the highlighting of a certain object it should map it to the available objects, e.g. highlighting a row could be mapped to highlighting its table. This is very important since it allows views to relate to each other.

[2]In the implementation each view module will also contain the according controller even though they do not communicate directly.

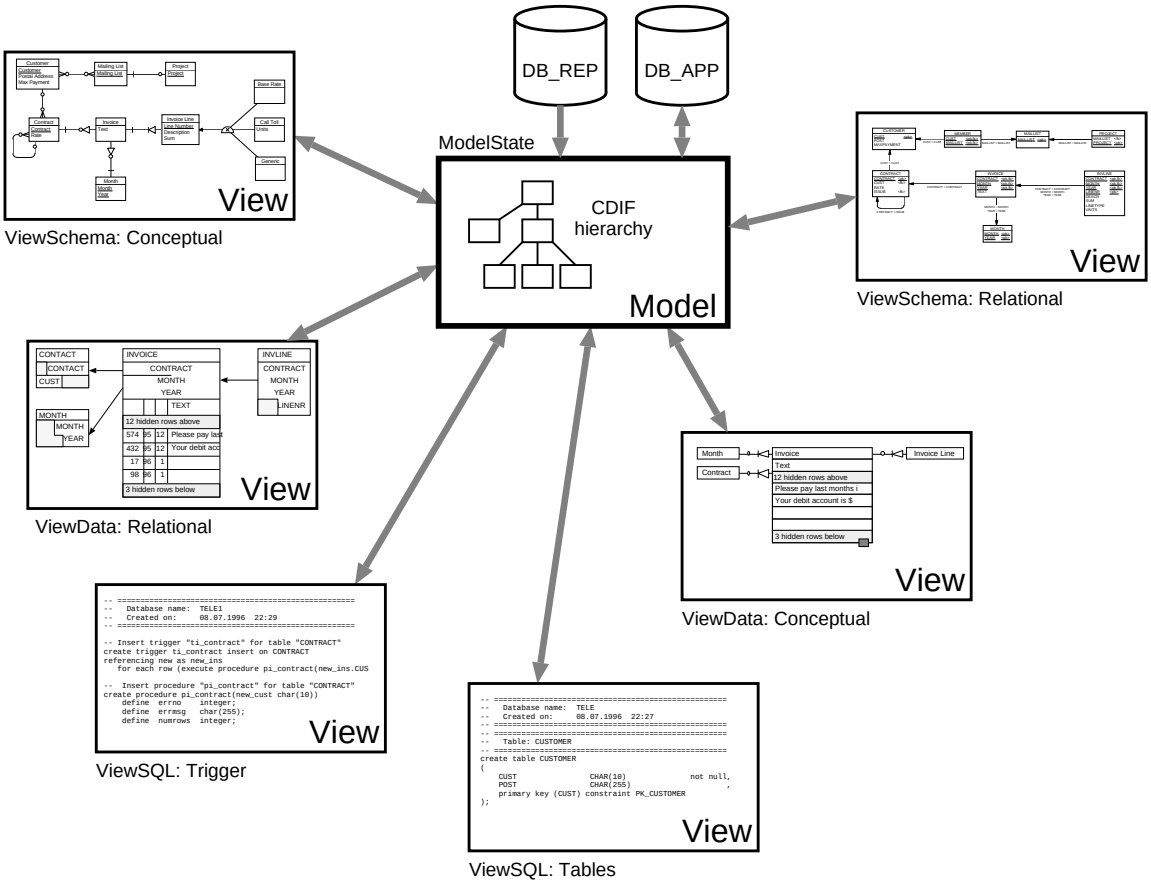


Figure 32: Model and Views

Proposed database views

The following sections propose a collection of views. The kinds of view offered and their functionality will be the major influence on the usefulness of the program. Some examples are anticipated in Figure 32. The content of each view will be deliberately limited to a certain aspect that the particular view is showing to maintain a practical and scalable presentation. Each view may be classified according to the scheme in Figure 28 on page 50. If more information is needed the user may choose an appropriate additional view. The visualization architecture may be extended by new views as needed. Those introduced here can be seen as a collection of ideas. Only a few will be implemented within this work to show exemplarily

the use of the repository structure and the program framework. Section 3.3 introduces views on meta-data only. In Section 3.4 views are discussed which allow the inclusion of instance level data. Implementation aspects are not dealt with in this Chapter. They are discussed in Chapter 5.

3.3 Views of Schema

ViewSchema

`ViewSchema` provides a conventional graphical representation of the conceptual or relational layer. This must be specified when starting the view. The notation used for each layer was described in Sections 2.3 and 2.4. According to [18] textual annotations within the schema are important during modeling and for the comprehension of schemas. They should be visible in the schema and be related to the object which they describe.

Controller: A menu may allow the specification of certain actions connected to clicks on objects. Possible actions would be to highlight the object, or to show additional explanations such as textual annotations, or to enter one of the other views (e.g. a view on data) with the visual focus set to the selected object.

View: The objects are drawn using repository data, e.g. coordinates and annotations. Highlighting of objects works the same way only with a different color. Different colors could be used by messages send by the model to indicate some additional meaning when highlighting. This issue will be raised in Section 3.4.

ViewSQL

`ViewSQL` displays the SQL scripts which are generated by the schema design tool for direct use with the target application database. Browsing these files is more meaningful if their content is related to the other views. In order to achieve this the source file has to be parsed and the text positions of each section have to be identified and stored in the repository. The granularity of highlighting objects will depend on the granularity of the indexing. Two separate SQL scripts are of concern, the table definitions and the trigger definitions, so one has to be chosen at startup time.

Controller: The user may select parts of the text by dragging the pointer. These portions of the SQL script may contain the SQL representation of some semantic object of the repository. Thereby, this view can be used to pinpoint and highlight objects. Buttons to select a certain update operation (insert, modify or delete) may be provided if triggers of a certain reference role are shown to precisely specify the trigger definition of concern.

View: The SQL DDL of highlighted objects is shown in a special color. If a reference is selected, the DDL script for table definitions may highlight the foreign key attributes. The DDL for triggers may highlight any or all triggers (FK- and PK-role for insert, modify and delete) related to this reference.

ViewTree

The hierarchical structure of the repository (see Section 4.3) suggests a tree styled browser for the repository content. It would be a structured list of objects of in the repository. It is a way of viewing the schema by using the meta-model, i.e. the repository structure. The documentation index for the `cdif` package (see Section 5.2) could give an idea of how the structure can look.

Controller: The user may unfold parts of the tree and click on objects in the tree to get all available information and highlight them.

View: A highlight message may automatically unfold the part of the tree where the objects of concern are located.

ViewText

This view consists only of a text window. "Validation can be done by paraphrasing a conceptual schema in natural language and giving that paraphrase to a user for examination" [10]. The explanations may be automatically derived from the schema semantics or could include textual annotations and descriptions. The user may specify the desired level of

	Paraphrase of a relation in both directions
UoD	Each customer must have at least one contract, but contracts may be unrelated to a customer, i.e. exist for some internal purpose.
EER	Customer refers to at least one instance of Contract. Contract refers to one or zero instances of Customer.
Rel.	The foreign key NR of table CONTRACT references table CUSTOMER. Table CUSTOMER is referenced by the foreign key NR of table CONTRACT.

Figure 33: Example for `ViewText` messages for the relationship of Figure 20 on page 35

abstraction. Figure 33 shows an example of a relationship. The conceptual and relational paraphrases can be generated automatically. The level of detail may be selected by the user to suppress certain messages, because it is possible to include also low level information which is only of interest for application programmers.

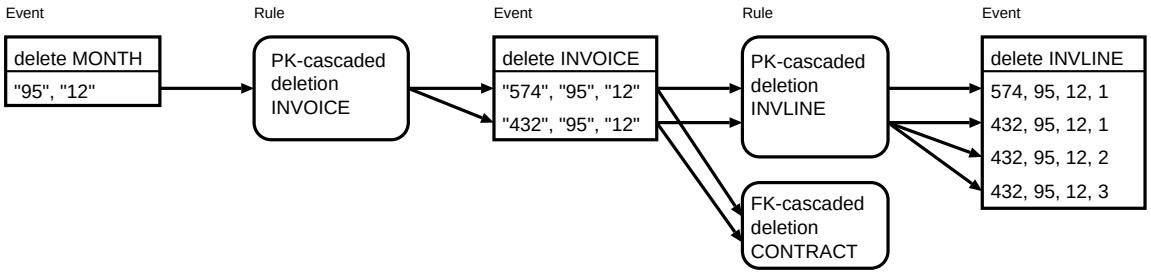


Figure 34: Sample triggering cascade displayed by `ViewCascade`

3.4 Views of Data

The views introduced in this section allow the user to browse data in the application database and some even allow updates to be applied which possibly result in update cascades. These cascades may be executed stepwise by the user, through the controller of some view. These actions may be traced through different views. Since the visualization should work on arbitrary data models, meta-data has to be looked up in the repository to build correct queries against the application database. These implementation issues are raised in Chapter 5.

ViewCascade

This view is an implementation of the event/rule trace trees used in [37] and [13]. It is limited to the visualization of trigger cascades at the relational level, i.e. rule trace trees. Event trees are not of concern since only simple events are considered within this work. The first event of a cascade is drawn at the left border as depicted in Figure 34, and all subsequent events are drawn right of it. Therefore this view incorporates the notion of time. If transactions would are to modeled, it would be possible to mark their start and end point in such a diagram. The view may show any update, not only updates which involve more than one table, due to a cascade. If old update traces remain visible then a history of all updates is generated.

Controller: A cascade cannot be initiated using this view. This has to be done in an instance of `ViewData`. However, the user may click on any displayed item of a triggering cascade.

Clicking on rules[3] will result in a highlight message of the particular PK or FK-role action (insert, modify or update). For now only this view and `ViewSQL` work at this level of granularity. A button for executing the next step is provided. This should be grayed out if no triggering cascade is active. Another button for clearing the display may be added.

View: A sample triggering cascade is shown in Figure 34. The update operations are grouped according to their appendant table to maintain a clear screen layout. Scrollbars are provided to scroll back and forth in time (horizontal scrollbar) and to handle any number of rows involved (vertical scrollbar). A time line could show a sequential number of executed database actions and the execution point in time of each update. If visualization of instance the level is not feasible it could be restricted to just showing the number of rows involved.

ViewData

`ViewData` can be used at the conceptual (see Figure 35) and the relational level (see Figure 36). A parameter of the constructor defines the layer for a particular view instance. In this paragraph the term table is used for a relational table with rows as well as an entity with entity instances depicted in the same tabular fashion. This view is most meaningful for displaying update propagation. It contains elements of the schema and the data contained. Tables are displayed in a way that allows not only rows and entity instances but also reference instances and relationship instances to be visualized. All of them may be inserted, deleted and modified at the conceptual as well as the relational level.

Controller: Visual controls allow the user to change the appearance of the tables displayed. Tables may be popped up and down and attributes may be hidden or unhidden. Some mouse operations are illustrated in Figures 35 and 36. Also relationships and references

[3] A rule is a part of a trigger. If multiple triggers where allowed each rule could be modeled with its own trigger

