Bernhard Thalheim
Kuwait, Rostock, Cottbus
1988 - 1999

Database Design
based on the
Higher-Order Entity-Relationship Model
with the tool boxes
(DB)$^2$, ID$^2$ and RADD

6. Oktober 2013
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Higher-Order Entity-Relationship Model</td>
<td>3</td>
</tr>
<tr>
<td>Database Design Using the Higher-order Entity-Relationship Model</td>
<td>4</td>
</tr>
<tr>
<td>Doing the split: Putting ER theory into practice</td>
<td>20</td>
</tr>
<tr>
<td>Intelligent Database Design using an Extended Entity-Relationship Model</td>
<td>34</td>
</tr>
<tr>
<td>Strategies and Methodologies of Database Design</td>
<td>75</td>
</tr>
<tr>
<td>Supporting Entity-Relationship Database Design Strategies</td>
<td>76</td>
</tr>
<tr>
<td>Towards Theory-Based Database Design Strategies</td>
<td>90</td>
</tr>
<tr>
<td>Towards a Framework for Database Design Strategies</td>
<td>110</td>
</tr>
<tr>
<td>Das Abstraktionsschichtenmodell für den integrierten Entwurf von Dialogen, Sichten, Funktionen und Schemata von Datenbankanwendungen</td>
<td>130</td>
</tr>
<tr>
<td>Human-Computer Interaction in Heterogeneous and Dynamic Environments: Requirements and Conceptual Modeling</td>
<td>136</td>
</tr>
<tr>
<td>Natural Language Descriptions Mapped To Database Schemata</td>
<td>192</td>
</tr>
<tr>
<td>Applying a Natural Language Dialogue Tool for Designing Databases</td>
<td>193</td>
</tr>
<tr>
<td>Designing EER-Skeleton Schemes based on Natural Language</td>
<td>209</td>
</tr>
<tr>
<td>From NL DB Request to Intelligent NL DB Answer</td>
<td>234</td>
</tr>
<tr>
<td>Semantics and Acquisition of Constraints</td>
<td>250</td>
</tr>
<tr>
<td>Semantikakquisition im Datenbankentwurf und Reverse-Engineering</td>
<td>251</td>
</tr>
<tr>
<td>An Informal and Efficient Approach for Obtaining Semantic Constraints using Sample Data and Natural Language Processing</td>
<td>263</td>
</tr>
<tr>
<td>Ein Werkzeug zur Gewinnung semantischer Constraints aus natürlichsprachlichen Eingaben und Beispielen</td>
<td>288</td>
</tr>
<tr>
<td>Semantische Analyse im RADD-NLI: Zur Interpretation von Possessiv- Relationen</td>
<td>308</td>
</tr>
<tr>
<td>Reuse of Database Design Decisions</td>
<td>317</td>
</tr>
<tr>
<td>Optimisation and Database Performance Tuning</td>
<td>341</td>
</tr>
<tr>
<td>A Computational Approach for Conceptual Database Optimization</td>
<td>342</td>
</tr>
<tr>
<td>A Computational Approach to Conceptual Database Optimization</td>
<td>359</td>
</tr>
<tr>
<td>The Tool Box (DB)$^2$ Database Design by Beta</td>
<td>364</td>
</tr>
<tr>
<td>The Database Design System (DB)$^2$</td>
<td>365</td>
</tr>
<tr>
<td>Design with the Database Design System (DB)$^2$</td>
<td>376</td>
</tr>
<tr>
<td>The Tool Box RADD (Rapid Application and Database Design)</td>
<td>394</td>
</tr>
<tr>
<td>Die Entwicklung einer Datenbankentwurfsumgebung der dritten Generation: RADD – Rapid Application and Database Development</td>
<td>395</td>
</tr>
<tr>
<td>The Design of RAD: Towards an Interactive Toolbox for Database Design</td>
<td>407</td>
</tr>
<tr>
<td>Information Services</td>
<td>422</td>
</tr>
<tr>
<td>Conceptual Design and Development of Information Services</td>
<td>423</td>
</tr>
</tbody>
</table>
The Higher-Order Entity-Relationship Model
Database Design Using the Higher-order Entity-Relationship Model*

Bernhard Thalheim

Computer Science Institute
Cottbus Technical University
K.-Marx-Str. 17
D-03013 Cottbus, FRG
thalheim @ informatik.uni-rostock.dbp.de

Abstract

Database design is one of the most difficult tasks. Usually, a high level abstraction level, various skills and a deep knowledge is required from the designer. However, the situation can be substantially improved if the known database theory, techniques known in artificial intelligence and visualization and interface methodologies are used in a proper way. The reachability of this goal is demonstrated in the paper. We extend the Entity-Relationship Model to the Higher-order Entity-Relationship Model (HERM) by adding relationships of higher degrees, semantical constraints, operations and constructs for the behavioral description of the database. This model is used for the representation of the whole designers information in the design workbench \((DB)^2\). The system \((DB)^2\) does not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The system supports an efficient translation to nested relational, relational, network and hierarchical schemes according to different environments. The system supports different design methodologies one of those - design by units - is discussed in the paper.

1 Introduction

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

- to meet all the information (content) requirements of the entire spectrum of users in the given application area;
- to provide a ”natural” and easy-to-understand structuring of the information content;
- to conserve the whole semantic information of the designers for a later redesign;

*The research has been supported by the Kuwait University Research Grant SM 057.
1 INTRODUCTION

- to achieve all the processing requirements and achieve a high degree of efficiency of processing;
- to achieve the logical independence for query and transaction formulation on this level;
- supporting bad schemes and good for the manipulation of schemes that violate normalization, nonredundancy, etc.;
- coping with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user;
- designing for consistency if the same type of information appears in several design phases.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process in such a formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

Database designers have to have a deep knowledge in database theory especially normalization theory, a good understanding of the application area and a good intuition on the implementational restrictions of the environment and on the programming problems. Doing this split is in most cases intractable for a single designer or even groups of designers. However, most of the database theory is well formalized. The systems can be formally characterized. Therefore, this knowledge can be used for the automatic support and the simplification of the design task.

The design system \((DB)^2\) captures a lot of information about schemes under design. It has a data dictionary in which schema information can be expressed and transformed and is based on a database model that can support all phases of the design process. Nowadays, the design process is understood to capture the structural design and the modelling of the semantics, as well as the description of the behavior of the database, especially the operations defined for the database. We evolved the classical entity-relationship model to the higher-order entity-relationship model (HERM) which can support design in any of the main classical data models and higher order data models and also translation among them. It supports integrity constraints. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated.

At present the system \((DB)^2\) is widely used for database design. There is a large number of active groups which are using the system.
For the effective utilization of the system, the system should support different design methodologies. The proposed system supports effectively several methodologies. There are several design methodologies [3, 6, 9, 10, 19]. Top-down and bottom-up design methodologies with/without graphical representation are widely discussed in the literature. There are also some design methodologies which are developed for specific models. Most of them are general methods for the database design using the waterfall design approach known in software engineering. In AI and other branches of Computer Science other methodologies are developed. For knowledge acquisition, for example, interview techniques are applied. The design-by-units design methodology is a technique which could be understood as a step-by-step reification and generalization method. The designer develops first the design for the basic, independent units. Then these units are considered together for the design of more general units. This methodology is very general. Among the methodologies supported by the design system \((DB)^2\) this methodology was the most frequently used.

The paper is divided into three parts. First we introduce the object-oriented entity-relationship model, the higher-order entity-relationship model. Then we show how this model is to be used in the design system \((DB)^2\). In the last part we illustrate the design-by-units design methodology using an example.

2 The Object-Oriented Entity-Relationship Model

2.1 The Object Model

Objects are to be handled and modelled in databases. They can own an object identifier and are to be characterized by values and references to other objects, i.e.

\[ o = (i, \{(s, v)\}, \{ref\}) \]

The value characterization is to be bound to an already defined structure \(s\). Characterized properties of objects are to be described by attributes which forms the structure of the object. Objects have furthermore a special semantics and a general semantics. Operators are to be associated to objects. These operators have a behavior. Object which have the same structure, the same general semantics and the same operators are be collected in classes. The structure, the semantics and the operations of a class is represented in types. Modelling of objects includes in this case the association of objects to classes \(C\) and their corresponding value type \(T\) and reference type \(R\).

Therefore, objects are to be represented by \(o = (i, \{(C, T, v)\}, \{(C, R, ref)\})\).

The known design methodologies vary in the scale of information to be modelled in the types. If objects in the classes can be distinguished by their values then the identifiers can be omitted and we use value-oriented modelling. In the other case, we use an object-oriented approach. In the object-oriented approach different approaches can be distinguished. If all objects are identifiable by their value types or by references to identifiable objects the database is called value-representable. If the database is not value-representable then we have to use object identifiers. It is well-known that in this case either the identifier handling should be made public or the databases can not be updated and maintained. Therefore, value-representable databases are of special interest.
Normally, objects do not exist in a database independently. An object is to be called *kernel objects* (or independent) if its existence in the database is independent of the existence of any other object in the database. An object is called *characteristic* if it describes some other object. Objects can perform a superordinate role in interrelating other objects, in which case they are called *associative*. The exists *associations* among objects. Associations can be also objects.

Kernel objects are to be described by *entities* in the valued-oriented approach. All other object can be entities or *relationships*. Kernel objects can be distinguished by their values of some attributes. These attributes are called *key*. In value-representable databases objects are kernel objects if they are identifiable by their values. These objects are represented by entities. All other objects are to be represented by relationships.

The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [11]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types. The clear distinction of independent objects from dependent objects is another advantage of the HERM model and the system \((DB)^2\). This simplifies the modeling since we do not need additional constructs for subtyping.

### 2.2 The Higher-Order Entity-Relationship Model

We introduce now the higher-order entity-relationship model. Besides the existence of a strong theoretical basis there are several other advantages of the HERM approach:

- HERM-schemes are much simpler and are easier understandable than ERM-schemes.
- HERM-schemes support abstraction in a simple but comprehensible manner.
- HERM-schemes can be translated together with the corresponding constraints, with the corresponding user-defined operations and with the generic operations to normalized relational, hierarchical or network schemes.

The HERM type consists of information on the structure, (static) semantics, operations and behavior (dynamic semantics), i.e.