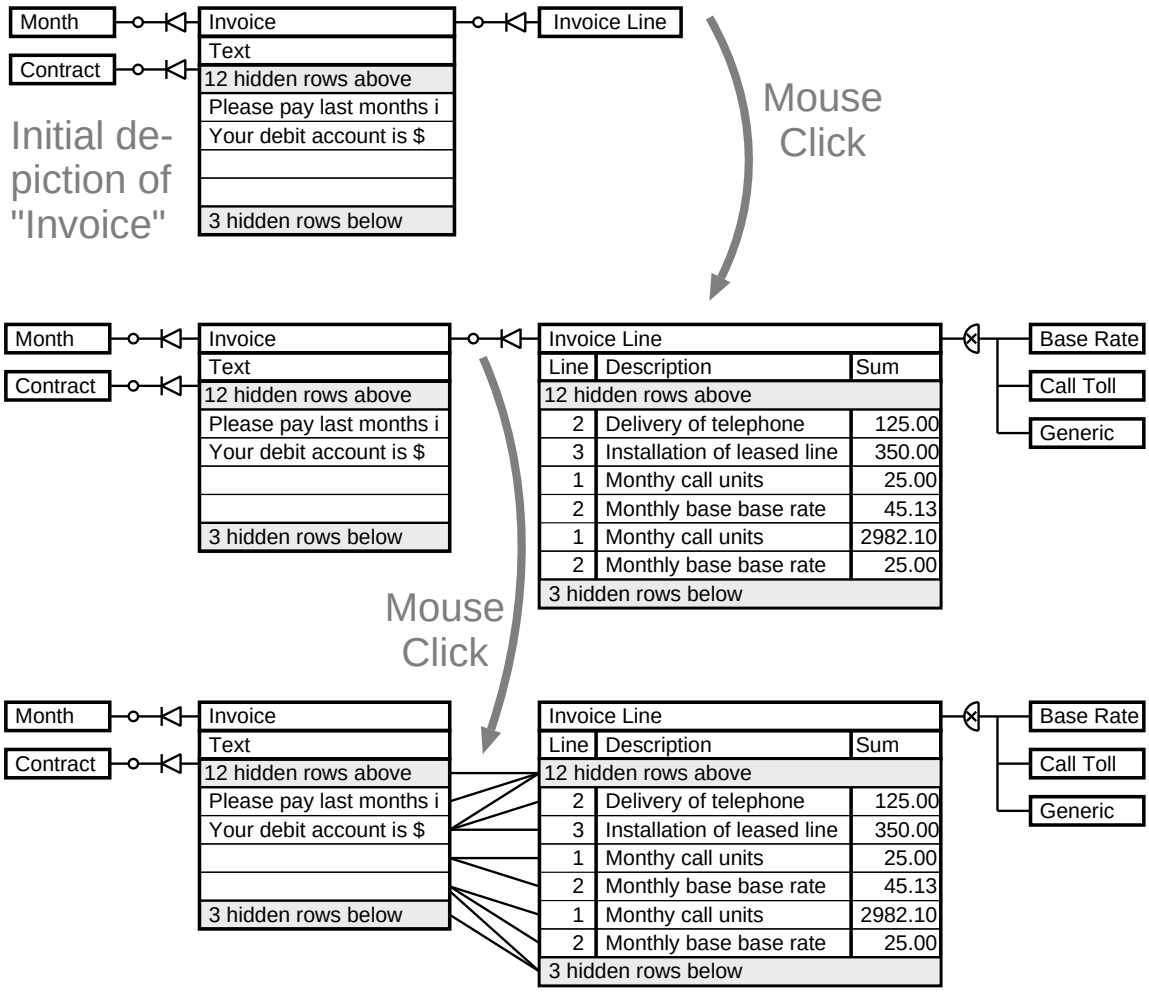


Figure 35: `ViewData` showing conceptual data

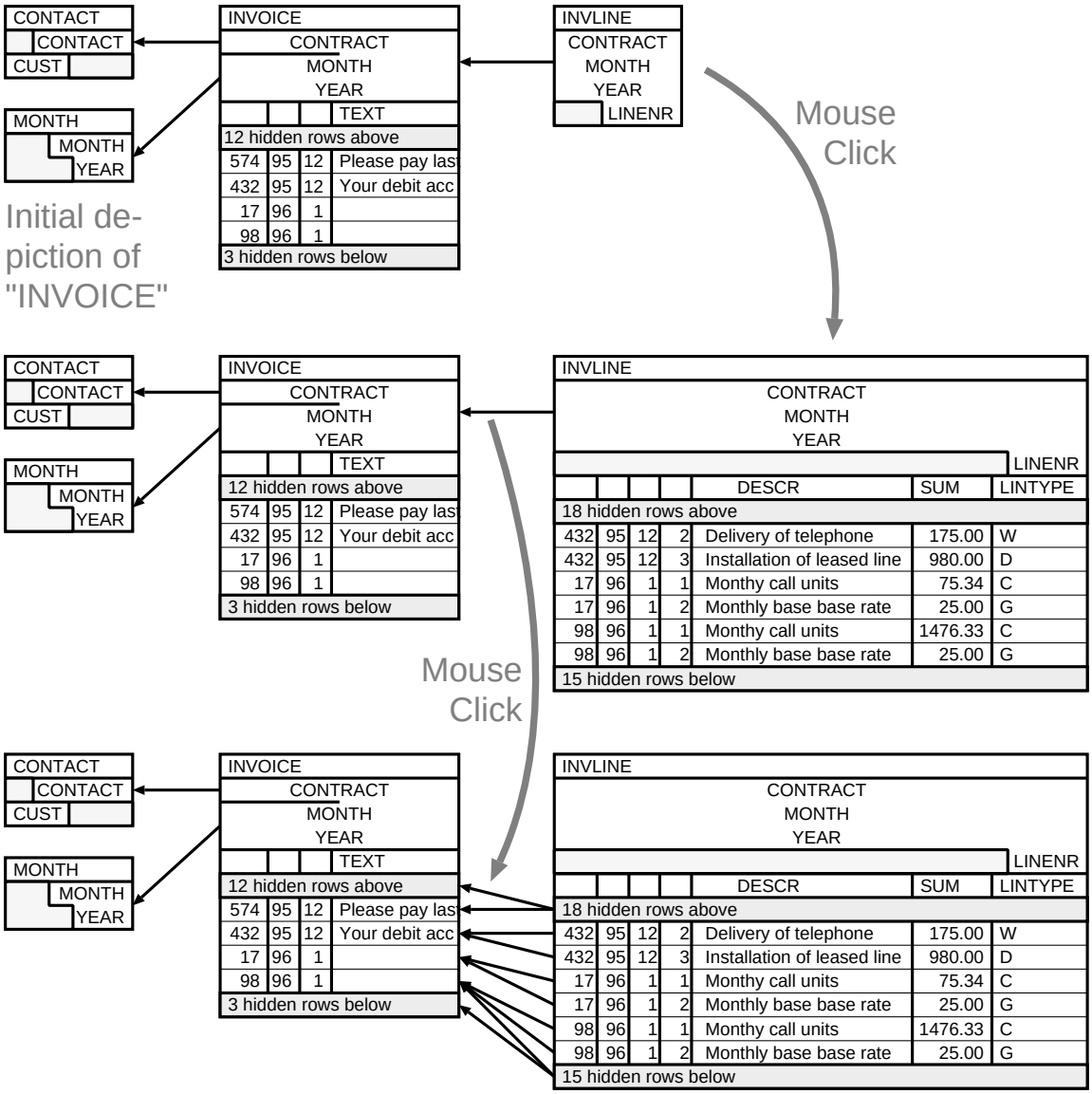


Figure 36: `ViewData` showing relational data

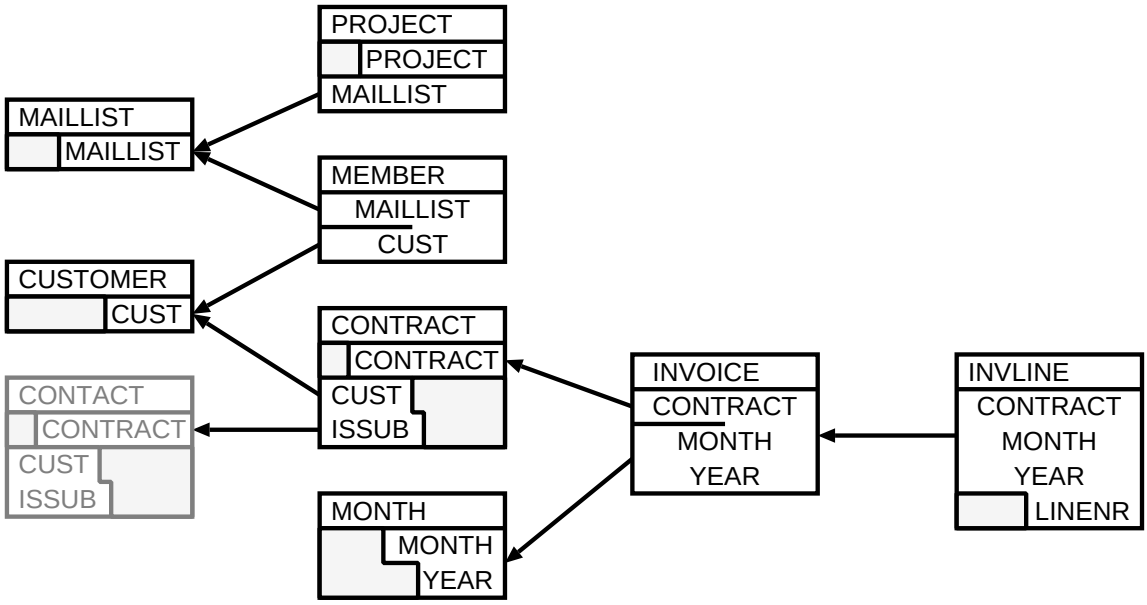


Figure 37: complete automatic layout of the UoD schema

can be switched into instance level visualization by a mouse click. Horizontal scrolling within a table may be necessary when it is too wide. The user may choose a highlighting or an update action (insert, modify or delete) which is connected to clicks on objects, e.g. a row of data. Highlighting works in the same way as in other views. Insertion and modification results in a data entry form (or permits direct editing of values in the table) which allows the entry of row data. Submission of this data may cause triggered action. By choosing the delete action, rows can be deleted by clicking into them. If update propagation action results from these operations a single step button is activated, as in `ViewCascade`, to allow single step execution of triggered action. After each step all views are updated including the `ViewData` view that actuated the cascade. Thereby the number of tables depicted may increase as necessary for visualizing updates.

View: This view requires a layout manager which places the tables automatically according to their neighbors in the schema. The order of tables can be derived directly from the schema semantics for the relational layer, because references, which connect the

tables, are by their nature directed. An example of a complete layout for the example UoD (c.f. the relational schema in Figure 12) of all tables which can be automatically generated illustrates this in Figure 37. The layout of EER entities is not as obvious, since relationships are not directed. However, 1:N relationships, weak relationships and supertype/subtype relationships can be regarded as having an implicit direction. 1:1 and N:M relationships may be layed out with the aid of their mapping to the relational layer.

The user may browse the full schema by clicking on tables, which unfolds their neighbors (see Figures 35 and 36). Also the schema may be unfolded by the update propagation visualization. A special problem in this context concerns circular dependencies. If a table occurs for the second time in a dependency sequence it will be grayed out and further browsing is disallowed as depicted in Figure 37. Automatic update propagation will in any case stop at this point, since each table can only be manipulated once during a triggering cascade. Updates may propagate in both directions. Updates due to PK-role integrity propagate from left hand side to right hand side, while FK-role integrity causes propagation in the opposite direction.

ViewTrace

This view contains a text window which displays the active behavior of the database in a comprehensible, textual and chronologically ordered way. This can be compared to the log file of a DBMS. `ViewCascade` records the database history in a graphical way. This view is less appealing, but easily extensible. Additional information can be given without the problem of graphical scalability. Like `ViewText`, this view is passive. No user interaction is propagated to other views.

ViewTree

`ViewTree` of Section 3.3 could be extended by instance level data at the conceptual or relational level.

Chapter 4

Development of Repository

4.1 Introduction

In this chapter the design of the repository structure is described. It concerns meta-data only. Data, which populates the schemas, resides in the application database only. Update propagation, which will be simulated by the visualization tool, is a topic of Chapter 5. The repository structure stores just the necessary update propagation properties.

First the adopted structure is introduced and motivated. Then this structure will be refined, implemented and mapped to a relational database schema. The modeling tool S-Designor is employed for this task. A set of utilities to load the repository and the application database with data and meta-data was developed. The second use for S-Designor in the project is thereby to generate conceptual and relational schemas which are loaded into the repository and visualized by the rapid prototype tool.

4.2 Motivation of Structure

The repository will be implemented using a relational DBMS, because this is a transparent way of storing repository data, i.e. data models. As stated in Section 2.8 the repository standards CDIF is used as a basis for the repository structure. The primary purpose of CDIF, namely transferring CASE data as described in Section 2.8, is not topic of this project. Just the structures offered by the subject areas *Data Modeling* (DMOD) and *Presentation, Location & Connectivity* (PLAC) were chosen as a basis for a repository structure to store EER and relational models and their graphical representation for the schemas. DDEF is related, since it allows the specification of data types for attributes. However, it was not used because in data modeling only simple 1NF data types occur so most of the DDEF definitions do not apply. The SQL layer of modeling (see Section 2.8 on page 48) was neglected, only simple markers are stored in the repository to allow meaningful browsing of SQL script files. A idea of a full implementation of this level in the repository, which includes the mapping, is taken up again in Section 6.4.

Unneeded features have been dropped from the subject areas allowed in a CDIF transfer file and some parts have been simplified. This does not mean that we lose the option to implement a CDIF export facility[1]. Figure 38 gives an idea of the broad scope of DMOD. The objects in the graph are called *meta-entities*. They have local *meta-attributes* and inherited meta-attributes[2]. The solid straight lines denote inheritance relationships. The curved dotted lines denote *meta-relationships* which are, of course, also inherited.

Two properties of the structure are apparent: (1) its hierarchical structure and (2) its universal applicability, i.e. one can find relational concepts like "foreign key" as well EER concepts like "sub type sets".