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior}.
\]

This notation can be generalized to the more general which is out of scope for the purposes of this paper

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior} + \text{Environment}.
\]

The higher-order entity-relationship model has the following modelling constructs:

**Simple attributes** For a given set of domains there are defined attributes and their corresponding domains.

**Nested attributes** Using basic types complex attributes can be defined by means of the following constructors:
• Tuple constructor. Using nested attributes a new nested attribute is defined by the cartesian aggregation.
• Set constructor. Using a nested attribute a new nested attribute is defined by the set aggregation.

Additionally, the bag and the list constructors can be used. For the sake of simplicity we use here only the tuple and set constructors.

Entities Entity types are characterized by their attributes. Entity types have a set of attributes which can be used as the identifier of the type.

Clusters The union of types is called cluster.

First-order relationships First-order relationships types are defined to be associations between entity types or clusters of those. They can be additionally characterized by attributes.

Higher-order relationships The relationship type of the order \(i\) is defined as an association of relationship types of order less than \(i\) or entity types and can be additionally characterized by attributes.

Integrity constraints A corresponding logical operator can be defined for each type. A set of logical formulas using this operator can define the integrity constraints which are valid for each instance of the type.

Operations Each type can have a set of (conditional) operations and query forms. Furthermore, generic operations like projection, restrictions, insert, (tagged) clustering are predefined for each type.

Let us consider examples of these constructs in a medical database. This database covers information: on known diseases (symptoms, drugs which may aggravate, cause or cure the disease), known drugs with their components and patients, physicians and other persons.

A name of a person is defined by the cartesian aggregation
\[ \text{Name(First, Fam)} \]
The membership in societies can be defined by the set aggregation attribute
\[ \text{Membership}\{\text{Member(Society,Since)}\} \]
The address of persons is usually defined to a complex attribute, for example
\[ \text{Address(State, City(Code, Town), Street(Name, House(Numb, Appartm))}} \]
A person can be characterized by its social security number, its name, its address and the sex and has the set \{ SSN \} as the identifier, i. e.
\[ \text{Person} = (\{ \text{SSN, Name(First, Fam), Adr(Zip, Town, Street(Name, Nr)), Sex} \}, \{\text{SSN}\}) \]
The relationship cure of patients can be modelled by
\[ \text{Cure} = (\text{Patient}, \text{Physician}, \text{Drug}, \text{Disease}, \{\text{Date}\}) \]
An example of an integrity constraint over \( \text{Cure} \) is the following predicate
\[ \text{CorrectCure} := \text{Cure(Patient)} \neq \text{Cure(Physician)} \]
which indicates that nobody is curing himself. We can also use labels instead of the \((\text{Person},1)\).

The function
\[
\text{FamilyName}(x) := (\text{Person}(x)[\text{Name}])[\text{Fam}]
\]
is an example of an operation defined for the type \text{Person}.

The function \(\text{Doctor}(x) := (\text{Select}(\text{Person}(\text{Name})=x)(\text{Cure}))[\text{Physician}(\text{Name})]\)

generating the physicians of a given person is an example of an operation which involves more than one type. Such operations can be used to define roles of relationships.

Operations can be defined using preconditions and postconditions. If, for instance, the types

\[
\text{Person} = (\{ \text{ID}, \text{Name}, \text{Address} \}) \quad \text{and its subtype} \\
\text{Physician} = (\text{Person}, \{ \text{LicenceNr}, \text{Salary}, \text{Dept} \})
\]

are defined and any physician is a person which is stored in the database then the insertion operation for the type \text{Physician} can be generically defined with one precondition or with a post-condition based on the operation \text{Insert}:

\[
\text{Add1}(\text{Physician},(x,y,z,u,v,w)) := \text{if Person}(x,y,z) \text{ then Insert}(\text{Physician},(\text{Person}(x,y,z),u,v,w)); \\
\text{Add2}(\text{Physician},(x,y,z,u,v,w)) := \text{Insert}(\text{Physician},(\text{Person}(x,y,z),u,v,w)) \\
[\text{if not Person}(x,y,z) \text{ then Insert}(\text{Person},(x,y,z))].
\]

The HERM differs from the classical ERM and from extended ERM in several constructs.

1. Constructs which can be found in different extensions of the classical ERM are: nested attributes, first-order relationships of higher arities, clusters and some integrity constraints like the complexity.

2. The following constructs are new: higher-order relationships, integrity constraints including the generalized complexity, operations and conditional operations.

3. Since weak entity types and Is-A-relationships can be represented directly in the HERM there is no direct need for these constructs in the HERM.

4. Kernel objects are distinguished from other objects and differently represented by entity types. All other objects are to be represented by relationship types. For this reason the schemes in HERM are simpler several times than ERM schemes.

The entity-relationship (ER) model is one of the most popular database design models. Despite numerous positive features of the ER approach there still exists a strong need for a theoretic basis. This theory must be able to define sets of semantically well-formed ER schemes for particular user-specific ER-techniques as well as for subproblems as scheme design, view integration, query generation, and scheme transformation. Codd [5, p. 477] states even that the entity-relationship model "is clearly the winner in terms of its lack of precise definitions, lack of clear level of abstraction, and lack of mental discipline". In [17] the entity-relationship model is extended and it is shown that a precise, formal definition exists. This is based on the HERM-methodology which
3 HERM AND THE DATABASE DESIGN SYSTEM

is using an abstract-data-type-approach.

3 HERM and the Database Design System

The system \((DB)^2\) (Data Base Design by Beta: \(DBDB = (DB)^2\)) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. \((DB)^2\) supports database design from requirements analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual design, thus offering a choice between the relational, network, or hierarchical models for the logical design. Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can be used for creating query compilers which are more productive, effective and forceful.

One of the most important advantages of the system is the interactive user-driven input of information. It is used an easily intelligible, comprehensible and understandable “fill in the blanks” input procedure which is required in the literature as the ideal input form.

The tools developed in \((DB)^2\) can be divided into the following groups:

1. Analyzers (for reports, schema checking, the normalization, access evaluation tools).
2. Transformers (for new schemes which are information content equivalent).
3. Heuristic, user-dialog-driven tools (for solving view cooperation problems).
4. Translators appropriate for the target data model (relational, network or hierarchical) and system.
5. Query definition module (which is also used for behavioral normalization).

The general system architecture is represented in figure 1 where thicklines denote the data flow and thinlines the control flow.

4 Methodologies and Design-By-Units

The system is based on the HERM methodology. In database modeling three different perspectives can be identified. Different models stress only some of those. The structure-oriented, semantic perspective focuses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The process-oriented perspective is concerned with the processes or activities performed in the application area. The behavior-oriented perspective is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The HERM methodology supports this approach and is an extension of [1, 4, 9, 10]. We start with meaningful examples (the first step), develop the structure and the semantics of the database under design (the next four steps), generate the operations and model the behavior of the database (the next two steps). Throughout
the procedure, checks are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational design can require for performance reasons the redesign of the structure. The eight steps can be defined as follows:

1. transform examples in elementary entity and relationship types, analyze the structure;
2. graphical representation of types and their associations, population check;
3. eliminate surplus, derivable types, determine roles;
4. modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. normalize the types;
6. modeling of operators over the types and the behavior;
7. optimizing types and deriving views;
8. check the completeness, consistency of the design with the examples.

This methodology is directly supported by the design system $(DB)^2$.

The HERM and $(DB)^2$ methodologies generalizes different approaches presented in the literature (see for instance [6, 10, 14]). The most important classes of integrity constraints used in
the database modeling system [15] are the functional dependencies, the generalized functional dependencies, the multivalued dependencies, the inclusion dependencies, the exclusion dependencies. In $(DB)^2$ there are implemented three different choices for the translation of Is-A- and generally (0,1)-relationships. The design system $(DB)^2$ is able to change from one representation to the other.

The system can be used for the support over the complete lifecycle of database systems. Most known methodologies are not well adapted to the lifecycle because the design information is not used after the design process. Design design is a far more iterative process as captured in the straight forward lifecycle model of these methodologies. Using $(DB)^2$ during the whole system development the complete design information is usable for restructuring the database. This makes it possible to restructure the database and during restructuring to recompile in accordance to the old and new structures the programs which are specified in the HERM algebra.

The behavior can be specified using generic operations like insert and deriving restrictions to the behavior from the semantical information. The maintenance complexity is one of the most important behavioral database design criterias. The database designer can design the structural, semantical, operational and behavioral information on the application area and the system generates in accordance to the DBMS’s type and the distribution the corresponding data structures and transactions.

Database design is at present considered as top-down design. This approach is only useful for single-designer support systems. However, in the case the designer can detect only in a restricted manner similarities between the concepts. This is especially useful if the design is made by several designers. In this case the view integration is one of the most difficult problems which has to be solved for an efficient management of the whole database system. This task can not be solved using only methods of top-down design. The window manager approach in $(DB)^2$ can be used to detect similarities. Moreover, designing the process information at the same time adds the possibility to adapt the structure to the process efficiency. For this reason, new normalization algorithms are used in $(DB)^2$ which prohibit normalization if this contradicts the process efficiency.

In [20, 18, 17] another new design methodology was developed: Design-by-units. This methodology supports directly the above discussed distinction between kernel and dependent object. It is a recursive method. Each step is based on the above discussed eight design steps.

**Design-by-units - Structural design**

1. **Basic step.**
   Design the types for the independent kernel object classes (units). Design a raw skeleton on units. Develop a raw specification on interfaces. The checkpoint after this step examines the interface description and the skeleton. Since the schema is based on the HERM approach the skeleton is non-cyclic. Further, the skeleton is checked according to the completeness of structures. The skeleton is structurally complete if there any further relationship type is a relationship type which can be defined inside a unit.
   
   Agenda1 := \{units\}  
   Agenda2 := \{unit − schema\} .

2. **Recursion step.**
   Repeat until Agenda1 := ∅ .
   Either reification :
Choose \( u \in \text{Agenda}1 \).

i. Refine the units introducing subtypes (in HERM represented by unary relationship types).

ii. Refine the relationship types which use \( u \) as a component according to new subtypes.
Change if necessary \( \text{Agenda}2 \) and the interface of the unit \( u \).

(b) \( \text{Agenda}1 := \text{Agenda}1 \setminus \{u\} \)

or choose \( u \in \text{Agenda}1 \) and design \( u \) applying the design-by-units strategy and then \( \text{Agenda}1 := \text{Agenda}1 \setminus \{u\} \)

or generalization of units:

(a) If there are associations among units then introduce a new unit containing the associated units and the relationship type according to the association.

(b) Add other relationship types if new associations exist.

(c) Refine \( \text{Agenda}2 \).