1. CDIF meta-models are hierarchically structured and make extensive use of inheritance

[1] This has never been a goal of the project
[2] Not included in the graph

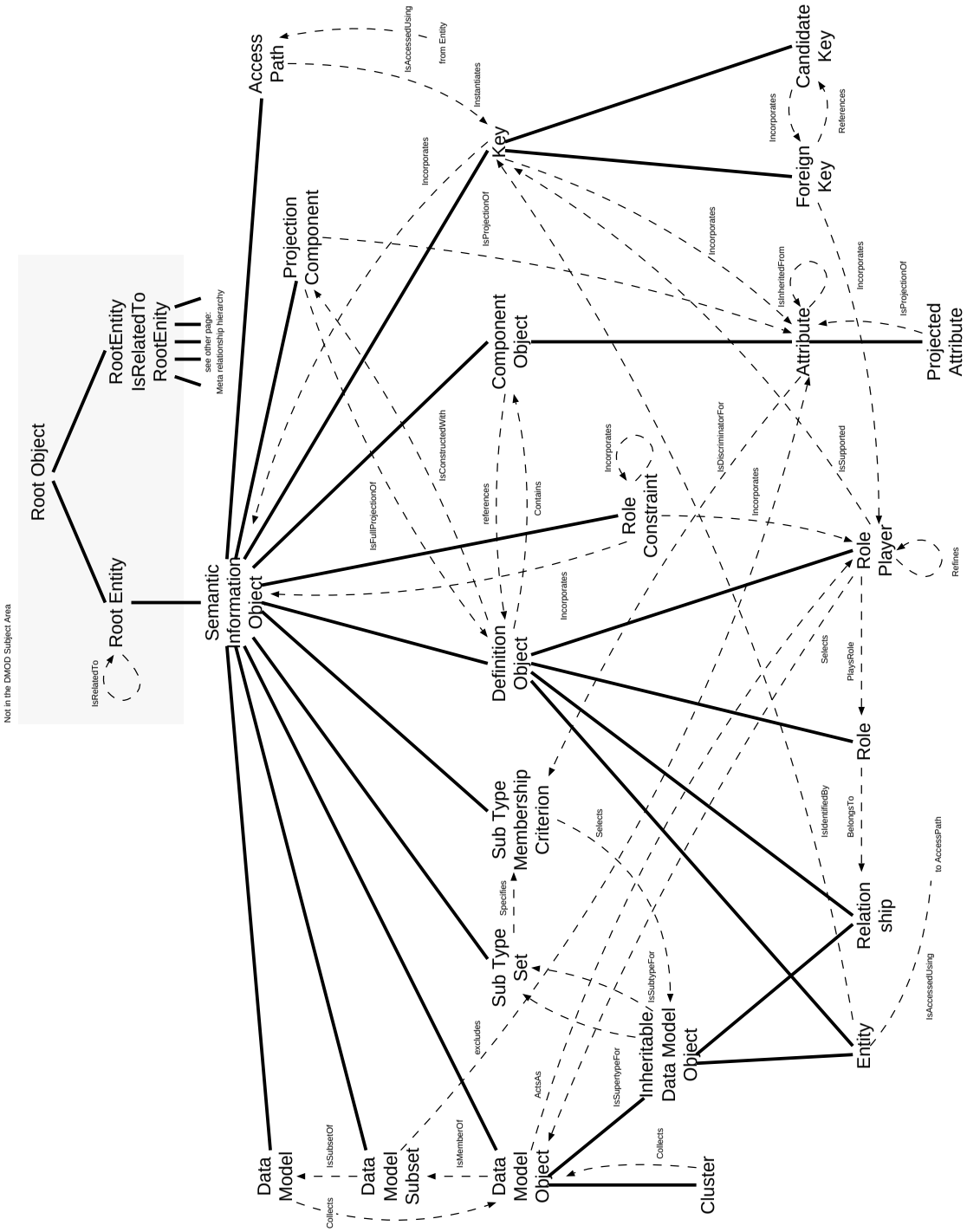


Figure 38: CDIF - Data Modeling Subject Area (DMOD)

to minimize redundancy in the meta-model. Hence they fit perfectly in an *object oriented* (OO) programming environment. It is possible to map the subject areas of CDIF directly to a *class hierarchy*[3] of an OO programming language that supports multiple inheritance. A particular schema instance will determine the *object hierarchy*. Each item in a particular schema (entities, attributes, ...) is instantiated as an object. Object references establish CDIF *meta-relationships*[4] by referencing related objects, e.g. an Entity is related to its Key.

CDIF features a clear separation of the semantic properties of a data model (subject area DMOD) and its graphical representation in some design tool (subject area PLAC, see Figure 39). This separation supports a modular design of software.

2. The CDIF subject areas try to encompass the full range of existing model variants, i.e. it is possible to store the EER representation as well as the relational representation of a data model using the same meta-model (DMOD). The approach taken in this work was inspired by the DMOD subject area. It reduces redundancy and hence the complexity of applications based on the repository. The alternative would have been to have separate structures for the relational model and for the conceptual layer. This would limit the extensibility of the meta-model if, for example, a new physical SQL layer should be introduced. The drawback of a single structure is that some parts are only applicable for a single layer, e.g. FK (relational) or Supertype/Subtype relationships (conceptual).

   CDIF subject areas may be extended if necessary[5]. As discussed in Section 2.6 the mapping between those layers is not clear cut, rather additional information must be supplied to allow the complete mapping. This information cannot be derived automatically and should be stored explicitly to allow for browsing and mapping between the

[3] The class hierarchy models the inheritance relations between classes

[4] dotted lines in Figure 38

[5] Even though this doesn't support the transferability of these particular features of a CASE model to a different tool, which doesn't know the extensions. "Extensibility is a concept inherent to CDIF: Tool vendors can define their own extensions to CDIF and represent data that uses these extensions seamlessly with data that uses the standardized CDIF meta-model only" [31]

layers. Hence extensions are needed to identify the layers and the mapping between the layers as shown later.

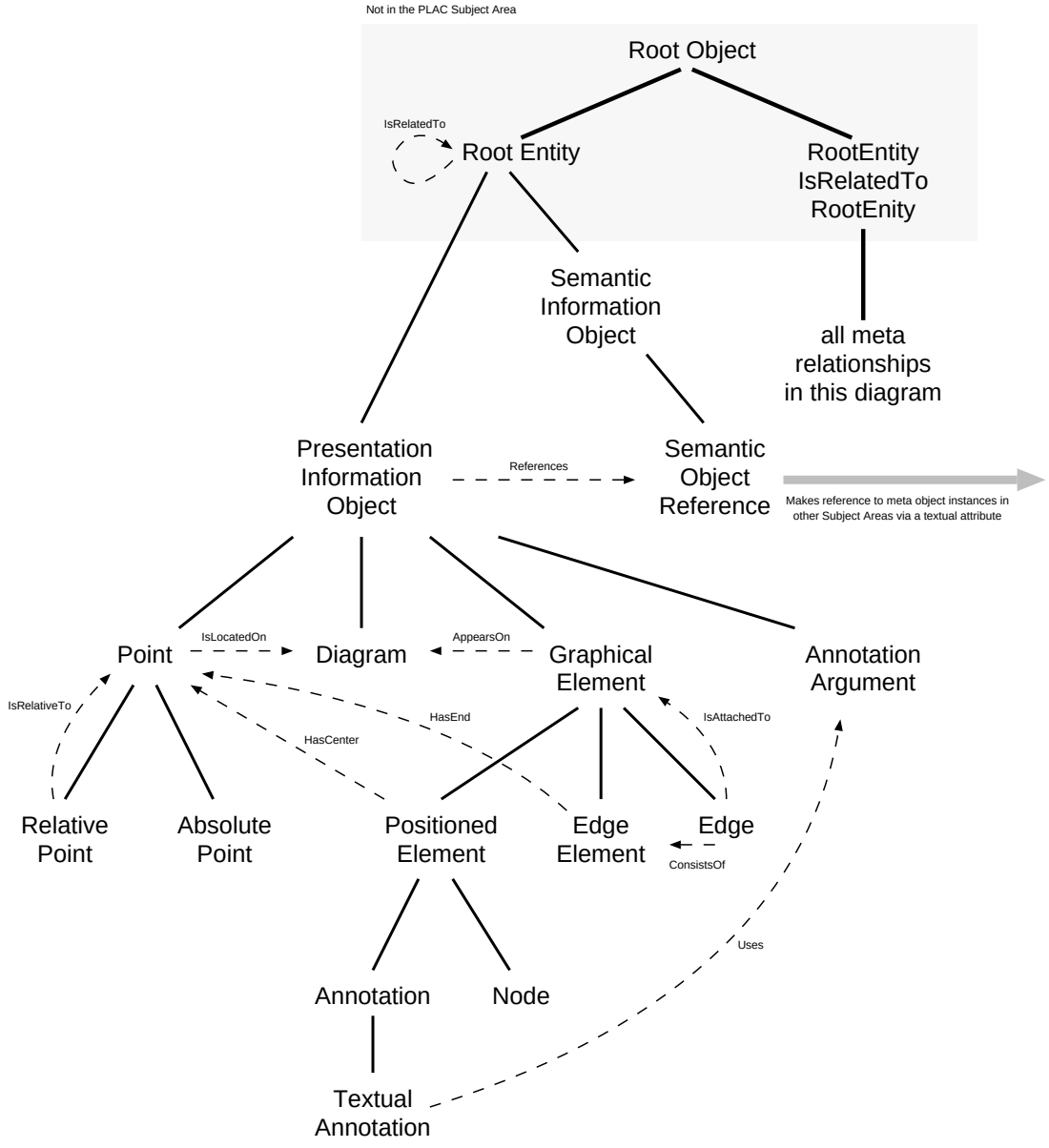


Figure 39: CDIF - Presentation, Location and Connectivity Subject Area (PLAC)

4.3 Repository Schema

The subject areas introduced thus had to be refined and implemented using a DBMS schema. For this project the modeling tool S-Designor (see Section 2.7) is used. The possible models are constrained by the modeling features of this tool. The implemented repository structure should nevertheless be as general as possible to allow the use of other tools. However, certain features of the CDIF structure are dropped or simplified because they would complicate the matter without any gain. The design simplifications are listed below (see Figure 38 for reference):

- *DataModelSubset*[6], *Cluster*, *Projected Attribute*, *ProjectionComponent* and *AccessPath* are dropped. Data model subsets and clusters are dropped to simplify the visualization part, however they are of importance for real-world problems to maintain scalability.

- *DataModelObject* and *InheritableDataModelObject* are merged into *DefinitionObject*. This way the multiple inheritance for *Entity* and *Relationship* is avoided. The purpose of the separation in CDIF-DMOD is that it allows inheritance from other data model objects. By merging the meta-objects into one, we restrict inheritance to *Entities*.

- The functionality of *RolePlayer* is merged into *Role*. CDIF-DMOD allows more than one role player for each role [44, p. 27], which is an advanced, and in our case, unnecessary modeling feature. Arbitrary *RoleConstraint*s are not allowed with the modeling tool S-Designor and therefore are dropped for the sake of simplicity. This problem was discussed for the Subtype/Supertype case in Section 2.7 but within CDIF-DMOD it is meant in a general way: any two roles (or even role constraints) may be mutually exclusive (XOR) to each other, dependent on each other (AND) or they may be disjuncts (OR). These constraints could be mapped to a DDL using the mechanisms introduced in Section 2.5.

[6]The context of CDIF-DMOD terms is shown in Figure 38 and their meaning is explained in [44]

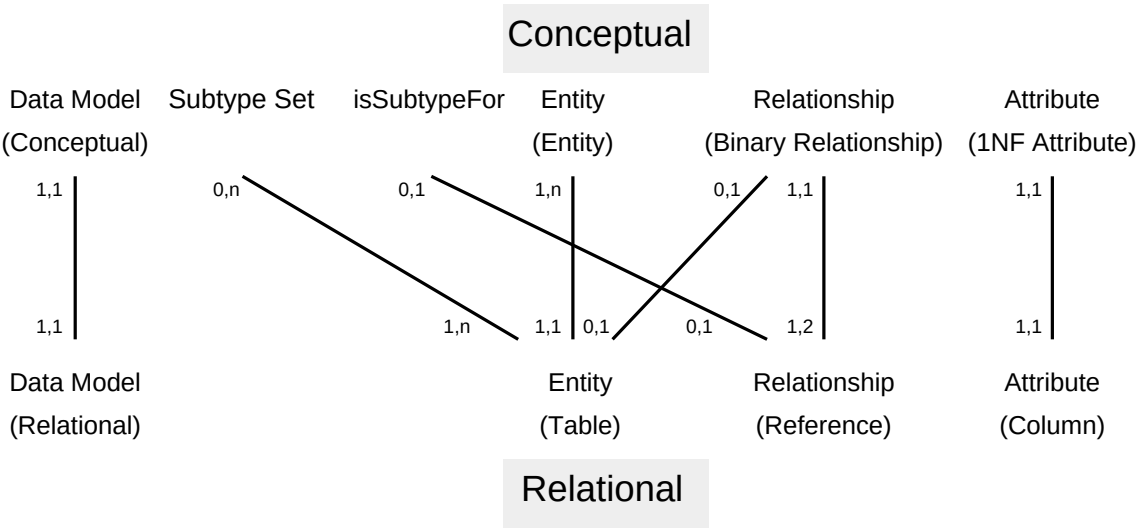


Figure 40: Mapping-meta-relationships, see Figure 23 on page 38

- *RootObject* was merged into *RootEntity*.

Explicit storage of mapping

The mapping of the layers should be explicitly stored as motivated before. The objects are assigned to a modeling layer by an additional meta-attribute *layer* which is part of the *RootObject*. Figure 40 shows conceptual and relational terms (in parenthesis) and a mapping. The semantics of which were discussed in Section 2.6. The mappings will be modeled as additional meta-relationships. They are called *Mapping-meta-relationships*. Their implementation can be seen in the conceptual (Figure 41) and the relational schema (Figure 42). The Mapping-meta-relationships are depicted with dashed lines and their roles are annotated in Figure 41 with PDM and CDM respectively.

Conceptual and relational repository schema

The repository schema was designed with S-Designor and its conceptual representation is depicted in Figure 41. The resulting relational schema is shown in Figure 42. The CDIF inheritance hierarchy was implemented with Subtype/Supertype relationships and transformed

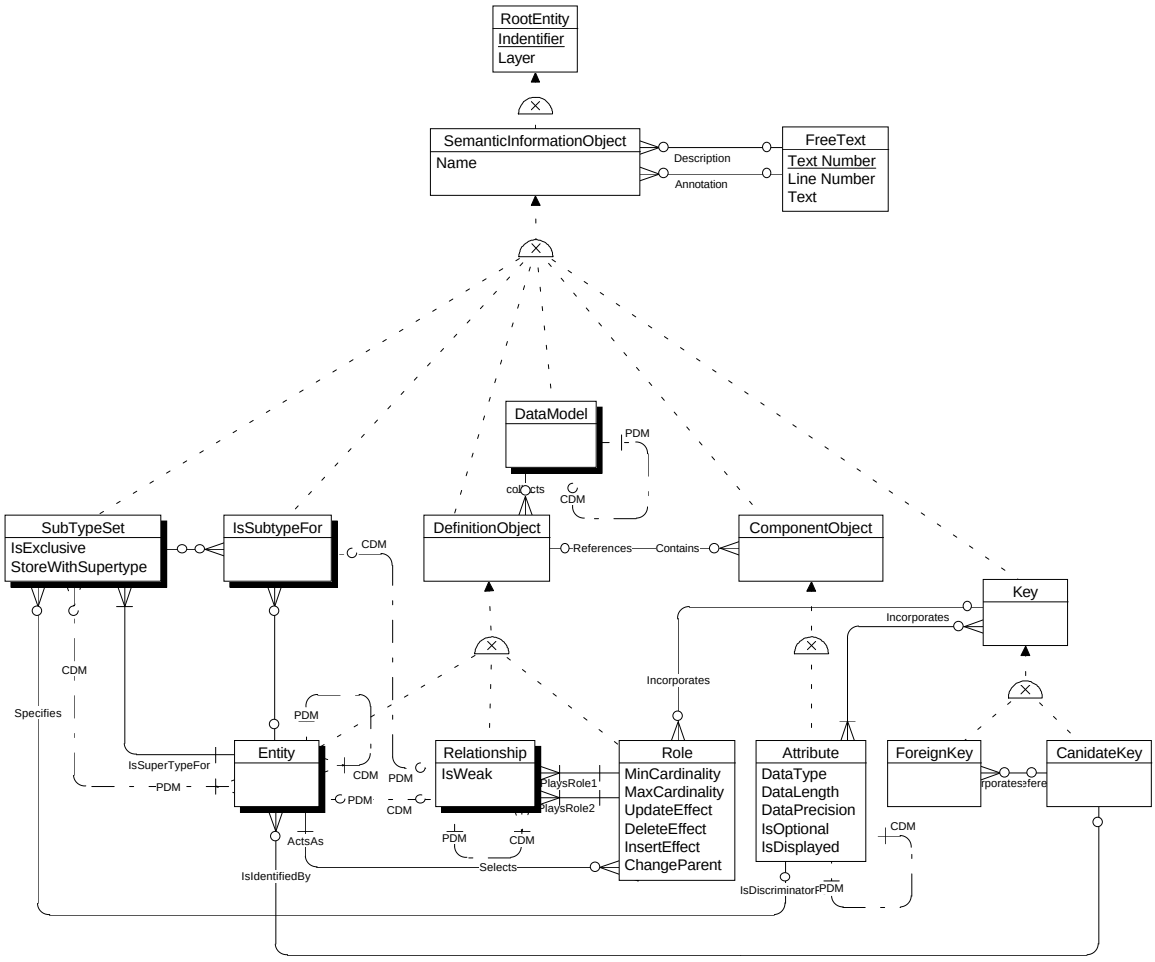


Figure 41: Conceptual schema of the repository structure (related to DMOD)

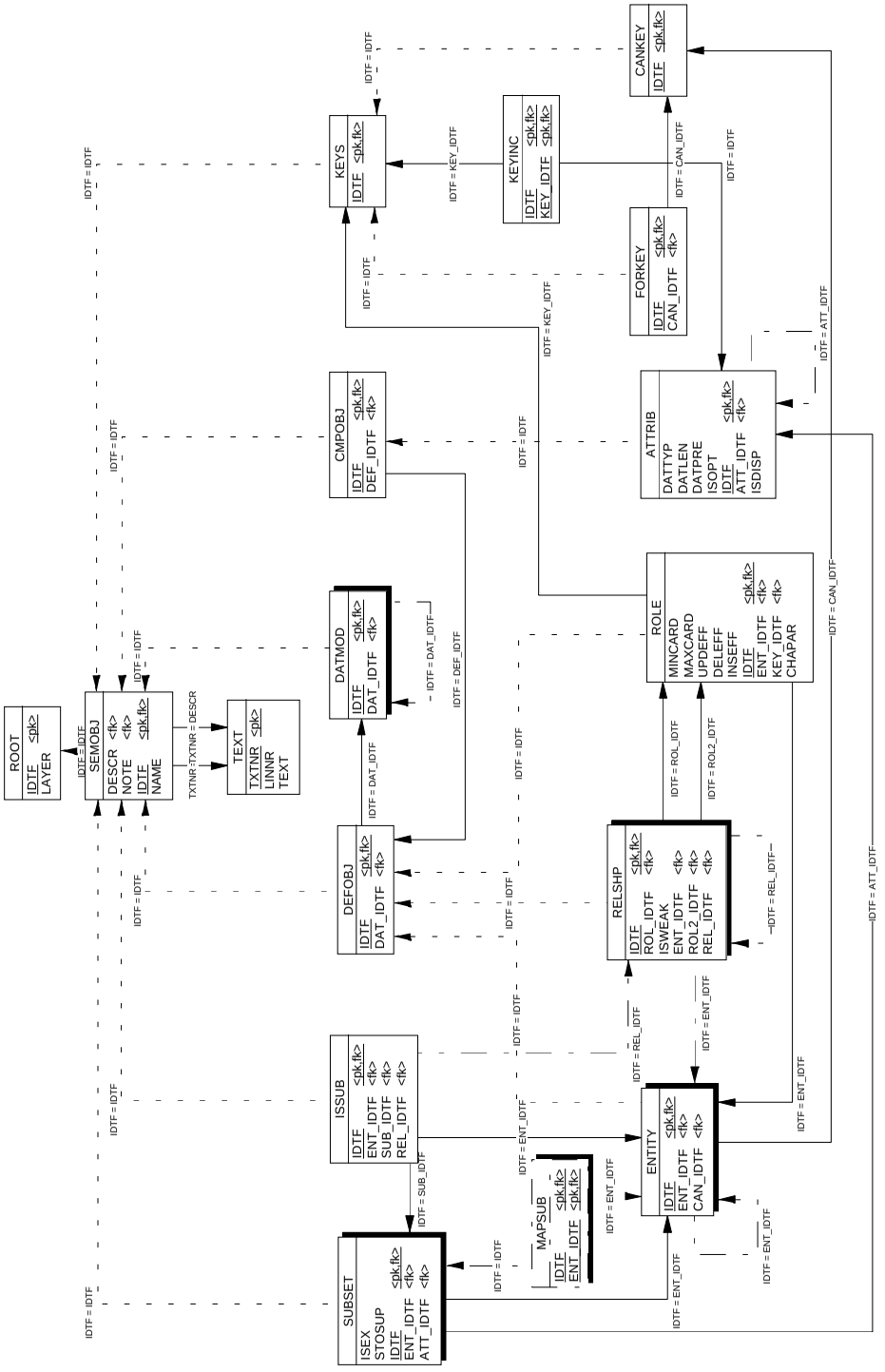


Figure 42: Relational schema of the repository structure (related to DMOD)

using mapping option A from Figure 21[7]. The *RootEntity* contains a meta-attribute *Identifier* which is an uniquely identifying integer for each object, i.e. a PK. Every meta-relationship is implemented as a FK containing such a foreign PK-value.

Schema visualization data

The repository should also capture the information needed for the visualization of the schemas, i.e. lines with arrows, points and boxes. The CDIF subject area *Presentation, Location & Connectivity* (PLAC) is shown in Figure 39 and should be used as a basis for the repository structure for the presentational objects (see Figure 43 for resulting conceptual schema) in the same fashion as DMOD was used for the semantic objects. Figures 39 and 43 may be compared to see how the original subject area was altered. This information is essential for some *views* and requires that each semantic object from DMOD knows its representation in PLAC. The other way round (association of a PLAC object with a DMOD object) is needed for some *controllers* to know which graphical object in the schema was located by the user. This two way linkage is accomplished by a *SemanticObjectReference* in PLAC (see Figure 39). It contains originally a textual pointer into any of the other subject areas, including DMOD, and is implemented as a FK attribute of *PresentationInformationObject* as depicted in Figure 43. This FK attribute simply contains an object identifier of the DMOD *SemanticInformationObject* which is represented by the PLAC *PresentationInformationObject*. The relational schema of PLAC is depicted in Figure 44 and is merged with the DMOD relational schema, i.e. they both share the same database. Table *ROOT* of Figures 42 and 44 are therefore identical.

[7] The same Figure shows that *outer joins* [32, p. 168] or *equi-joins* [32, p. 160] are necessary to retrieve the full information of a particular object. This may cause a performance loss, which indeed turned out to be a problem with the implementation techniques described later in Section 5.3

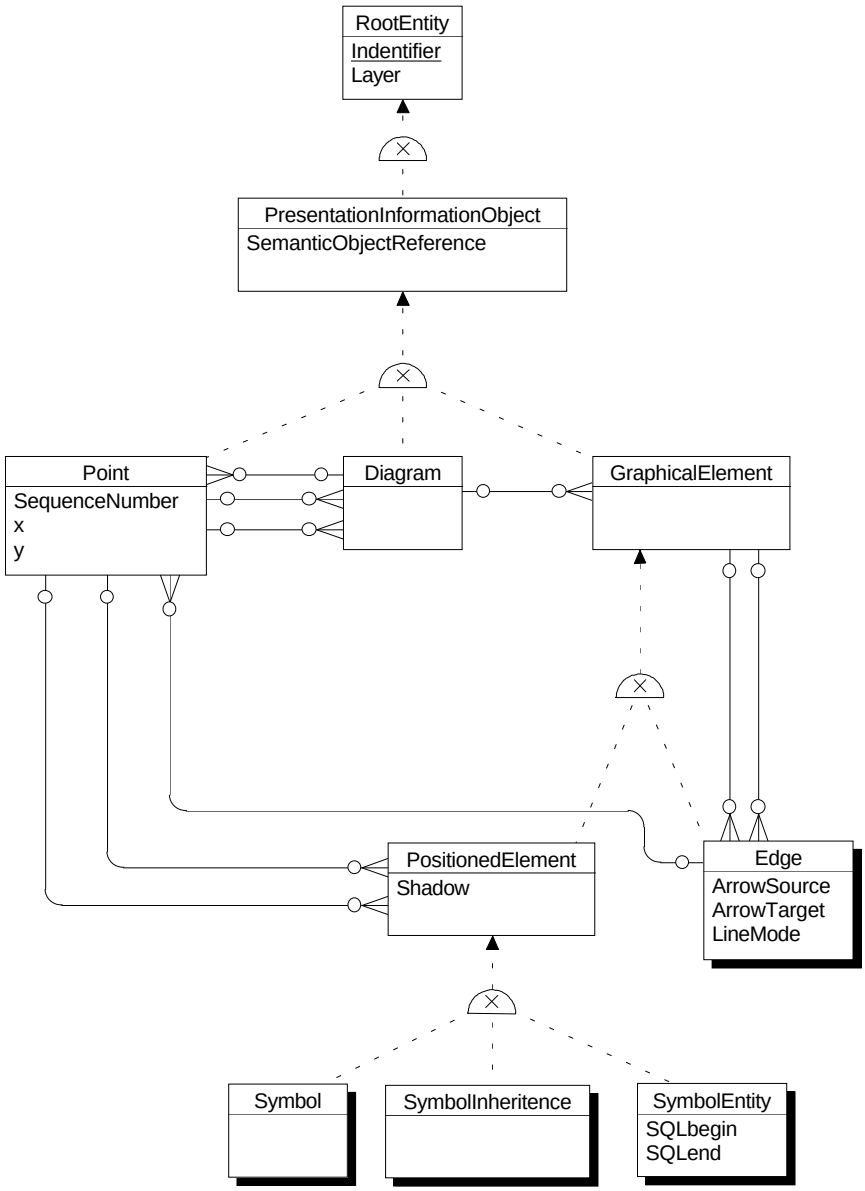


Figure 43: Conceptual schema of the repository structure (related to PLAC)