3. Design relationships among units in \( \text{Agenda}2 \).

The process design and the structural design need to be integrated. We use a dataflow approach [3]. A process is an activity of the information system. The dataflow is an exchange of information between processes. The processes use information from the database and create temporary databases necessary for the process. Since processes use different databases and these databases are usually not fully integrated, interfaces are to be used for view cooperation. Generally, the interface is the description of the cooperation between different users (originator/receiver of the dataflow).

The processing-requirement modeling at the data level has several applications:
1. **Completeness and consistency checking of information requirements:** One requirement in conceptual database design is that the information requirements of a user group are complete and semantically consistent (not conflicting) with the processing requirements. A process model can be used to verify the completeness and consistency of other requirements.

2. **Identification of proper data entries:** Processing requirements can add different temporal and persistent database structures to the schema. In this case the view integration is more complex.

One of the difficult tasks in processing modeling is to evaluate whether the designed data structures are appropriate for an effective processing of data. It is known already that relational normalization can contradict effective processing. Sometimes unnormalized relations can be used simpler. For handling this we need a cost model for processes. The cost model can be based on models of complexity for operations and on priority functions for queries and transactions. Therefore, we need a representational and a graphical language for the representation of processes. We decided to use three different languages, one high-level abstract language which is directly supported by the modular design-by-units-strategy, another one for a more detailed procedural description and and another more dataflow oriented which shows the level of the database directly.

Let us first illustrate the structural part of this method using the above mentioned medical example. First we design the basic units and then we relate these basic units like the units PERSON and DISEASE in figures 2 and 3. Then we merge these units obtaining larger, more general units. One relationship type which is to be designed during the merge of the units PERSON, DISEASE and DRUG is the type Cure. In the last iteration the schema in figure 4 is designed.

![Figure 3: The unit DISEASE](image)

Let us extend the previous design by a units Pharmacy and CityPharmacy. Then we can model the complex transaction RequestADrugFrom Pharmacy. If the drug is available in the departments pharmacy then the physician is informed. If not then the pharmacies of other hospital units are
Figure 4: HERM-Diagram of the Medical Application Database
consulted and requested. If the drug is not available in the hospital the central pharmacy is consulted. The central pharmacy use its store or requests the supplier. The drug is then send back to the hospital with an invoice. If the drug is not available the physician is to be informed.

For modeling of processes we have developed different languages which can be considered as a seria of abstract specification languages. A process specification written in an algebraic language can be transformed to the process specification in procedural language and can be represented in graphical transaction language based on the graphical process language of [3].

The specification of the above presented example would lead to the following algebraic process specification:

\[
\text{Request} := \text{get Drug from DirectPharmacy } \cup \text{HospitalPharmacy } \cup \text{CityPharmacy}
\]

where the exclusive ordered union is denoted by \(\cup\).

This leads to the procedural process specification

\[
\text{Request} := \begin{cases} \text{if AvailDrugPharmacy then AvailInfo} \\ \text{else if AvailAnotherHospPharmacy then MakeListToPharmacy} \\ \text{else OrderFromCityPharmacy} \end{cases}
\]

The unit \(\text{Pharmacy}\) has also an active operation

\[
\text{Active MakeOrder} := \text{activation RequestPharmacy } \neq \text{Empty} \\
\quad \text{if DrugAvailable then ProceedRequestByPharmacy} \\
\quad \text{else OrderFromCityPharmacy}
\]

and the transaction with the complete description

\[
\text{ProceedRequestByPharmacy}(d,doc,ord) := \\
\quad \text{Update(DrugOthPh,-d); DeleteRequPhar (ord); AvailInfo(doc)}.
\]

## 5 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system \((DB)^2\) was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and system have the following advantages:

1. The model is easy understandable, simple and perceivable.

   The model can be used as a basis of database design tools [15, 18]. The system \((DB)^2\) is used at present by more than 95 user groups.

   The modeling approach is simple to use even for large problems.

   Since the model uses graphical representations the modeling result is easier to understand and visible.
In an experiment, 20 novice or end-user database designers learned the HERM methodology and later designed different database schemes in different areas. Our experience was that the methodology was easily accepted, led to more accurate, to less redundant schemes and to a correct modeling of complex relationships.

Using query forms [20] the user can specify application transactions and programs on the top of the HERM which reduces substantially the complexity of application programming.

2. The results of the design are much simpler than in other approaches.

We have used the model for modeling also some more complex applications. One observation is that the obtained schemes are from three to five times simpler than those obtained by other models. The example of [14] is simplified by four times and can be placed on one page or one screen. In other examples, the simplification makes it possible to find a model. Using this modeling approach, an airport counter application was modelled by less than 40 entity types and less than 120 relationship types whereas the original solution with more than 150 entity types and more than 400 relationship types was unacceptable by users because of complexity and non-transparency.

The simplification leads also to a better understanding of the application and makes normalization easier to perceive.

The schemes avoid additional redundancy. Using HERM, the normalization and the minimalization of schemes can be considered together.

3. The utilization of $(DB)^2$ is more natural and simpler. Only necessary facts are to be expressed.

The model supports a direct translation to the three classical database models. This translation preserves normal forms. Since a direct translation to relational, network and hierarchical schemes can be used the design decisions directly could be used to obtain schemes in normal forms. The translation theory can be used for a multimodel- and multisystem-support [20] and presents a practical solution to interoperability of systems.

The HERM algebra is used for query definition. The corresponding relational, network or hierarchical queries can be automatically generated.

The model supports a rich set of constraints. These constraints are used for the development of the scheme equivalence. Although the excessive number of fact-encoding mechanisms means that the same semantic unit can be declared in many syntactically different and compatible ways, the information described is equivalent. This equivalence theory can be used for automatic modification of schemes [2].

The database maintenance procedures can be derived using the design information.

Using a knowledge base previous and system-provided design decisions can be reused or partially reused what simplifies the design task. Furthermore, similarities in the design can be detected and used for simplification of the implementation.

Using the whole design information the retranslation of application programs can be used for the adaption of existing database systems to changing environments.
4. The model and the design system have a strong theoretical basis.

The model is based on a multitype logic which is equivalent to the first-order predicate logic.
For this reason, results known from discrete mathematics and relational theory [16] can be
used.

The model covers the complete modeling information. The structure, static semantics,
generic and user-specified operations and behavior of an application can be described by
the model.

The theory is simplified and cleaned up. Sequences, subsets and powersets of objects can be
modeled directly. Is-A-Relationships are treated in a better way. Weak entity types can be
avoided completely. A normal form theory is developed for the HERM. Using this normal
form theory, we can obtain normalized schemes like in the classical theory.

5. The theory is applicable to practical needs.

Based on the theory a multi-paradigm, robust design methodology is developed [18, 20] which
encorporates approaches known in object-oriented modeling [12], modular programming [15]
and programming in large.

Using this modeling approach, a view cooperation concepts was developed. Since full view
integration is not decidable and not axiomatizable view cooperation is the only applicable
approach.

The approach can be used for reverse engineering. Systems and programs developed for one
management system can be recompiled and adapted to other management systems.

Acknowledgement

I would like to thank Sabah Al-Fedaghi, Mustafa Yaseen and Peter Bachmann for their stimulating
discussions.

Further, I have to thank my students in Kuwait which implemented the main parts of (DB)².
The ideas of this paper have been prepared during my stay in Kuwait University from 1988 until
1990. The system reported in this overview has been build together with students in Kuwait, has
been extended in Rostock and is still widely applied. The first draft version of this paper was
co-authored by Fahmi Ahmad Zaki Iftaiha.

References

toolbox for database design. RWTH Aachen, Fachgruppe Informatik, Aachener Informatik-Berichte,
Benjamin Cummings, Redwood, 1992.
REFERENCES


Abstract

Database design is one of the most difficult tasks. Usually, a high level abstraction level, various skills and a deep knowledge is required from the designer. However, the situation can be substantially improved if the known database theory, techniques known in artificial intelligence and visualization and interface methodologies are used in a proper way. The reachability of this goal is demonstrated in the paper. We extend the Entity-Relationship Model to the Higher-order Entity-Relationship Model (HERM) by adding relationships of higher degrees, semantical constraints, operations and constructs for the behavioral description of the database. This model is used for the representation of the whole designers information in the design workbench \((DB)^2\). The system \((DB)^2\) does not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The system supports an efficient translation to nested relational, relational, network and hierarchical schemes according to different environments. The system supports different design methodologies one of those - design by units - is discussed in the paper. Design by units is a truly object-oriented methodology following the main principles of object-oriented thinking.

1 Introduction

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

- to meet all the information (content) requirements of the entire spectrum of users in the given application area;

- to provide a "natural" and easy-to-understand structuring of the information content;
- to conserve the whole semantic information of the designers for a later redesign;
- to achieve all the processing requirements and achieve a high degree of efficiency of processing;
- to achieve the logical independence for query and transaction formulation on this level;
- supporting bad schemes and good for the manipulation of schemes that violate normalization, nonredundancy, etc.;
- coping with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user;
- designing for consistency if the same type of information appears in several design phases.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process in such a formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

Database designers have to have a deep knowledge in database theory especially normalization theory, a good understanding of the application area and a good intuition on the implementational restrictions of the environment and on the programming problems. Doing this split is in most cases intractable for a single designer or even groups of designers. However, most of the database theory is well formalized. Therefore, this knowledge can be used for the automatic support and the simplification of the design task.

The design system $\text{(DB)}^2$ captures a lot of information about schemes under design. It has a data dictionary in which schema information can be expressed and transformed and is based on a database model that can support all phases of the design process. Nowadays, the design process is understood to capture the structural design and the modelling of the semantics, as well as the description of the behavior of the database, especially the operations defined for the database. We evolved the classical entity-relationship model to the higher-order entity-relationship model (HERM) which can support design in any of the main classical data models and higher order data models and also translation among them. It supports integrity constraints. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemas. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated.

At present the system $\text{(DB)}^2$ is widely used for database design. There is a large number of active
For the effective utilization of the system, the system should support different design methodologies. The proposed system supports effectively several methodologies. There are several design methodologies [3, 6, 9, 10, 20]. Top-down and bottom-up design methodologies with/without graphical representation are widely discussed in the literature. There are also some design methodologies which are developed for specific models. Most of them are general methods for the database design using the waterfall design approach known in software engineering. In AI and other branches of Computer Science other methodologies are developed. For knowledge acquisition, for example, interview techniques are applied. The design-by-units design methodology is a technique which could be understood as a step-by-step reification and generalization method. The designer develops first the design for the basic, independent units. Then these units are considered together for the design of more general units. This methodology is very general. Among the methodologies supported by the design system \((DB)^2\) this methodology was the most frequently used.