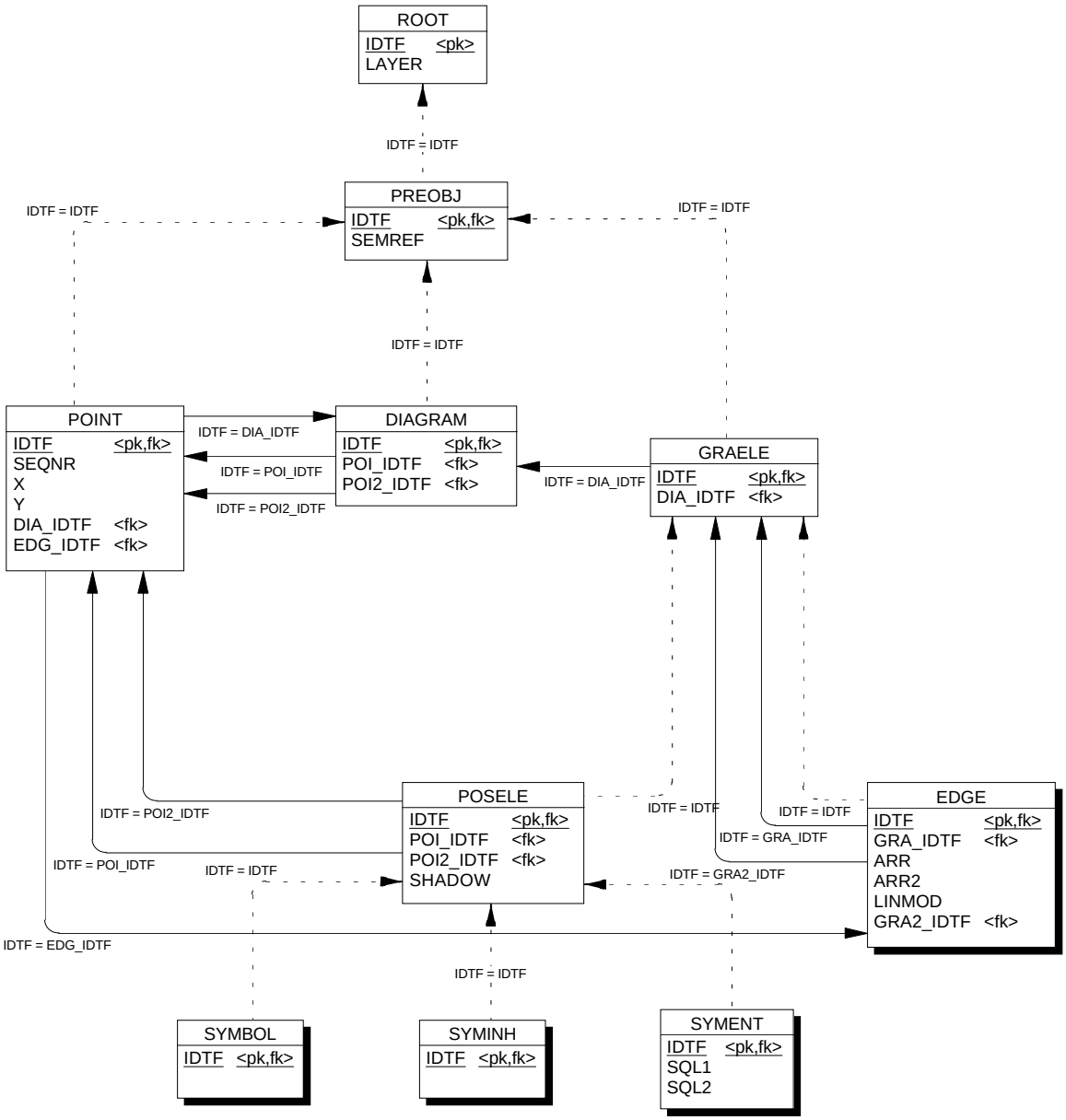


Figure 44: Relational schema of the repository structure (related to PLAC)

4.4 Support Programs

The following describes how the schemas are loaded into the repository. An overview of the process including databases, files and programs is depicted in Figure 45. The S-Designor database design tool (upper dotted box in Figure 45) saves conceptual models and their relational representation into separate files. These files consist of plain text and are organized in a relational fashion. The program `sdda2sql` translates both files to simple SQL DDL and DML statements. `sdda2sql` merges the relational and conceptual definitions by assigning new unique numerical identifiers to each object. The manual page for the program can be found on WWW pages[8]. The generated files can be used as a script for loading the data into any SQL DBMS. Once the data is accessible in a DBMS it can be manipulated, i.e. transformed into the desired repository structure as described in the previous section. This is done by the program `trans-rep`. Attributes which are non existent in a S-Designor schema are given meaningful default values. Finally the tool `sql-rep` writes starting and ending line numbers of the table and trigger definitions to the repository to allow for meaningful visualization of the SQL scripts generated by S-Designor (see Section 3.3). All three programs are written in Perl [15] and access the Mini-SQL database server [45] through the Perl-DBI/DBD interface [4]. Mini-SQL is a simple SQL DBMS. It is accessed through *embedded SQL* in the *host language* Perl. The sequence of necessary transformation steps is controlled by the *make* utility. After saving the necessary files with S-Designor a single invocation of *make* loads the repository and the application database. All programs, including some Unix-shell functions for convenient database access, make use of a set of common environment variables which contain the names of databases and relevant file system paths.

The access of the repository by the visualization tool (see Figure 45, box `visualDB`) needs more explanation. Even though this description is closely related to the material in this chapter we leave it to Section 5.3, where the overall design of the visualization tool is introduced.

[8] http://www.his.se/ida/msc/finished_projects/HS-IDA-MD-96-001/doc/man

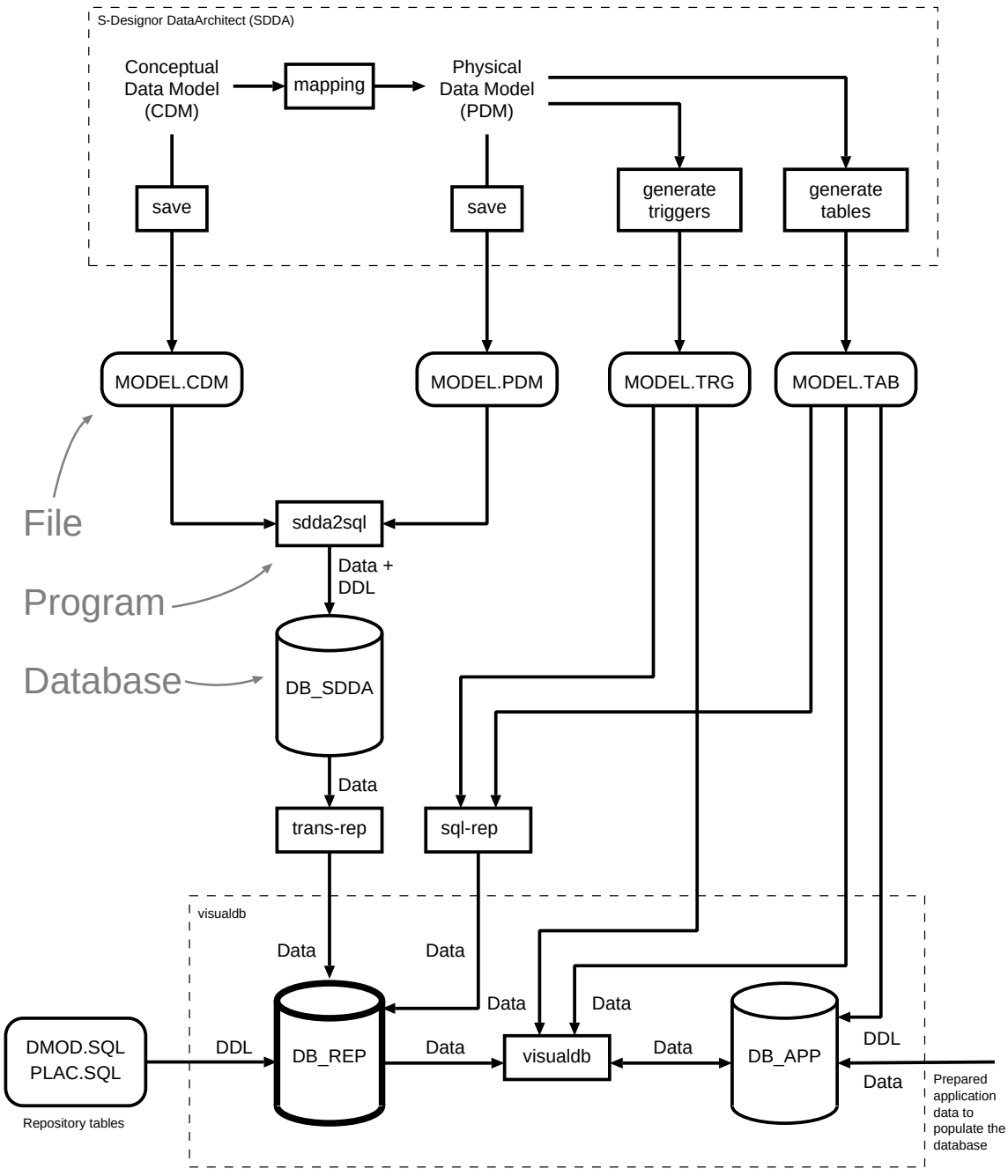


Figure 45: Overview of Programs, databases, and files for the repository administration

Chapter 5

Development of Visualization

5.1 Introduction and Programming Language

Chapter 3 emphasized that the MVC paradigm can serve as a basis for database visualization and showed how the model, view and controller interact. The intended functionality of a number of views were introduced. This chapter describes the design and implementation of the building blocks of this concept which should result in a suitable behavior according to the postulated requirements in Chapter 3. The main program classes are introduced first in Section 5.2 and then a description of their repository access mechanism follows in Section 5.3.

Programming Language Java

The visualization was implemented using the programming language Java [71]. Java was developed at Sun Microsystems [73] and is, according to [70], "a simple, robust, object-oriented, platform-independent, multi-threaded, dynamic, general-purpose programming environment". One reason for choosing Java was that the programs could be presented on WWW pages[1] together with online documentation, as for example is the text of this thesis. Other reasons for choosing Java concern various features of the language. This is, of course,

[1] http://www.his.se/ida/msc/finished_projects/HS-IDA-MD-96-001/

not an introduction to the language but each concept which was used is briefly introduced at the appropriate place in this chapter. Sometimes example code is provided, which should look familiar due to Java's syntactic affinity to the well known language C.

Java is a pure object oriented language, therefore OO-Design methodologies can be applied during development and for documentation purposes. Figures 46 and 47 are based on the *Unified Method* [8]. "Various language-dependent refinements are useful when class diagrams are used for detailed design and coding" [8]. Its application for designing Java programs is described in [7].

Purpose of rapid prototype implementation

It was mentioned that Java programs can be executed within WWW pages assuming that a Java capable browser is used. In this way the programs become part of the overall presentation. It is also possible to present implemented parts directly next to parts which are only designed or planned. Beside its documentary value the implementation forces a more concrete presentation of the design and reveals some insights for future work (see Section 6.4). The entire implementation of the rapid prototype was an optional objective and no claims about completeness and usability are made.

5.2 Class Structure

The implementation consist of two parts, each implemented in a separate Java package. The main program, the initialization of the repository structure and the implementation of different views are included in the Java package `visualDB` which will be described first. The second Java package called, `cdif`, contains the repository class hierarchy and will be discussed in Section 5.3.

A *Class Diagram* (Unified Method [8]) is used for depicting the classes of the package `visualDB` in Figure 46. The main program (`visualDB.Main`) inherits from `java.applet.Applet`, which allows its inclusion in a WWW page. Figure 47 shows a *Message Trace Diagram* [8], which illustrates a "single history without conditionality" [8]. Step 1 of Figure 47 marks the point in time when the page is selected by the user. The `Main` class is loaded by a contained HTML (Hyper Text Markup Language) `APPLET`-tag[2] and started by the browser. The initialization instantiates a single instance of the class `Loader` and `ModelState`. The `Loader` class contains the direct access to the underlying DBMS though embedded SQL calls and is discussed in Section 5.3. The user may select a database schema as shown in step 2 of Figure 47.