Doing the split and putting theory into practice means:

- To use a well-developed design methodology which is easy perceivable, transparent and supports abstraction.
- To be based on a sound theory which supports each of the design steps and supports also maintenance of the database.
- To build a whole environment for the database design, database use and database change during lifetime as well as for changing the processing environment.

The paper is divided into three parts. First we introduce the object-oriented entity-relationship model, the higher-order entity-relationship model. Then we show how this model is to be used in the design system \((DB)^2\). In the last part we illustrate the design-by-units design methodology using an example.

2 The Object-Oriented Entity-Relationship Model

2.1 The Object Model

Objects are to be handled and modelled in databases. They can own an object identifier and are to be characterized by values and references to other objects, i.e. the structural part of the object model could be characterized by

\[ o = (i, \{(s, v)\}, \{ref\}). \]

The value characterization is to be bound to an already defined structure \(s\). Characterized properties of objects are to be described by attributes which forms the structure of the object. Objects have furthermore a special semantics and a general semantics. Operators are to be associated to objects. These operators have a behavior. Object which have the same structure, the same general semantics and the same operators are be collected in classes. The structure, the semantics and the operations of a class is represented in types. Modelling of objects includes in this case the
association of objects to classes $C$ and their corresponding value type $T$ and reference type $R$. Therefore, objects are to be represented by $o = (i, \{(C, T, v)\}, \{(C, R, ref)\})$.

The known design methodologies vary in the scale of information to be modelled in the types. If objects in the classes can be distinguished by their values then the identifiers can be omitted and we use value-oriented modelling. In the other case, we use an object-oriented approach. In the object-oriented approach different approaches can be distinguished. If all objects are identifiable by their value types or by references to identifiable objects the database is called value-representable. In this case, the database can be modelled by the value-oriented approach too and a mapping from the value-representable scheme to a value-oriented scheme exists. If the database is not value-representable then we have to use object identifiers. It is well-known that in this case either the identifier handling should be made public or the databases cannot be updated and maintained. Therefore, value-representable databases are of special interest.

Normally, objects do not exist in a database independently. An object is to be called kernel objects (or independent) if its existence in the database is independent of the existence of any other object in the database. An object is called characteristic if it describes some other object. Objects can perform a superordinate role in interrelating other objects, in which case they are called associative. The exists associations among objects. Associations can be also objects.

Kernel objects are to be described by entities in the valued-oriented approach. All other object can be entities or relationships. Kernel objects can be distinguished by their values of some attributes. These attributes are called key. In value-representable databases objects are kernel objects if they are identifiable by their values. These objects are represented by entities. All other objects are to be represented by relationships.

The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [12]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types. The clear distinction of independent objects from dependent objects is another advantage of the HERM model and the system $(DB)^2$. This simplifies the modeling since we do not need additional constructs for subtyping.

2.2 The Higher-Order Entity-Relationship Model

We introduce now the higher-order entity-relationship model. Besides the existence of a strong theoretical basis there are several other advantages of the HERM approach:

HERM-schemes are much simpler and are easier understandable than ERM-schemes.

HERM-schemes support abstraction in a simple but comprehensible manner.

HERM-schemes can be translated together with the corresponding constraints, with the corresponding user-defined operations and with the generic operations to normalized relational, hierarchical or network schemes.

The HERM type consists of information on the structure, (static) semantics, operations and behavior (dynamic semantics), i.e.

$$\text{HERM-Type} = \text{Structure + Semantics + Operations + Behavior}.$$
This notation can be generalized to the more general which is out of scope for the purposes of this paper

HERM-Type = Structure + Semantics + Operations + Behavior + Environment.

The higher-order entity-relationship model has the following modelling constructs:

**Simple attributes** For a given set of domains there are defined attributes and their corresponding domains.

**Nested attributes** Using basic types complex attributes can be defined by means of the following constructors:

- Tuple constructor. Using nested attributes a new nested attribute is defined by the cartesian aggregation.
- Set constructor. Using a nested attribute a new nested attribute is defined by the set aggregation.

Additionally, the bag and the list constructors can be used. For the sake of simplicity we use here only the tuple and set constructors.

**Entities** Entity types are characterized by their attributes. Entity types have a set of attributes which can be used as the identifier of the type.

**Clusters** The union of types is called cluster.

**First-order relationships** First-order relationships types are defined to be associations between entity types or clusters of those. They can be additionally characterized by attributes.

**Higher-order relationships** The relationship type of the order \( i \) is defined as an association of relationship types of order less than \( i \) or entity types and can be additionally characterized by attributes.

**Integrity constraints** A corresponding logical operator can be defined for each type. A set of logical formulas using this operator can define the integrity constraints which are valid for each instance of the type.

**Operations** Each type can have a set of (conditional) operations and query forms. Furthermore, generic operations like projection, restrictions, insert, (tagged) clustering are predefined for each type.

Let us consider examples of these constructs in a medical database. This database covers information: on known diseases (symptoms, drugs which may aggravate, cause or cure the disease), known drugs with their components and patients, physicians and other persons.

A name of a person is defined by the cartesian aggregation

\[
\text{Name}(\text{First}, \text{Fam})
\]

The membership in societies can be defined by the set aggregation attribute
Membership{Member(Society,Since)}.
The address of persons is usually defined to a complex attribute, for example
Address(State, City(Code, Town), Street(Name, House(Numb, Appartm))).
A person can be characterized by its social security number, its name, its address and the sex and
has the set \{SSN\} as the identifier, i.e.

\[\text{Person} = (\{\text{SSN, Name(First, Fam)}, \text{Adr(Zip, Town, Street(Name, Nr))}, \text{Sex}\}, \{\text{SSN}\})\].

The relationship cure of patients can be modeled by
\[\text{Cure} = (\text{Patient, Physician, Drug, Disease, \{Date\}})\].
An example of an integrity constraint over Cure is the following predicate
\[\text{CorrectCure} := \text{Cure(Patient)} \neq \text{Cure(Physician)}\]
which indicates that nobody is curing himself. We can also use labels instead of the (Person,1).
The function
\[\text{FamilyName}(x) := (\text{Person}(x)[\text{Name}])[\text{Fam}]\]
is an example of an operation defined for the type Person.
The function \[\text{Doctor}(x) := (\text{Select}_{\text{Person}(\text{Name})=x}(\text{Cure}))[\text{Physician}(\text{Name})]\]
generating the physicians of a given person is an example of an operation which involves more
than one type. Such operations can be used to define roles of relationships.
Operations can be defined using preconditions and postconditions. If, for instance, the types
\[\text{Person} = (\{\text{ID, Name, Address}\})\] and its subtype
\[\text{Physician} = (\text{Person}, \{\text{LicenceNr, Salary, Dept}\})\]
are defined and any physician is a person which is stored in the database then the insertion
operation for the type Physician can be generically defined with one precondition or with a post-
condition based on the operation Insert:
\[\text{Add1(Physician,(x,y,z,u,v,w)) := if Person(x,y,z) then Insert(Physician,(Person(x,y,z),u,v,w));}\]
\[\text{Add2(Physician,(x,y,z,u,v,w)) := Insert(Physician,(Person(x,y,z),u,v,w))}\]
\[\text{if not Person(x,y,z) then Insert(Person,(x,y,z))}].\]
The HERM differs from the classical ERM and from extended ERM in several constructs.
1. Constructs which can be found in different extensions of the classical ERM are: nested
attributes, first-order relationships of higher arities, clusters and some integrity constraints
like the complexity.
2. The following constructs are new: higher-order relationships, integrity constraints including
the generalized complexity, operations and conditional operations.
3. Since weak entity types and Is-A-relationships can be represented directly in the HERM
there is no direct need for these constructs in the HERM.
4. Kernel objects are distinguished from other objects and differently represented by entity
types. All other objects are to be represented by relationship types. For this reason the
schemes in HERM are simpler several times than ERM schemes.

The entity-relationship (ER) model is one of the most popular database design models. Despite numerous positive features of the ER approach there still exists a strong need for a theoretic basis. This theory must be able to define sets of semantically well-formed ER schemes for particular user-specific ER-techniques as well as for subproblems as scheme design, view integration, query generation, and scheme transformation. Codd [5, p. 477] states even that the entity-relationship model ”is clearly the winner in terms of its lack of precise definitions, lack of clear level of abstraction, and lack of mental discipline”. In [18] the entity-relationship model is extended and it is shown that a precise, formal definition exists. This is based on the HERM-methodology which is using an abstract-data-type-approach.

3 HERM and the system \( (DB)^2 \)

The system \( (DB)^2 \) (Data Base Design by Beta; \( DBDB = (DB)^2 \)) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. \( (DB)^2 \) supports database design from requirements analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual design, thus offering a choice between the relational, network, or hierarchical models for the logical design. Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can be used for creating query compilers which are more productive, effective and forceful.

One of the most important advantages of the system is the interactive user-driven input of information. It is used an easily intelligible, comprehensible and understandable ”fill in the blanks” input procedure which is required in the literature as the ideal input form.

The tools developed in \( (DB)^2 \) can be divided into the following groups:

1. Analyzers (for reports, schema checking, the normalization, access evaluation tools).
2. Transformers (for new schemes which are information content equivalent).
3. Heuristic, user-dialog-driven tools (for solving view cooperation problems).
4. Translators appropriate for the target data model (relational, network or hierarchical) and system.
5. Query definition module (which is also used for behavioral normalization).

The general system architecture is represented in figure 1 where thicklines denote the data flow and thinlines the control flow. At present, the system \( (DB)^2 \) is extended to the design system RAD [2] incorporating different AI techniques for an efficient acquisition of the designers knowledge.
4 Methodologies and Design-By-Units

The system is based on the HERM methodology. In database modeling three different perspectives can be identified. Different models stress only some of those. The structure-oriented, semantic perspective focuses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The process-oriented perspective is concerned with the processes or activities performed in the application area. The behavior-oriented perspective is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The HERM methodology supports this approach and is an extension of [1, 4, 9, 10]. We start with meaningful examples (the first step), develop the structure and the semantics of the database under design (the next four steps), generate the operations and model the behavior of the database (the next two steps). Throughout the procedure, checks are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational design can require for performance reasons the redesign of the structure. The eight steps can be defined as follows:

1. transform examples in elementary entity and relationship types, analyze the structure;
2. graphical representation of types and their associations, population check;
3. eliminate surplus, derivable types, determine roles;
4. modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. normalize the types;
6. modeling of operators over the types and the behavior;
7. optimizing types and deriving views;
8. check the completeness, consistency of the design with the examples.