Observer and Observable

The class `ModelState` will serve as the model in the MVC sense. Java provides the class `java.util.Observable` which is a superclass for all "observable" objects (in this case for the model class `ModelState`) and the interface[3] `java.util.Observer` which must be implemented by all classes which want to "observe" an "observable" class. Three example view classes are depicted in Figure 46. Since the views should be stand alone windows, they inherit from the Java class `java.awt.frame`. The user may start any number of view instances of

[2]The `APPLET`-tag was introduced by Sun Microsystems and taken up by the World Wide Web Consortium [82] for HTML 3.2 [81]

[3]The Java term `interface` refers to an entirely abstract class, i.e. without any implementation. A class may "implement" any number of those "interfaces"

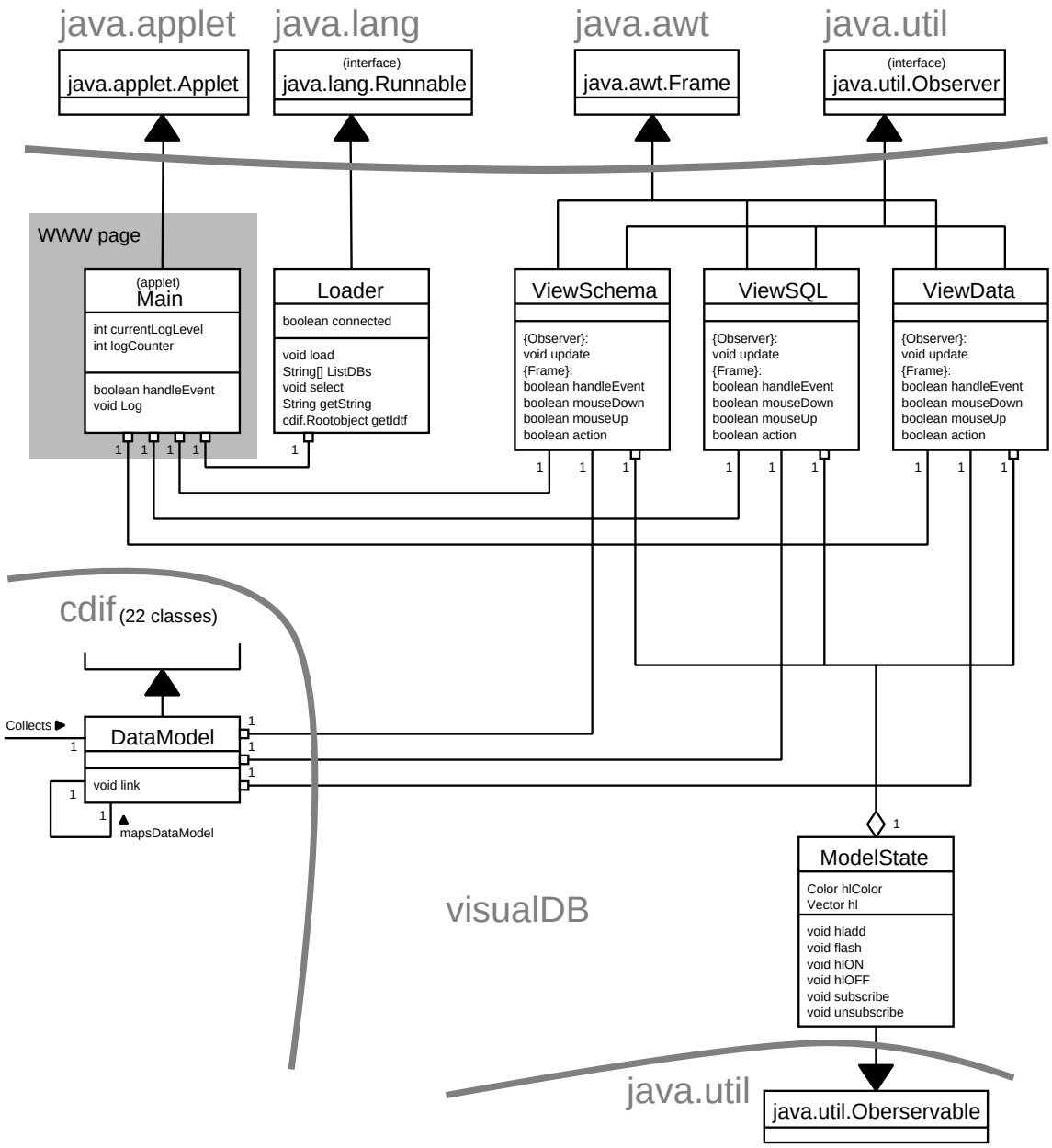


Figure 46: Class Diagram of visualDB Java-package

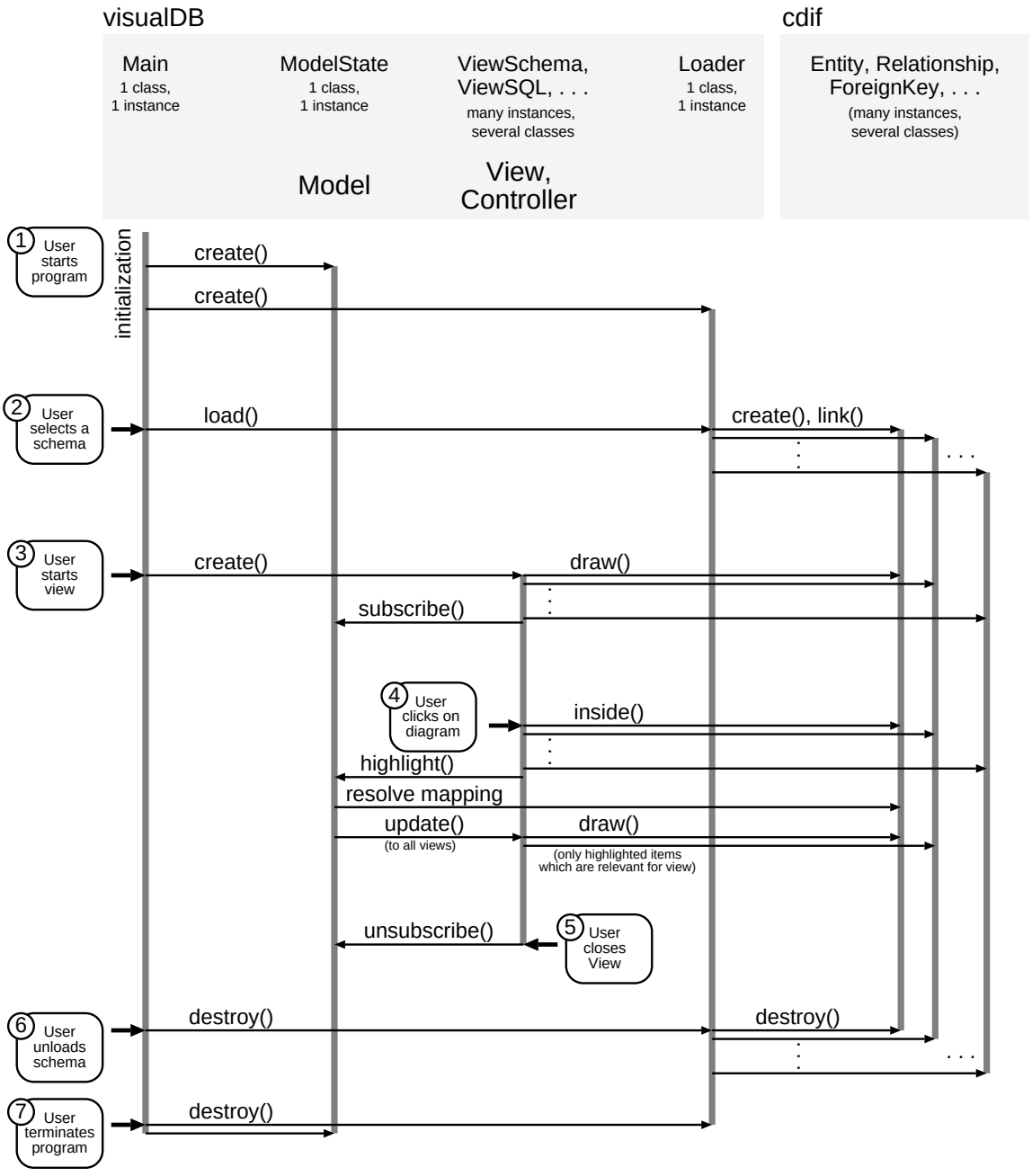


Figure 47: Message Trace Diagram, of simple example session

each of the view classes. Speaking in Smalltalk terms, the views may send messages (i.e. call class methods) to the model using a protocol (i.e. set of class methods) defined in the model. After starting a view (see step 3 in Figure 47) the new view subscribes in its constructor to the `ModelState` class. A set of such messages is listed in Figure 48. Step 4 in Figure 47 suggests that the user clicks on some object and causes a highlight message. It is important to note that if, for example, a relational object is added to the set of highlighted objects a mapping procedure adds the corresponding conceptual object as well. Step 5 of Figure 47 shows the termination of a view by unsubscribing the view and disposing of the window.

Message	**Meaning**
`subscribe( Frame )`	A view must subscribe from the `ModelState` instance in order to get notifications of state changes.
`unsubscribe( Frame )`	Before terminating itself a view must unsubscribe with the `ModelState` instance.
`hightlightAdd( SemObject )`	Add any object of the repository hierarchy to the set of highlighted objects
`hightlightAdd( Row )`	Add a row instance to the set of highlighted objects.
`hightlightAdd( PK )`	Add a key instance to the set of highlighted objects. By adding a FK, a reference instance (and therefore a relationship instance) may be included.
`hightlightAdd( PK, event )`	Add a key instance to the set of highlighted objects. The event is either insert, modify or delete and denotes an update propagation action related to a PK or a FK.
`hightlightOn( Color )`	Highlights all objects which were added by `hightlightAdd` calls with the specified color. Color constants with semantic meaning (e.g. "to be deleted" or "inserted") are used. A broadcast of `update` messages is issued by the `ModelState` instance.
`hightlightOff()`	Sets all objects which were added by `hightlightAdd` calls to the default color and removes all objects from the set of highlighted objects. A broadcast of `update` messages is issued by the `ModelState` instance.
`stepWait()`	This method returns to the calling method as soon as `step()` is called, i.e. it waits for a method invocation of `step()` by suspending its thread of execution.
`step()`	Actuates a step of the execution of triggering cascades. It causes `stepWait()` to stop waiting by resuming its thread the of execution.
`redraw()`	Forces each view to redraw its screen. It must be called after data was changed by some view.

Figure 48: Example for "Protocol" of the `ModelState` class

5.3 Repository Access

The Java package `cdif` contains a class for each meta-entity of the repository as depicted, for example, in Figure 41. The `cdif` class hierarchy is not explicitly depicted, since it would be nearly the same structure as shown in Figure 41 and 43 just in a different notation. Each EER Supertype/Subtype relation becomes an Java inheritance relationship and each EER relationship is modeled as class reference or, if necessary, a class reference array. A hypertext tree-structure of the `cdif` Java package can be found in the class documentation.

As already mentioned in the previous section the embedded SQL statements are limited to the `Loader` class. The Java mechanism of accessing databases is very new and subject to discussion and change. Sun Microsystems [73] proposed the JDBC interface [72] (a set of abstract classes) but no major DBMS vendor has yet implemented it. However, a free implementation for the Mini-SQL database server (see Section 4.4) does exist [62].

The `Loader` class loads the repository contents by establishing an object hierarchy in the main memory which resembles the data model using the refined CDIF class hierarchy (see conceptual repository structure in Figures 41 and 43). Meta-Relationships are modeled as Java object references, e.g. the class `Entity` has a data member `ActsAs` which is an array of references to objects of the class `Role`, i.e. an entity may have any number of relationship roles. These references have names similar to the names in the original CDIF structure as depicted in Figures 38 and 39. Through them any object can reach its neighbors. For example, the following Java code lists the names of all attributes belonging to the PK of `entity`:

```
for( int i=0; i < entity.isIdentifiedBy.Incorporates.length; i++ )
{
  System.out.println( entity.isIdentifiedBy.Incorporates[i].Name );
}
```

`entity` is an entity, `entity.isIdentifiedBy` its PK and `entity.isIdentifiedBy.-Incorporates` the array of attributes which constitute the PK. The complete structure of the Java package `cdif` can be browsed on WWW pages[4].

Loading of Repository

Schema loading is a two step process done by the `Loader` class through embedded SQL statements. First, all numerical identifiers of each particular class (e.g. Entity) are retrieved. For each of these identifiers a constructor of the appropriate class is called which retrieves the values for its meta-attributes. The "linkage" of the objects with object references which model the meta-relationships is done in a second run, after all objects are instantiated. Meta-relationships are CDIF meta-relationship plus the relationships for the mapping of the layers. Each of the views has a class reference to the `DataModel` object of the loaded schema as shown in Figure 46.

Due to the object-oriented approach the retrieval of data through SQL statements is fragmented into several constructor and method invocations. This is already implied by the hierarchical, relational DMOD structure as shown in Figure 42. The network access of the database is potentially via the Internet and therefore a lot slower than local access. The process of loading a database schema needs to run in its own thread (Java allows multiple threads) to maintain a good interactive behavior of the user interface. To speed up the retrieval of database schemas the result of each SQL query is written once to a file. Later on this file can be read to substitute for slow database accesses. It has to be updated whenever the repository database content has been changed.

[4]http://www.his.se/ida/msc/finished_projects/HS-IDA-MD-96-001/doc/javadoc

5.4 Application Data Access

The views which include application data require that queries are build against the application database by making use of the meta-data stored in the repository. Methods for doing this are included in the appropriate classes, e.g. `cdif.Entity`. If update propagation should be made visible in a step by step fashion it has to be executed by methods of the repository itself. As stated before, the built-in active mechanisms of a DBMS are unsuited since they do not allow a step by step execution.

The whole repository structure is instantiated at startup time. This is straight forward since the visualization programs do not allow the user to manipulate the meta-data. Changing data is, of course, allowed. Loading the data in the same fashion as the meta-data is not possible because main memory capacity is limited. Therefore, data is only instantiated as needed and freed when no longer needed. As depicted in Figure 28 (page 50) the data at the conceptual level will only be mapped from the relational data, without being actually stored in some object. Data is kept within the repository classes only at the relational level, in the same fashion as in the DBMS tables. A class `Row` is provided for that purpose. One instance of `Row` is created for each accessed row in the application database. The class offers the basic relational operators, e.g. `Row.delete`. It may instantiate and return an array of referenced rows (`Row.FKselect`). A separate class `PK` is provided for keys. This class is used by the `Row` class for PK and FK access and manipulation. It provides basic methods, e.g. for comparing keys (`PK.equals`) and for instantiating a new `Row` instance for a given key of a certain table. Some of these methods are used in the following example.