This methodology is directly supported by the design system $(DB)^2$.

The HERM and $(DB)^2$ methodologies generalizes different approaches presented in the literature (see for instance [6, 10, 15]). The most important classes of integrity constraints used in the database modeling system [16] are the functional dependencies, the generalized functional dependencies, the multivalued dependencies, the inclusion dependencies, the exclusion dependencies. In $(DB)^2$ there are implemented three different choices for the translation of Is-A- and generally (0,1)-relationships. The design system $(DB)^2$ is able to change from one representation to the other.

The system can be used for the support over the complete lifecycle of database systems. Most known methodologies are not well adapted to the lifecycle because the design information is not used after the design process. Design design is a far more iterative process as captured in the straightforward lifecycle model of these methodologies. Using $(DB)^2$ during the whole system development the complete design information is usable for restructuring the database. This makes it possible to restructure the database and during restructuring to recompile in accordance to the old and new structures the programs which are specified in the HERM algebra.

The behavior can be specified using generic operations like insert and deriving restrictions to the behavior from the semantical information. The maintenance complexity is one of the most important behavioral database design criterias. The database designer can design the structural, semantical, operational and behavioral information on the application area and the system generates in accordance to the DBMS’s type and the distribution the corresponding data structures and transactions.

Database design is at present considered as top-down design. This approach is only useful for single-designer support systems. However, in the case the designer can detect only in a restricted manner similarities between the concepts. This is especially useful if the design is made by several designers. In this case the view integration is one of the most difficult problems which has to be solved for an efficient management of the whole database system. This task can not be solved using only methods of top-down design. The window manager approach in $(DB)^2$ can be used to detect similarities. Moreover, designing the process information at the same time adds the possibility to adapt the structure to the process efficiency. For this reason, new normalization algorithms are used in $(DB)^2$ which prohibit normalization if this contradicts the process efficiency.

In [21, 19, 18] another new design methodology was developed: Design-by-units. Most of the well-known design methodologies think as in the relational approach. But each of the database models should have its own methodology. It is surprising that most of the proposed models do not have its own design methodology. If the model is getting richer in construct the methodology should be deepen. One of the database models with its own methodology is the ER model. However, there is still a little agreement in which cases objects from the real world should be modelled by attributes, entities or relationships. A part of the problems of view integration is caused by this modelling problem. And this contradicts the belief of experienced database designers. Those
assume that the views of an enterprise can be integrated since there is an internal integration in
the enterprise. The reason for this mismatch is that methodologies are not supporting abstrac-
tion in an efficient manner. The new design methodology can be understood as a step towards a
well-integrated methodology.

The proposed methodology is truly object-oriented and at the same time also theoretically based
what supports the implementability. This methodology support also extensibility since using this
methodology an existing design and implementation can be extended without introducing changes
to it. It promotes reuse and inheritance as well as behavioral extension. To some extend, this
approach is similar to modular design known in software engineering. The orientation is different
only. We are first interested in the data representation part and then in the processing part since
a part of the processing part is based on generic operations which are defined according to the
structure.

This approach has further other advantages: it is easier to detect similarities among design units
and to reuse parts of design units in other units; changes to the scheme and to parts of units are
directly reflected in all other units which are using the changed. The new methodology supports
directly the above discussed distinction between kernel and dependent object. This is especially
useful, if abbreviation techniques [11] are used in query forms [21]. It is a recursive method. Each
step is based on the above discussed eight design steps. This methodology is not following the
classical waterfall model with iterations but rather supporting a high level inside-out-strategy [3].
Experience in utilization of \(DB^2\) has shown that this methodology was the most often choosen
for practical design.

**Design-by-units**

1. **basic step.**

   *Design the types for the independent kernel object classes.*

2. **Recursion step.**

   *Repeat until the schema is not changed.*

   *Either reification :*

   *Refine the units introducing subtypes (in HERM represented by unary relationship
types).*

   *Refine the relationship types according to the subtypes.*

   *or generalization of units:*

   *If there are associations among units then introduce a new unit containing the associated
units and the relationship type according to the association.*

   *Add other relationship types if there exist new associations.*

Let us illustrate this method using the above mentioned medical example. First we design the
basic units and then we relate these basic units like the units **Person** and **Disease** in figures 2
and 3. Then we merge these units obtaining larger, more general units. One relationship type
which is to be designed during the merge of the units **Person**, **Disease** and **Drug** is the type
**Cure**. In the last iteration the schema in figure 4 is designed.

The final result of the design by units is represented in figure 4.
Figure 2: The unit PERSON

Figure 3: The unit DISEASE
Figure 4: HERM-Diagram of the Medical Application Database
5 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system \((DB)^2\) was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and system have the following advantages:

1. **The model is easy understandable, simple and perceivable.**
   - The model can be used as a basis of database design tools [16, 19]. The system \((DB)^2\) is used at present by more than 95 user groups.
   - The model supports high-level user interfaces, extensibility of databases and has a set of special methodologies for its utilization.
   - The modeling approach is simple to use even for large problems.
   - Since the model uses graphical representations the modeling result is easier to understand and visible.
   - In an experiment, 20 novice or end-user database designers learned the HERM methodology and later designed different database schemes in different areas. Our experience was that that the methodology was easily accepted, led to more accurate, to less redundant schemes and to a correct modeling of complex relationships.
   - Using query forms [21] the user can specify application transactions and programs on the top of the HERM which reduces substantially the complexity of application programming.

2. **The results of the design are much simpler than in other approaches.**
   - We have used the the model for modeling also some more complex applications. One observation is that the obtained schemes are from three to five times simpler than those obtained by other models.
   - The example of [15] is simplified by four times and can be placed on one page or one screen. In other examples, the simplification makes it possible to find a model. Using this modeling approach, an airport counter application was modelled by less than 40 entity types and less than 120 relationship types whereas the original solution with more than 150 entity types and more than 400 relationship types was unacceptable by users because of complexity and non-transparency.
   - The simplification leads also to a better understanding of the application and makes normalization easier to perceive.
   - The schemes avoid additional redundancy. Using HERM, the normalization and the minimalization of schemes can be considered together.

3. **The utilization of \((DB)^2\) is more natural and simpler.** Only necessary facts are to be expressed.
   - The model supports a direct translation to the three classical database models. This translation preserves normal forms. Since a direct translation to relational, network and hierarchical schemes can be used the design decisions directly could be used to obtain schemes in normal forms. The translation theory can be used for a multimodel- and multisystem-support [21] and presents a practical solution to interoperability of systems.
The HERM algebra is used for query definition. The corresponding relational, network or hierarchical queries can be automatically generated.

The model supports a rich set of constraints. These constraints are used for the development of the scheme equivalence. Although the excessive number of fact-encoding mechanisms means that the same semantic unit can be declared in many syntactically different and compatible ways, the information described is equivalent. This equivalence theory can be used for automatic modification of schemes [2].

The database maintenance procedures can be derived using the design information.

Using a knowledge base previous and system-provided design decisions can be reused or partially reused what simplifies the design task. Furthermore, similarities in the design can be detected and used for simplification of the implementation.

Using the whole design information the retranslation of application programs can be used for the adaption of existing database systems to changing environments.

4. The model and the design system have a strong theoretical basis.

The model is based on a multitype logic which is equivalent to the first-order predicate logic. For this reason, results known from discrete mathematics and relational theory [17] can be used.

The model covers the complete modeling information. The structure, static semantics, generic and user-specified operations and behavior of an application can be described by the model.

The theory is simplified and cleaned up. Sequences, subsets and powersets of objects can be modeled directly. Is-A-Relationships are treated in a better way. Weak entity types can be avoided completely. A normal form theory is developed for the HERM. Using this normal form theory, we can obtain normalized schemes like in the classical theory.

5. The theory is applicable to practical needs.

Based on the theory a multi-paradigm, robust design methodology is developed [19, 21] which incorporates approaches known in object-oriented modeling [13], modular programming [16] and programming in large.

Using this modeling approach, a view cooperation concepts was developed. Since full view integration is not decidable and not axiomatizable view cooperation is the only applicable approach.

The approach can be used for reverse engineering. Systems and programs developed for one management system can be recompiled and adapted to other management systems.

The theory, the model and the developed design systems can be used for managing a heterogeneous mix of databases, may be even under the illusion of a single common data model. This supports directly the migration from one system to another system.

References


Intelligent Database Design
using an
Extended Entity-Relationship Model

Bernhard Thalheim
Computer Science Department, Rostock University
Albert-Einstein-Str. 21, Post Box 999, D-O-2500 Rostock
thalheim @ informatik.uni-rostock.dbp.de

Preprint - CS - 11 - 91
Dezember 1991,
Final revision: May 92

Abstract

Database design methodologies and tools should facilitate database modeling, effectively support database processing, database redesign and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970’s, various tools for database design have been introduced. Most of them, however, are dependent on the knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The proposed systems (DB)² and RAD do not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The Entity-Relationship Model is extended to the Higher-order Entity-Relationship Model (HERM) by adding structural constructs and using integrity constraints and operations. Different database design methodologies are developed based on the HERM approach.

1 Database Design

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

• to meet all the information (content) requirements of the entire spectrum of users in the given application area;
• to provide a "natural" and easy-to-understand structuring of the information content;
• to conserve the whole semantic information of the designers for a later redesign;
• to achieve all the processing requirements and achieve a high degree of efficiency of processing;
• to achieve the logical independence for query and transaction formulation on this level.
While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process in such a formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

A schema is called high quality or conceptually adequate if:

1. It describes the concepts of its application naturally (as they are in the real world; the user can translate ideas easily in both directions between concepts of the schema and the natural concepts of the applications world). An important factor is the clarity of representation. It is important that the user is able to understand the representation, and this puts a premium of clarity and comprehensibility of the representation. This is also an important consideration when debugging or tuning the database. Linked to this feature is the conciseness of the representation. The more concise a representation the more likely it is to be readily understood. Conciseness can have also implications for computational efficiency.

2. The schema contains no or very little or necessary redundancy (redundancy is the possibility of representing a fact of the applications world more than once in the same instantaneous database). Storage efficiency, preventing inconsistency of the database and its update anomalies require avoiding redundancy. Each update should be made on all facts otherwise the facts would contradict (may cause unpredictable behavior of application programs). When redundancy is needed for the convenience of the users, it should be introduced into the user views but not into the schema. In some database models we cannot eliminate the redundancy completely. When we need a certain redundancy, we should at least bind it by integrity constraints. When such constraints are implemented, the user is forced to update all the related facts simultaneously. Redundancy is also useful for an ease of querying.

3. The schema does not impose implementational restrictions, that is, every situation probable in the real world of the application is fully representable under the schema.

4. The schema covers as many integrity constraints as possible. Then the class of instantaneous databases formally possible according to the schema is not much larger than the class of all possible situations of the real world.
   But: - they are hard to specify and to formulate;
   - they are seldom enforced by the DBMS and require additional programming;
   - they are often incorrect implemented and usually prevent direct interaction;
   - users and application programmers often forget or misunderstand such constraints.

5. The scheme is flexible (probable changes in the application world can be modeled without drastic changes in the schema).

6. The concept is conceptually-minimal: it does not involve concepts which are irrelevant and limits the accumulation of information which is irrelevant in that world.

The most important issue of the database is the design of a high-quality schema within the restriction of the model and the class of DBMS or more general within the restrictions of the modeling paradigms. Low-quality schemes are hard to use, to maintain, and to adjust and tend to corruption. Lastly, but by no means discountable, is the purely pragmatic consideration of who is to use the conceptual model. Personal preferences and past experience can make on representation better than
another, given an application and a team of people to build it. Therefore, database design tools need to be adaptable to a wide range of designers.

Database design could be viewed as a special knowledge representation process. It should make processing easy and should map onto problems that we know how to deal with. It must not only be convenient but also adequate to the task it is asked to perform. We can distinguish three different types of adequacy:

- **Metaphysical adequacy** obtains if there are no contradictions between the objects we wish to represent and our representation of them.

- **Epistemological adequacy** is about the ability to express the real world. To be epistemologically adequate the representation must be capable of being used to express the objects we know about the real world.

- **Heuristic adequacy** obtains if we can express in our representation the database processing.

The reusability of design decisions is another important feature of modern design tools. Reusable development information includes not only generic code components but also generic system specification like generic schema components and generic requirements. The availability of reusable development information is a key factor for design reusability. Requirement and design reuse have greater return on investment than simple reuse of code. Reused designs can guide the retrieval and reuse of code components best-suited to the current application. Object-oriented design techniques provide a basic support to the construction of generic specifications. Modular design and abstraction promote the design of generic classes (later called units) through the specification of the external behavior of units while hiding the internal details. Our modular design methodology addresses the reuse problem by using a high-level abstract object-oriented approach and viewing the database design process as two complementary activities:

- Construction of generic development information to be reused in the development of similar applications in a particular domain.

- Construction of applications with specific requirements by reusing the available generic development information for the application domain.

A database changes also its structure over its lifecycle. In some cases such changes are only extensions of the structure. For instance, in relational database system relations are to be added. If the new components are independent from the previous then this does not cause any problem. However, this independence is not the normal case. Even adding structures means in most cases also to add certain integrity constraints, e.g. referential integrity constraints. This lead to a more complex management and then later to database tuning and redesign by restructuring. In other cases, the database is directly to be restructured. A database design tool should actively support this restructuring. This can be based on a theory of equivalent schemata or subschemata.

Redesign is closely related to reverse engineering. The aim of reverse engineering is to answer the question: what could be a possible specification of this implementation. The problem is particularly complex for old and ill-designed applications which are to shifted to other technological platforms. The success of reverse engineering depends whether there is a documentation, whetehr there was used a systematic methodology and which implementational restrictions led to tricky design decisions. Since data are a stable part in an information system the task of reverse engineering is simpler than in software engineering.

The published and known literature underlies many parts of the database design process. In particular, the following theories can be used for the database design \[\text{EMN89}, \text{Heu89}, \text{Hul89}, \text{NiH89}, \text{Ris88}, \text{Tha91}\]:

- theory of data models;
• theory of normalization of relational databases;
• theory of scheme equivalence.

But the published work needs also several extensions and adaptations in several respects [Tha91], [Tha92]:

• supporting bad schemes and good for the manipulation of schemes that violate normalization, nonredundancy, etc.;
• coping with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user;
• designing for consistency if the same type of information appears in several design phases.

In database modeling three different perspectives can be identified. Different models stress only some of those. The structure-oriented, semantic perspective focusses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The process-oriented perspective is concerned with the processes or activities performed in the application area. The behavior-oriented perspective is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The following strategy supports this approach and is an extension of [AbH84],[Che83],[NiH89],[Ris88]. We start with meaningful examples (the first step), develop the structure and the semantics of the database under design (the next four steps), generate the operations and model the behavior of the database (the next two steps). Throughout the procedure, checks are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational design can require for performance reasons the redesign of the structure. The eight steps can be defined as follows:

1. transform examples in elementary entity and relationship types, analyze the structure;
2. graphical representation of types and their associations, population check;
3. eliminate surplus, derivable types, determine roles;
4. modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. normalize the types;
6. modeling of operators over the types and the behavior;
7. optimizing types and deriving views;
8. check the completeness, consistency of the design with the examples.

This strategy is directly supported by the design systems (DB)$^2$ and RAD which are to be described in section 4. The strategy is one refinement of the general design methodology described in section 3.

The design systems (DB)$^2$ and RAD capture a lot of information about schemes under design. They have a data dictionary in which schema information can be expressed and transformed and is based on a database model that can support all phases of the design process. They support further synergistic database design because it relies on the systems to discover and suggest appropriate transformations leading to a theoretically optimal design and relies on the designer to supply additional information sought by the system in an effort to have a complete set of useful information and to make exceptions to rules whenever real-world demands are in conflict with theoretical ideals. Nowadays, the design process is understood to capture the structural design and the modelling of the semantics, as well as the description of the behavior of the database, especially the operations defined for the database. We
evolved the classical entity-relationship model to the higher-order entity-relationship model (HERM) which can support design in any of the main classical data models and higher order data models and also translation among them. It supports integrity constraints. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated.

Database design is based on one or more data models. A large number of conceptual data models have been proposed. However, actual experience with the use of these models as a basis for implementing a generalized DBMS is very scant. While most models have been proposed primarily for stand-alone database management systems and are adapted to implementational restrictions in database systems, it is increasingly apparent that data models will be from one hand side incorporated directly into programming languages and a variety of tools (e.g. CAD/CAM, expert systems, knowledge bases) and from the other hand side have to be extended to interoperating environments and multisystem- and multimodel-paradigms. Nearly all early commercial DBMS implementations were based on the hierarchical model such as IMS and SYSTEM-2000 or the network model such as IDS and IDMS or the relational model such as INGRES, DB2. The relational data model was proposed as a simple and theoretically well-founded representation of data, and it has soon become the most important model for database systems (see for example [PBG89],[Ull89]). The primary virtues of the model are its rigorous mathematical foundation and the correspondence of a relation with the notion of a table. However, research efforts have highlighted a large number of drawbacks to the relational model. Rather than abandon the relational paradigm because of these disadvantages, we are interested in extending relational languages in a way that incorporates useful ideas from alternative language paradigms but allows the retention of most, if not all, of the advantages of the relational approach.

The entity-relationship (ER) model is one of the most popular database design models. Despite numerous positive features of the ER approach there still exists a strong need for a theoretic basis. This theory must be able to define sets of semantically well-formed ER schemes for particular user-specific ER-techniques as well as for subproblems as scheme design, view integration, query generation, and scheme transformation. Additionally, the formalism has to be suited for computer aided software engineering tools. In [YaT89],[Tha89] the suitability of the HERM approach for the solution of database design problems is shown. One reason for the huge variety of extensions of the ER model is that a well-founded theory is still under development. Codd [Cod90, p. 477] states even that the entity-relationship model "is clearly the winner in terms of its lack of precise definitions, lack of clear level of abstraction, and lack of mental discipline". In [Tha92"] the entity-relationship model is extended and it is shown that a precise, formal definition exists. This is based on the HERM-methodology which is using an abstract-data-type-approach.

The paper is divided into three parts. First we introduce briefly the object-oriented entity-relationship model, the higher-order entity-relationship model. In section 3, the database design process is to be discussed. Different classical design strategies are to be reviewed. Further, we introduce the design by units design process which is a unifies modular and object-oriented database design. In section 4, we show how the extended ER-model and the underlying methodology is to be used in the design system (DB)\textsuperscript{2}. Although this system is widely used it has several restrictions which we try to overcome in another design tools RAD. This tools was and is to be developed in a joint project of the universities Rostock, Aachen (now Gießen) and Dresden.
2 The Object-Oriented Entity-Relationship Model

2.1 Object-Oriented Modeling - The Structural Part

Objects are to be handled and modeled in databases. They can own an object identifier and are to be characterized by values and references to other objects, i.e.

\[ o = (i, \{(s, v)\}, \{ref\}) \]

The value characterization is to be bound to an already defined structure \( s \). Characterized properties of objects are to be described by attributes which forms the structure of the object. Objects have furthermore a special semantics and a general semantics. Operators are to be associated to objects. These operators have a behavior. Object which have the same structure, the same general semantics and the same operators are be collected in classes. The structure, the semantics and the operations of a class is represented in types. Modelling of objects includes in this case the association of objects to classes \( C \) and their corresponding value type \( T \) and reference type \( R \). Therefore, objects are to be represented by

\[ o = (i, \{(C, T, v)\}, \{(C, R, ref)\}) \].

The known design methodologies vary in the scale of information to be modelled in the types. If objects in the classes can be distinguished by their values then the identifiers can be omitted and we use value-oriented modeling. In the other case, we use an object-oriented approach. In the object-oriented approach different approaches can be distinguished. If all objects are identifiable by their value types or by references to identifiable objects the database is called value-representable. In this case, the database can be modelled by the value-oriented approach too and a mapping from the value-representable scheme to a value-oriented scheme exists. If the database is not value-representable then we have to use object identifiers. It is well-known that in this case either the identifier handling should be made public or the databases can not be updated and maintained. Therefore, value-representable databases are of special interest.

Normally, objects do not exist in a database independently. An object is to be called kernel objects (or independent) if its existence in the database is independent of the existence of any other object in the database. An object is called characteristic if it describes some other object. Objects can perform a superordinate role in interrelating other objects, in which case they are called associative. The exists associations among objects. Associations can be also objects.