Example of update propagation

An example should give an idea of how the semantics of update propagation could be modeled within the repository class structure. The hierarchy is enriched with class methods which implement the relational semantics of update propagation. The level of abstraction for update

propagation is the abstract relational level, as discussed in Section 2.4, and involves the FK and PK role properties. A problem is that if the specific used target SQL DDL (SQL layer) does not allow certain IC or triggers, or if the mapping procedure is deficient it might be impossible to implement the simulated ideal behavior. The two-way mapping will allow these operations to be viewed at the conceptual level. Also updates to the conceptual level will be translated to their relational representation and carried out at this level.

The example to be used is a method `Delete` of the class `Entity`, which takes a key (instance of class `PK`) as a parameter. Its Java source code is listed in Figure 49. The following code is taken from a trigger definition of a cascaded delete for an INVOICE (see Figure 12).

```
--  Delete all children in "INVLINE" reference SQLPD "RELATION_29"
delete from INVLINE
 where CONTRACT = old_contract
   and MONTH = old_month
   and YEAR = old_year
```

It is recursive because other rows (of INVLINE) will possibly be deleted. If during such a cascade any update is restricted, the whole cascade must be undone. A limitation of SQL triggers is that each table may only be affected once by a triggering cascade. Therefore all updates are aborted which try to affect a table twice due to a triggering sequence. Circular references, e.g. self-references (see Figure 4, page 13), may lead to such a situation.

This abort behavior is modeled by Java Exceptions. The program defines its own exception for that purpose. The `Delete` method is called for a PK of an existing row. It calls the recursive method `delete` twice. The first time (boolean flag `really` is set to `false`) is just for highlighting all participating rows. If, during such a traversal, a table is accessed twice or an update is restricted, an exception will be raised to abort the traversal. Thereby the second invocation of `delete`, which would actually delete the rows, is skipped (see `try/catch` statement of `Delete`). This rollback mechanism requires that no other programs manipulate

```
ModelState state;


class RollbackException extends Exception
{
  public RollbackException() { super(); }
  public RollbackException( String s ) { super( s ); }
}


public void Delete( PK pk )
{
  try
  {
    delete( false, pk );
    delete( true, pk );
  }
  catch( RollbackException )
  {
    state.highlightOff();
  }
}
```

Figure 49: Java code of update propagation for deletions (part 1 of 2)

the data in between the two runs. Locking must be used, if this cannot be ensured. The `delete` method stores the complete path of the traversal explicitly in a vector (Java class of a linked list) to check whether a table was accessed twice during one cascade. If this occurs, a `RollbackException` is raised.

```
private void delete( boolean really, Row row, Vector path ) throws RollbackException
{
  Row children[];

  if( path.contains( row ) )                          // was table already modified during this cascade?
    throw new RollbackException( "Tried to modify table twice" );
  path.addElement( this );                             // add this table to call stack for future checks.

  if( !really )
  {
     state.highlightOn( row );                                  // highlight row for deletion at first time
     state.stepWait();                                                  // wait for the next single step
  }

  for( int i=0; i<ActsAs.lenght; i++ )
  {
    role = ActsAs[i];
    if( role.isPK )
    {
      children = row.FKselect( role.other.Selects.Name,
                               role.other.IsSupportedBy.Incorporates );
      switch( role.DeleteEffect )
      {
      case Role.NOACTION:
        break;
      case Role.RESTRICT:
        if( children.length > 0 )
          throw new RollbackException( "restr.  delete:  Children still exist" );
        break;
      case Role.SETNULL:
        for( int i=0; i<children.length; i++ ) children[i].FKsetnull();
        break;
      case Role.SETDEFAULT:
        for( int i=0; i<children.length; i++ ) children[i].FKsetdefault();
        break;
      case Role.CASCADE:
        for( int i=0; i<children.length; i++ ) delete( really, children[i], path );
        break;
      }
    }

    if( really )
    {
      row.delete();                                                        // actual deletion of row
      state.redraw();                                          // data-views need to update their tabulars
    }
  }
}
```

Figure 50: Java code of update propagation for deletions (part 2 of 2)

Chapter 6

Summary and Conclusions

6.1 Introduction

In this Chapter the achievements of this work are summarized by taking up the three points of the introduction in Chapter 1: (1) the three layers and their associated mappings, (2) the proposed repository structure and (3) the visualization tool design. The last will be compared with the ideas presented in [18]. After that, open issues within this work are identified. These lead to a section on possible future work. The dissertation concludes with some thoughts about future directions in the area of data modeling.

6.2 Achievements

Three layers and their associated mapping

We investigated the mapping procedure as described in the literature or used by modeling tools and identified their shortcomings and the inherent loss of semantics. The focus has been on the semantics of different kinds of relationships and how they are mapped to the relational layer (weak relationships, 1:1, Supertype/Subtype). We were concerned about the correct treatment of cardinalities specified at the EER layer and the resulting active update behavior. A classification scheme for referential integrity constraints has been introduced which allows the systematic categorization of update propagation, as a result of the mapping procedure, on the abstract relational level. This matrix may be used to classify and compare IC features of DDLs, modeling tools or DBMSs. The abstract relational level has been identified as the most suitable abstraction for the simulation of active model behavior. Only the relational level can directly incorporate real-world data from application databases. The active behavior at the EER layer is derived by the stored two-way mapping. By using an abstract relational layer we gain independence from specific DBMS features or advances in the standardization of SQL (third modeling layer). A notation for visualizing properties of references in a relational schema has been introduced.

Proposed repository structure

The above cognitions led to the implementation of a DBMS-based repository which includes the EER and the relational layer and the two-way mappings. We adopted the CDIF data modeling standard (DMOD) for structuring. Its major advantages are (1) universal applicability for the EER as well as the relational layer, (2) its extensibility, which thereby allows the inclusion of the mapping, (3) its separation between model semantics and graphical representation and (4) its hierarchical structure, which reduces redundancy and allows for convenient access from an OO host language. The repository was modeled using the S-Designor modeling tool

[61]. EER supertype/subtype relations were used to model the inheritance hierarchy.

We also used the modeling tool S-Designor later, in conjunction with the rapid prototype use for graphical model creation and automatic mapping. These models were then transfered into the repository structure developed.

Visualization tool design

We proposed a design for extensible and scalable visualization including data and meta-data which is based on the developed repository structure. We argued that scalability and comprehensibility suffer if too much information is presented in a single user interface. The Model-View-Controller paradigm was borrowed from object-oriented terminology to overcome this problem. It separates the semantic content (Model) from the presentation (View) and the interaction (Controller). Any number of different views may show the database state including data. Each view focuses on limited aspects and since all user interaction is propagated through the model, the user can see the relations between views. For example, highlighting of data model objects and changes to application data can be tracked in any view. A collection of eight different views was proposed and described in Chapter 3.

Java was used for the implementation of a rapid prototype. The Java object hierarchy resembles the hierarchical repository structure. The repository contents is loaded completely into a Java object hierarchy. All data model objects are therefore accessible and interconnected with plain object references. Views are only partly implemented and complemented with WWW pages which show the design ideas of the missing parts. The rapid prototype is neither complete nor are any claims about cognitive properties made. It is, rather, a usability test of the repository structure from the programmer's point of view and a presentation of design ideas on the WWW. However, the design can be compared with other efforts in the area which include cognitive aspects and evaluations. The TOTEM tool has already been referenced in Section 2.9 on page 53 and will be related to this work in the following paragraph.

Comparison with TOTEM, "creative data modeling"

The TOTEM tool is a "rapid 'design and test' environment for extended Entity-Relationship modeling" [18]. The tool is "primarily a vehicle for teaching and research" and used for an investigation of *creative data modeling*. It is also compared with "pen and paper" modeling and some elements of manual modeling are introduced in the tool. A cognitive evaluation is planned in the future. It "has two modes, design (and validate) and explore (and test), corresponding to design (and code) and run (and test)". The explore mode serves the same intention as the visualization in this work. However, TOTEM considers only the EER layer and EER data instances which must be kept in a "database set up for the purpose" [18]. This makes testing with real-world data difficult and may be the reason why visualization of update propagation is not considered at all in the design of the tool. Another limitation is that only a single kind of view is proposed, even though multiple instances may be spawned and configured differently. The extensible MVC approach of this work might facilitate cognitive experiments with modeling and testing tools since the user interface is decoupled from the semantics of the data model.

6.3 Open issues

This section contains some open issues and a critical review of the design decisions made.

Repository issues

- One could question the advantage of basing the repository structure on a published standard like CDIF, because data exchange with other tools has not been a goal. A structure designed directly for the intention of this project would have been simpler. In particular, the structure which stores the graphical representation of models resulted in many classes and a multitude of objects. However, a look into standards may always inspire the design and make the designer aware of the range of modeling variants.

- All layers of abstraction are stored in one structure with our design. This reduces the complexity of the implementation and worked well in this project because of structural similarity of the relational model and the Information Engineering EER notation used which allows only binary relationships with no attributes. Only the repository objects for foreign key (relational) and subtype/supertype relations (EER) were merely for one layer of abstraction. If the methods for update propagation were to be extended they would only be used for the relational layer, thereby increasing the mismatch of, for example, a "conceptual" entity and a "relational" entity, i.e. a table.

- A minor performance issue results from the hierarchical structure, since the data retrieval through embedded SQL statements is split into many sub queries which in turn results in bad performance, especially if the database is accessed via a slow network connection. A caching mechanism was implemented to ease the problem.

Status of implementation

The open issues below reflect the status of implementation and lead directly to the section on future work which follows.

- The implementation of the repository Java classes is incomplete. They lack methods which implement application data access and update propagation functionality. The design as described in Section 5.4 needs to be further detailed and implemented.

- Once the above is completed, the views which include application data as described in Section 3.4 may be implemented. In particular, the `ViewData` view (Section 5.4) is central and will require a great deal of automatic screen layout management.

- The SQL level, the third layer of abstraction, is not included in the repository. We just implemented simple markers for text positions in SQL DDL script files to allow meaningful highlighting of parts of the script.

To employ the S-Designor modeling tool turned out to be a problem, because the mapping procedure is imperfect in many respects as detailed in Section 2.7. All PK-role properties and the FK-role property *unique*[1] are not derived from the EER layer and they cannot be stored in the internal S-Designor structures, i.e. we cannot transfer them into the repository. The proposed notation for the abstract relational layer is therefore incomplete in the implemented view `ViewSchema`, if the repository is not enriched manually with this information.

[1] These concepts were introduced in Section 2.4

6.4 Future Work

The scope of the thesis is broad and issues were left open, as described in the foregoing section. The ideas presented here are possible future directions of work which result directly from aspects of this work. Section 6.5 goes beyond this by raising more general issues in the area of EER data modeling.

Complete implementation and new views

An obvious continuation of work would be to implement the missing parts as identified in the previous section. One should always keep in mind, and properly define, the purpose of such a continued implementation. New views may be added to improve schema based data exploration. Inspiration may be drawn from research and existing applications in the area of Data Mining.

An implementation of a dedicated SQL modeling layer in the repository, as for the EER and abstract relational layer, would allow detailed mapping relations from the abstract relational to the SQL layer. This would further facilitate the understanding of SQL IC and triggers, but requires a comparatively costly repository structure. An example for such a structure can be found in the SQL-92 standard [49], as mentioned in Section 2.8.

Debugging through visualization

Some directions in the area of visualization of active database behavior were described in Section 2.9 (related work) and may be considered to extend this work, for example, by letting a DBMS log file of the application database control the visualization and thereby allowing one to track actions performed by real application programs.

Implementation of mapping procedure, "abstract triggers"

One problem was the limitations of the S-Designor tool. A clean mapping with a minimal loss of semantics as described in Section 2.6 is essential for simulating different kinds of update propagation as listed in Figure 9 on page 21. To be able to change and enhance the mapping, an implementation of the procedure, possibly based on existing published algorithms (e.g. [32]) is needed. A clean, three layered approach will require some kind of "abstract relational trigger", due to the problems identified in Section 2.6 (see Figure 25 on page 41). This would be a research effort in its own right and may, as a side effect, allow the incorporation of arbitrary business rules into the EER model [75], which is useful since business rules might be a connection between entities/tables in addition to relationships/references. The MVC paradigm may lead to feasible presentations of business rules and the repository can be extended by these rules or abstract triggers.

	Repository database	Application database
Application programs	-	queries & updates
Schema based browsing	queries	queries
This visualization tool	queries	queries & updates
Future modeling/testing	queries & updates	queries & updates

Figure 51: Gradations of modeling scope

6.5 Some final thoughts on the unification of modeling and testing

The separation of a data modeling phase and a model exploration phase is inherent in our approach and can also be found in the TOTEM [18] tool. From the users point of view it would be an attractive idea to combine modeling and testing, i.e. to allow the user to gather and alter meta-data and sample data at the same time through the same views. A direct synchronization of the modeling layers at each point in time could be achieved by automatic mapping during modeling activity. Figure 51 compares different levels of modeling freedom. The last row refers to the idea presented.