Kernel objects are to be described by entities in the valued-oriented approach. All other object can be entities or relationships. Kernel objects can be distinguished by their values of some attributes. These attributes are called key. In value-representable databases objects are kernel objects if they are identifiable by their values. These objects are represented by entities. All other objects are to be represented by relationships.

The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [STH91]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types.

The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [STH91]. There are several approaches that encapsulate object-oriented features [NaP88]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types. This approach is not ambiguous and enforces therefore a better understanding of the object classes. Since subclasses consist of dependent objects subclasses are modeled
by unary relationship types. This is one of the first differences in the HERM modeling methodology [Tha88].

2.2 HERM - The Structural Part

We introduce now the higher-order entity-relationship model. Besides the existence of a strong theoretical basis there are several other advantages of the HERM approach:

- HERM-schemes are much simpler and are easier understandable than ERM-schemes.
- HERM-schemes support abstraction in a simple but comprehensible manner.
- HERM-schemes can be translated together with the corresponding constraints, with the corresponding user-defined operations and with the generic operations to normalized relational, hierarchical or network schemes.

The HERM type consists of information on the structure, (static) semantics, operations and behavior (dynamic semantics), i.e.

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior}. 
\]

This notation can be generalized to the more general which is out of scope for the purposes of this chapter

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior} + \text{Environment}. 
\]

The higher-order entity-relationship model has the following modelling constructs:

**Simple attributes** For a given set of domains there are defined attributes and their corresponding domains.

**Nested attributes** Using basic types complex attributes can be defined by means of the following constructors:

- Tuple constructor. Using nested attributes a new nested attribute is defined by the cartesian aggregation.
- Set constructor. Using a nested attribute a new nested attribute is defined by the set aggregation.

Additionally, the bag and the list constructors can be used. For the sake of simplicity we use here only the tuple and set constructors.

**Entities** Entity types are characterized by their attributes. Entity types have a set of attributes which can be used as the identifier of the type.

**Clusters** The union of types is called cluster.

**First-order relationships** First-order relationships types are defined to be associations between entity types or clusters of those. They can be additionally characterized by attributes.

**Higher-order relationships** The relationship type of the order \(i\) is defined as an association of relationship types of order less than \(i\) or entity types and can be additionally characterized by attributes.

**Integrity constraints** A corresponding logical operator can be defined for each type. A set of logical formulas using this operator can define the integrity constraints which are valid for each instance of the type.
Examples for the constructs are the following:

A name of a person is defined by the cartesian aggregation
\[ \text{Name}(\text{First, Fam}) \].

The membership in societies can be defined by the set aggregation attribute
\[ \text{Membership}\{\text{Member(Society,Since)}\} \].

The address of persons is usually defined to a complex attribute, for example
\[ \text{Address}(\text{State, City(Code, Town), Street(Name, House(Numb, Appartm))}) \].

A person can be characterized by its social security number, its name, its address and the sex and has the set \{ SSN \} as the identifier, i.e.
\[ \text{Person} = (\{ \text{SSN, Name(First, Fam), Adr(Zip, Town, Street(Name, Nr))}, \text{Sex} \}, \{\text{SSN}\}) \].

The relationship marriage of persons can be modelled by
\[ \text{Marriage} = (\text{Person}, \text{Person}, \{\text{From, To}\}) \].

An example of an integrity constraint over marriage is the following predicate
\[ \text{CorrectMarr} := \text{Marriage}(\text{Person}(\text{Sex}),1) \neq \text{Marriage}(\text{Person}(\text{Sex}),2) \]
which indicates that the sex of the first person on Marriage must be different from the sex of the second person in Marriage. We can also use labels instead of the (Person,1).

The HERM differs from the classical ERM and from extended ERM in several constructs.

1. Constructs which can be found in different extensions of the classical ERM are: nested attributes, first-order relationships of higher arities, clusters and some integrity constraints like the complexity.

2. The following constructs are new: higher-order relationships, integrity constraints including the generalized complexity, operations and conditional operations.

3. Since weak entity types and Is-A-relationships can be represented directly in the HERM there is no direct need for these constructs in the HERM.

4. Kernel objects are distinguished from other objects and differently represented by entity types. All other objects are to be represented by relationship types. For this reason the schemes in HERM are simpler several times than ERM schemes.

The model is well-founded [Tha92']. Problems known in nested relational models are omitted. Since the aggregate operations are based on the kind of equality concept used and since the key concept is based on the equality concept the user can specify different equality concepts. Based on this decision the operations can be specified. Queries can be specified on this operational environment. Therefore, the HERM is successing in solving the limitations of the flat first-order normal form relational model and limitations of the classical entity-relationship model.

A rich harvest of theoretical results [Tha91] have resulted from the study of relational models and should be used in other models. Research effort have highlighted drawbacks to the relational model and to the relational design methodology.

1. The uniformity of relations (i.e. a relation contains a set of tuples of a particular format) is too strong a restriction for such types of data as design data, image data, or audio data.

2. Certain restrictions on the set of admissible relations acceptable in database processing applications are not reasonable assumptions for nontraditional applications.

3. The mismatch between input of the design (attributes and dependencies) and the aim is one reason that the relational paradigm is not general enough.
4. Normalization algorithms can not cover the real normalization process. They are ordering dependent, have a too high time complexity. The structure of multivalued and functional dependencies is too general. There are no tools which are able to cover the whole design information.

5. The aggregation functions are only defined based on atomar relations and are not uniquely definable for views.

The classical Relational Model deals only with flat relations. It is not aware of any distinction between entity relations and relationship relations. In contrast, models like the network model, the hierarchical model and the entity-relationship model make distinctions between these two types of relations.

A conceptual scheme is a global description of the database that hides the details of physical storage structures and concentrates on describing entities, data types, relationships, and constraints. A high-level datamodel can be used at this level. Recent research in database design methods has developed the idea of using two distinct data models at the conceptual level. An enhanced conceptual model would provide an effective means of describing the database application environment. A representational data model would be used for efficient translation of a scheme into physical data structures. For instance, the relational data model could be employed as a representational data model. One of the most accepted enhanced conceptual models has been the entity-relationship model (ERM) of Chen [Che76, Che83]. It has been recognized as an excellent tool for high level database design because of its many convenient facilities for the conceptual modeling of reality. Its basic version deals with more static structural properties, such as entities, attributes and relationships. The shortcomings known for the relational model can be listed also for the ERM. One reason is that this model is unable to represent hierarchical and higher-order relationships. Other problems known for the entity-relationship model are the following:

- Only first-order relationships can be modeled. But the relational design leads already to higher-order relationships.
- Is-A-relationships cannot be modeled naturally.
- The concept of weak entities is not theoretically based.
- The classical entity-relationship model does not use n-ary relationships.
- The solution [Teo89] to define new entities as clusters of entities and relationships leads to a loss of information.
- Sets, sequences, null valued relationships can not be be simply represented.

We show in [Tha91, Tha92, Tha92'] that these limitations can be solved in the extended entity-relationship model. Further, in [Tha92'] and [YaT89] a normalization theory was developed. The main criterion of normalization is maintenance simplicity which consists of two parts: storage and access simplicity and operational simplicity. Since the two parts of maintenance simplicity are conflicting the user should specify his/her preferences. On the basis of integrity constraints several normal forms could be choosen. For instance, if there are specified only functional dependencies then the elementary key normal form (for nested relations) is the goal of normalization. Relationship types inherit the keys of the underlying component types. The decomposition of keys of these types leads to an increase of the arity of the relationship type. Therefore, a decomposition can be rejected for performance reasons. Integrity constraints can be used for the decision whether a decomposition is rejected or not.

2.3 Operations and Behavior

The generic definition of manipulation operations is one of the benefits of the relational model. This benefit is inherited by the HERM. The operations Insert, Delete and Update are defined if a rational
tree exists. If the definition of these operations leads to infinite cycles the operation is not defined for
the corresponding type.

There have been several attempts to define an algebra of operations in the ER-model. [CIC88]
introduces selection, projection, union, cartesian product and difference. [PRY90] uses additionally
relationship join, product, reduction, compression, renaming, simplify and the value selection operators
$\sigma_{A=c}$ and $\sigma_{A\neq c}$. These operations are too powerful for generation of generic queries. A similar system
is used in [CEC85]. As far as we know [AtC81] are the first who perceive that there is a need for an
ER algebra to define the semantics of query languages precisely. Many known ER algebras lack of
formal semantics. The languages based on data structures similar to networks [Sub87] and based on a
calculus [GoH88, HoG88] are an exception. The HERM uses a set semantics for the definition of entity
instances and relationship instances. This does not limit the power since for each structure $S_x$ which
is based on a pointer semantics another structure can be constructed which is equivalent to $S_x$ and
which is based on a set semantics. The equivalence is defined on the basis of bijective mappings from
one instance to the other.

The HERM algebra contains the following operations.

**Union:** The union operation creates a union of (non-homogeneous) sets.

**Difference:** The difference operation between two sets of the same type is the usual set difference

**Projection:** The projection operation creates a set of objects removing components in the set being
considered and eliminating duplicates.

**Product:** The cartesian product creates a new set from given sets by the usual cartesian product.

**Nest:** The nest operation compare component values using a certain equality function and collects
values for the nesting component into sets.

**Unnest:** The unnest operation is used to restructure sets of sets eliminating a level of brackets and
then eliminating duplicates.

**Selection:** The select operation creates a set of database objects which satisfy a simple selection
predicate and is built by $\sigma_{A=B}$, $\sigma_{A\neq B}$.

**Rename:** A given set (type) with a name is being renamed.

**Clustering:** The clustering operation creates a union of sets and provides a name for this union.

**Entity Introduction:** An entity set is to be generated by the cartesian product of the component
sets and by values from the attribute set according to a constraint.

**Relationship Introduction:** A relationship set is to be generated by the cartesian product of the
component sets and by values from the attribute set according to a constraint.

**Relationship Deletion:** A relationship set is to be deleted. This operation is partial. It can be only
applied if the result is a schema which types uses only types of the schema.

**Entity Deletion:** An entity set is to be deleted. This operation is partial. It can be only applied if
the result is a schema which types uses only types of the schema.

The HERM-algebra is complete. An algebra is HERM-complete iff the HERM-algebra can be
expressed.