Acknowledgments

The author would like to thank the supervisors Björn Lundell and Brian Lings. Björn Lundell assisted during the work in various talks, gave many valuable hints for selecting literature and gave feedback on the draft versions of the dissertation. Brian Lings provided the initial ideas for the project and gave additional valuable and encouraging feedback on the drafts. Many thanks also for correcting the English of a German.

Thanks to the M.Sc. class 1995/1996, especially to Mirko Kück for his motivation through the last weeks. Thanks also to my parents who, through their support, provided me with the opportunity of studying in Germany and Sweden.

Acknowledgments to the Department of Computer Science at the University of Skövde for providing the opportunity to study the program and write the dissertation.

Table of Abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
CAD	Computer Aided Design
CASE	Computer Aided Software Engineering
CDIF	CASE Data Interchange Format
CDM	Conceptual Data Model
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DLL	Data Definition Language
DML	Data Manipulation Language
ECA	Event-Condition-Action
EER	Extended Entity Relationship
EIA	Electronic Industries Association
ER	Entity Relationship
FIPS	Federal Information Processing Standard
FK	Foreign Key
GIS	Geographical Information System
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IC	Integrity Constraint
IDEF	Integration DEfinition for Function modeling
IE	Information Engineering
IRDS	Information Resource Dictionary System
IS	Information Systems
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JTC	Joint Technical Committee

MIS	Management Information Systems
M.Sc.	Master of Science
mSQL	Mini-SQL Database Server
MVC	Model View Controller
NF	Normal Forms
ODBC	Open DataBase Connectivity
OMG	Object Management Group
OMT	Object Modeling Technique
OO	Object Oriented
PCTE	Portable Common Tools Environment
PDM	Physical Data Model
PERL	Practical Extraction and Report Language
PK	Primary Key
RDBMS	Relational Database Management System
SQL	SQL is not an abbreviation
UoD	Universe of Discourse
WWW	World Wide Web

List of Figures

Bibliography

[1] Nathaniel K. Abablah. Capturing subtype semantics in relational database systems. Master's thesis, University of Exeter, May 1994.

[2] Russel L. Ackoff. *Creating the Corporate Future*. John Wiley & Sons, 1981.

[3] A. Aiken et al. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1), March 1995.

[4] Alligator Descartes Hermetica. DBI — a database interface module for perl5. *http://www.hermetica.com/technologia/DBI/index.html*, 1995.

[5] Applied Information Science International. Creatures — creative feature suggestion for s-designor. *http://www.aisintl.com/case/user_groups/s-designor/library/creatur.html*, 1996.

[6] E. Benazet. Vital: a visual tool for analysis of rules behavior in active databases. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, 1995.

[7] Grady Booch. The booch method — object-oriented development with java. *http://www.sigs.com/publications/docs/road/9603/road9603.c.booch.html*, 1996.

[8] Grady Booch and James Rumbaugh. *Unified Method for Object-Oriented Development.* Rational Software Corporation, 1995.

[9] John R. Bourne. *Object-Oriented Engineering — Building Engineering Systems Using Smalltalk-80.* Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992.

[10] Janis A. Bubenko and Benkt Wangler. research directions in conceptual specification development. Technical Report 91-024-DSV, SYSLAB, Department of Computer Science and Systems Science Stockholm University and SISU — Swedish Institute for Systems Development, November 1991.

[11] S. Chakravarthy, B. Blaustein, et al. Hipac: A research project on active, time-constraint database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[12] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. Eca rules integration into an oobdms: Architecture and implementation. Technical report, University of Florida, August 1994.

[13] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A visualization and explanation tool for debugging eca rules in active databases. Technical Report UF-CIS-TR-95-028, University of Florida, November 1995.

[14] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–39, March 1976.

[15] Tom Christiansen. The perl language home page. *http://www.perl.com/perl/index.html*, 1996.

[16] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[17] E. F. Codd. Extending the database relational model to capture more meaning. In *ACM Transactions on Database Systems*, volume 4, pages 397–434. ACM SIGMOD, February 1981.

[18] Alison Crerar, Peter J. Barclay, and Richard Watt. Totem: an interactive tool for creative data modeling. In *3rd international workshop on interfaces to databases*, 1996. Article is not the final version.

[19] Dansk UNIX-system Bruger Gruppe. Official home of ISO/IEC JTC1/SC22/WG22 - PCTE. *http://www.dkuug.dk/JTC1/SC22/WG22/*, 1996.

[20] Database Programming & Design. Database programming & design magazine homepage. *http://www.dbpd.com/*, 1996.

[21] DATAMATION. Datamation magazine. *http://www.datamation.com/*, 1996.

[22] C. J. Date. Referential integrity. In *Proc. 7th International Conference on Very Large Data Bases*, September 1981.

[23] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1994.

[24] C. J. Date. A note on one-to-one relationships. *InfoDB*, 3(4):295–302, 1995.

[25] Umeshwar Dayal. Ten years of activity in active database systems: What have we accomplished? In *Active and Real-Time Database Systems (ARTDB-95)*. Springer-Verlag London Ltd., 1995.

[26] DBMS. DBMS magazine. *http://www.dbmsmag.com*, 1996.

[27] Angelika Kotz Dittrich. Active database functionality in a real-world banking environment, March 1994. Main issues of a talk at the Dagstuhl seminar on active DBMS.

[28] A. Dogac, P. P. Chen, and N. Erol. The design and implementation of an integrity subsystem for the relational DBMS RAP. *IEEE*, pages 295–302, 1995.

[29] Asuman Dogac, Esen Ozkarahan, and Peter Chen. An integrity system for a relational database architecture. In *8th International Conference on the ER Approach*, pages 287–301. E/R Institute, 1990.

[30] EIA/CDIF. CDIF home page. *http://www.cdif.org*, 1996.

[31] Electronic Industries Association. CDIF goals. *http://www.cdif.org/goals.html*, 1996.

[32] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings Publ., 2nd edition, 1995.

[33] Johannes Ernst. Intro to CDIF. *http://www.cdif.org/intro.html*, 1996.

[34] Federal Information Processing Standards. Integration definition for information modeling (IDEF1X). *FIPS PUBS 184*, 1993.

[35] Stephen Ferg. Cardinality concepts in entity-relationship modeling. In *Proceedings of the ER Institute.* E/R Institute, 1991.

[36] Clive Finkelstein. *An introduction to information engineering — from strategic planning to information systems.* Addison-Wesley Publishing Company, 1989.

[37] Thomas Fors. Visualization of rule behavior in active databases. Technical Report HS-IDA-TR-94-009, Department of Computer Science, University of Skövde, 1994.

[38] Howard Gardner. *The mind's new science — A history of the cognitive revolution.* Basic Books Publishers New York, 1985.

[39] J. S. Gero. Design prototypes: A knowledge representation schema for design. *AI Magazine*, 11(4):27–36, 1990.

[40] Mark Gibson. HTML clickable maps for LBL-tool schemata. *http://agave.humgen.upenn.edu/lens/schema/lensschema.html*, 1996.

[41] Adele Goldberg and David Robson. *Smalltalk-80 — The Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

[42] David JL Gradwell and Peter Gradwell. Home page of the ISO/IEC JTC1/SC21/WG3 - IRDS. *http://www.irds.org*, 1996.

[43] Terry Halpin. Object-role modeling: Niam and beyond. In *The 13th International Conference on the Entity Relationship Approach Business Modeling and Re-Engineering.* The E/R Institute, 1994. Slides for a tutorial held on the 13th December.

[44] Rob Hill. CDIF — integrated CASE meta-model, data modeling subject area, cdif-draft-dmod-v13. The document represents ongoing work of the CDIF technical committee. it is not a approved standard, July 1995.

[45] Hughes Technologies Pty Ltd. Mini SQL home page. *http://Hughes.com.au/product/msql/*, 1996.

[46] Valerie Illingworth et al. *Dictionary of Computing.* Oxford University Press, 1990.

[47] JCC Consulting, Inc. SQL standards home page. *http://www.jcc.com/sql_stnd.html#Current Status*, 1996.

[48] Joint Technical Committee ISO/IEC JTC1/SC21/WG3. Information processing systems — concepts and terminology for the conceptual schema and the information base. Technical Report ISO/TR 9007:1987, ISO, 1987.

[49] Joint Technical Committee ISO/IEC JTC1/SC21/WG3. Database language SQL. *Document ISO/IEC 9075:1992 and ANSI X3.135-1992*, 1992.

[50] Joint Technical Committee ISO/IEC JTC1/SC21/WG3. DBL: MCI-004 and X3H2-96-059. *ISO Working Draft SQL/Foundation*, 1996.

[51] Pericles Loucopoulos and Vassilios Karakostas. *System Requirements Engineering*, chapter Chapter 6: Case Technology, pages 140–156. McGRAW-HILL, 1995.

[52] D. Lucarella and A.Zanzi. Visual retrieval environment for information systems. *ACM Transactions on Information Systems*, January 1996.

[53] Victor M. Markowitz and Arie Shoshani. An overview of the lawrence berkeley laboratory extended entity-relationship database tools. Technical report, Lawrence Berkeley Laboratory, 1993.

[54] J. Martin and C. McClure. *Recommended Diagramming Conventions for Analysts and Programmers*, pages 250–275. Prentice Hall, 1985.

[55] James Martin. *An Information Systems Manifesto.* Prentice Hall, 1984.

[56] Jim Melton and Alan R. Simon. *Understanding the New SQL (SQL-92).* Morgan Kaufmann, 1993. Description at http://Literary.COM/mkp/pages/2453/index.html.

[57] G. M. Nijssen and T. A. Halpin. *Conceptual Schema and Relational Database Design — a fact oriented approach.* Prentice Hall, 1989.

[58] D. Norman. *Things that make us smart.* Addison-Wesley Publishing Company, 1993.

[59] Patrick E. O'Neil. *Database Systems: Principles, Programming and Performance.* Afterhurst Limited Publishers, 1994.

[60] Joan Peckham and Fred Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3), September 1988.

[61] Powersoft. *S-Designor DataArchitect 5.0 User's Guide*. Powersoft Corp., 1996.

[62] George Reese. Imaginary JDBC driver for mSQL. *http://www.imaginary.com/ borg/Java*, 1996.

[63] Laila G. Robinson and Hunter Shu. On pendant reference. Technical Report X3H2-86-87, ANSI X3H2, July 1986.

[64] Hunter Shu. Referential integrity based on existential dependency concepts. Technical Report X3H2-86-59, ANSI X3H2, May 1986.

[65] Herbert Simon. *The science of the artificial*. Basic Books Publishers, New York, 1985.

[66] Graeme C. Simsion. Creative data modeling — encouraging innovation in data design. In *The 10th International Conference on the Entity Relationship Approach*. The E/R Institute, 1991.

[67] Jacob Stein and David Maier. Concepts in object-oriented data management. *Database Programming & Design*, 1(4), April 1988.

[68] Michael Stonebraker et al. Third generation database system manifesto. *ACM SIGMOD*, Record 19(3), September 1990.

[69] Veda C. Storey, Heng-Li Yang, and Robert C. Goldstein. Semantic integrity constraints in knowledge-based database design systems. *Data & Knowledge Engineering*, 1996.

[70] Sun Microsystems, Inc. All about java(tm). *http://java.sun.com/aboutJava/index.html*, 1996.

[71] Sun Microsystems, Inc. Java(tm) — programming for the internet. *http://java.sun.com*, 1996.

[72] Sun Microsystems, Inc. The JDBC(tm) database access API. *http://splash.javasoft.com/jdbc/*, 1996.

[73] Sun Microsystems, Inc. Sun microsystems, inc. *http://www.sun.com*, 1996.

[74] Sybase, Inc. Designor family of products. *http://www.powersoft.com/products/-design/des_fam.html*, 1996.

[75] Asterio K. Tanaka, Shamkant B. Navathe, Sharma Chakravarthy, and Kamalakar Karlapalem. ER-R: an enhanced ER model with situation-action rules to capture application semantics. In *Proceedings of the ER Institute*. E/R Institute, 1991.

[76] Adrienne Tannenbaum. *Implementing a Corporate Repository*. John Wiley & Sons, Inc., 1994.

[77] Laura Tarantino. Hypertabluar representations of database relations in world wide web front-ends. Technical report, Dipartimento de Ingegneria Elettrica, Università degli Studi dell' Aquila, 1996.

[78] U.S. Geological Survey. Earth and environmental science. *http://www.usgs.gov/network/science/earth/gis.html*, 1996.

[79] Jennifer Widom, Stefano Ceri, and Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.

[80] Paul Winsberg. Dictionary standards: ANSI, ISO and IBM. *InfoDB*, pages 12–21, Winter 1988/89.

[81] World Wide Web Consortium. Introducing HTML 3.2. *http://www.w3.org/pub/WWW/MarkUp/Wilbur*, 1996.

[82] World Wide Web Consortium. World wide web consortium (w3c. *http://www.w3.org/pub/WWW*, 1996.