Now this algebra can be used as a basis for a DBPL-like extension [ScM91]. Using quantifiers $\forall, \exists$
and general selection operations $\sigma_{A \theta c}$ for $\theta \in \{<, >, =, \neq, \leq, \geq\}$ general algebraic expression can be
introduced. We use here a logical extension.
A **quantified expression** yields a boolean result and may be nested, e.g.
\[ \forall x \in \text{Student} \exists y \in \text{Lecture}(x.\text{Attended}, y.\text{LNr} = y.\text{LNr}) \, . \]
Quantified expressions can be used also for membership tests, e.g.
\[ \exists x \in \text{Student}.x.\text{Name} = \text{`Lehmann'}. \]

**Selective access expressions** are rules that denote subsets of sets, i.e.
\[ \{ t \in r \mid \alpha(t) = \text{`true'} \} \] where \( \alpha \) is a quantified expression.

**Constructive access expressions** are rules for the construction of sets which marked by a name and based on the values of other sets, i.e. algebraic expressions using selective access expressions.

The general algebraic expressions are used for the introduction of opaque type declarations like selectors and constructors.

The function
\[ \text{FamilyName}(x) := (\text{Person}(x)[\text{Name}])[\text{Fam}] \]
is an example of an operation defined for the type \text{Person}.

The function \( \text{Spouse}(x) := \text{Marriage}((\text{Person}((\text{Name})=x,1))[(\text{Person}((\text{Name}),2)] + \text{Marriage}((\text{Person}((\text{Name})=x,2))[(\text{Person}((\text{Name}),1)] \)
is an example of an operation which involves more than one type. Such operations can be used to define roles of relationships.

Operations can be defined using preconditions and postconditions. If, for instance, a type
\[ \text{Student} = (\{ \text{SSN}, \text{Name}(\text{First}, \text{Fam}), \text{Adr}(\text{Zip}, \text{Town}, \text{Street}(\text{Name}, \text{Nr})), \text{Sex}, \text{StudentNumber} \}, \{ \text{StudentNumber} \}) \]
is defined and any student is a person which is stored in the database then the insertion operation for the type \text{Student} can be defined with one precondition or with a postcondition based on the operation \text{Insert}:
\[
\text{Add1}(\text{Student}, (x,y,z,u,v)) := \text{if Person}(x,y,z,u) \text{ then Insert}(\text{Student}, (x,y,z,u,v));
\]
\[
\text{Add2}(\text{Student}, (x,y,z,u,v)) := \text{Insert}(\text{Student}, (x,y,z,u,v)) \]
\[
\text{if not Person}(x,y,z,u) \text{ then Insert(Person, (x,y,z,u))}. \]

Since a database schema can be considered as a small database itself it is natural to model schema evolution as database evolution. Therefore, we use meta-operations for the schema evolution. This approach is not implemented in \((DB)\) but at present used in \(RAD\) [BOT90].

The behavior modeling is an inherent part of HERM modeling. The (dynamic) behavior can be characterized by sequences of database (states). It can be expressed by temporal logical formulas which are interpreted in such state sequences [Lip89]. Several classes of formulas are simpler noted by special integrity constraints. Behavior is specified according to the following layers:

The **interface layer** specifies the visibility of parts of databases and restricts the updateability of the database.

The **operational layer** specify the behavior in terms of operations which are allowed to modify the database contents.

The **dynamic constraint layer** specifies the admissible database sequences.

An important task for a database design tool is the automatic translation of constraints from the dynamic constraint layer to pre- and postconditions in the operational layer. Operations and dynamic integrity constraints might violate dynamic integrity constraints. If such a violation occurs, the corresponding transaction or operation should be rejected afterwards. For this purpose, an integrity browser is to be implemented in RAD [BOT90]. The browser is able to detect any violation of a constraint, no matter whether static or dynamic. There are several classes of dynamic integrity constraints which are directly supported on the basis of high-level constructs: transition constraints (e.g. the salary does not decrease), insert-delete-restrictions (e.g. the insert operation is not allowed for a class), future-related classical constraints (e.g. if a certain value exists then another value appears sometimes in the future).
The user can specify the type of integrity constraint enforcement. Integrity constraints can be enforced directly (without delay) or with a certain delay (deferred ... until ...). In the last case the operations are to be modeled by transactions which commit after checking of delayed integrity constraints. The default mode is immediate.

Further, for operations and integrity constraints the exception actions can be specified. Operations can be specified for a certain subscheme. Therefore, we obtain the following general framework for the definition of operations:

\[
\begin{align*}
\text{Operation } \phi & \quad \text{[Localization: < Unit_Name >]} \\
& \quad \text{[Precondition: < Aktivation, Condition >]} \\
& \quad \text{[Postcondition: < Commit, Condition >]} \\
& \quad \text{[EnforcedOperation: < Operation, Condition >]}
\end{align*}
\]

### 3 The Database Design Process

The database design process can be described according to the information units which are necessary:

- The **requirement description** is an informal description of the application and the goals (mission) to be achieved.
- The **interface** is the description of the organizational structure and the distribution of the database.
- The **structure** of the database objects is described by different classes of objects and their typing systems.
- The **semantics** describes the static properties of objects and object classes. Thus, the semantics is used for restricting the set of possible databases.
- The **views** describes the different subschemata for different parts of the application.
- The **operations** describe how the database should operate.
- Based on the semantical, structural and view information cooperating views can be defined for the different sub-databases.
- Using abstraction **query forms** describe in general the database information retrieval processes for the user.
- The **behavior** specifies the dynamic constraints of a database.
- The **transactions** are used for the specification of database programs.
- Finally, the **database modul** collects and re-organizes the information.

These units are not independent. The description of views depends from the structural, semantical and interface information. The complete dependency graph is displayed in Figure 1. Based on these information units different design decisions can be generated. For instance, using the platform information, the information on organizational structures and the information on data structures feasibility studies can determine the cost effectiveness of various alternatives in the design of the information system and the priorities among various system components.

The database design activities can be differently organized by different methodologies in accordance to the information systems life cycle. For instance, we can use the commonsense knowledge on platforms first for feasibility studies. Then requirements can be collected, analysed, structured and grouped.
Based on this information the design is performed. Further, a prototype is developed. The prototype allows the user to verify that the information system satisfies their needs. This can lead to an additional requirement collection and analysis. If the prototype satisfies the requirements the implementation of the design is performed. Implementation is concerned with the programming of the final, operating version. During the validation and testing process the developer assures that phase of the process is of acceptable quality and is an accurate transformation from the previous phase. Operation starts with the initial loading of data and terminates when the system eventually becomes obsolete and has to be replaced. During operation, maintenance is required for adopting the system to other requirements (conditions, functions, structures) and for correcting errors that were not detected during validation.

The design process is mainly concerned with the requirement acquisition, requirement analysis and conceptual design. We can distinguish different phases in database design. Based on the waterfall model of software development the classical scenario for structure-driven database design distinguishes between conceptual, logical and physical design. During conceptual design a high-level description of the database (structure, semantics, operations, distribution) is developed using the data and operations requirements. This description is independent from the platforms or the database management systems which are later used. Conceptual design results in a conceptual schema which is kept, for instance, for later database maintenance. During logical design a conceptual schema is translated to a logical schema which describes the structure, operations, semantics and distribution of the database according to a class of database management systems (or according to a single system or, in general, according to a set of paradigms). The logical schema is transformed into the physical schema during physical design. The storage structures, access methods, the organization in secondary memory is developed for a specific DBMS.

Alternatively, the processing-driven database design starts with the processing analysis of requirements which generates a functional schema. This is used during high-level application design for the generation of the application specification which describes the behavior of the database in a high-level abstraction. These specifications are transformed into a detailed program specification during the
application program design phase.

Modern, more recent approaches try to unify and to combine both the structure-driven and the processing-driven database design.

An optimal design of a database is difficult to achieve. It requires that a thorough and careful requirement analysis be undertaken to determine how user’s different and conflicting information need can best be satisfied. These requirements can be classified into three general categories of information:

- **Corporate constraints** describe such restrictions as the operational policies and rules of an organization, the law regulations, the spatial, temporal, and financial constraints, etc.

- **Processing requirements** describe planning (by top management), control (by middle management), and operations (by end users).

- **Information requirements** describe data structures, entities, associations, and semantic integrity constraints of objects which form the user’s view of the database.

The analysis, specification and modeling of these three categories of information is an essential step in database design. The three categories are very much interrelated and can conflict each other. The first category could be considered as a special kind of *commonsense knowledge*. From the AI point of view we can distinguish among different kinds of knowledge which is used during the design process. The theory behind the models and tools could be considered as the *strategic knowledge* or the *meta-knowledge* of the design process. The algorithmic support could be considered as the *judgemental knowledge* of the design process since it describes a reasoning procedure of the designer. The part of the information the designer uses during design could be considered as the *factual knowledge*. The designer uses for performing the task only some of the characteristics and some input information.

The information units in Figure 1 are generated from the above discussed information categories. For instance, the interface is designed in accordance to corporate constraints. The operations required for database operating in the given application are generated from the processing and corporate information.

There have been many publications in the area of specification, analysis, and modeling of information requirements. However, little work has been done on the specification, analysis, and modeling of corporate constraints and processing requirements (an exception is [Su85]). Processing requirements are mostly used only to guide the physical database design process. Its purpose is to produce a design which is *implementable* on a *given platform* and *efficient* and capable of meeting the predefined processing requirement of the present and projected applications. However, processing requirements should be specified independently from given platforms.

For requirement specification in database design, two things are essential: a set of constructs which represent the characteristics of structures, semantics, processes or organizations and a language for specifying the requirements of a database user in terms of these constructs. This language should be a high-level language easy for the users and database designer to employ. Further, the languages for the three categories should be well-integrated.

The requirement acquisition is the most difficult step during database design. There are several reasons for that:

- The designers start with vague ideas of the orderers. Those may have difficulty putting their ideas into words.
- It may be difficult to obtain reliable requirements. Some of the participating orderers may tell the designer what they think the designer wants to hear.
- There can be envisioned a functionality that is not consistent with the objectives that is accomplished from the system.
- Objectives of different groups may compete or conflict.
Organizational interactions may impede among orderers.

The development of a database system encompasses many knowledge engineering activities in addition to requirement acquisition like accessment of the suitability of the database technology for a prospective application, planning and management of a database project, design of appropriate representation formalism for the information that is acquired, design of the inference and control mechanisms to use the knowledge, implementation and selection of suitable platforms, design, prototyping and implementation of the system, integration of the system with its environment and evaluation of the developed system.

However, the information on which the complete development process is based is acquired during the requirement acquisition process. Therefore, this process could be the bottleneck of the complete design process. If the database design process is supported by tools then the requirement acquisition process is one of the basic processes which needs to be founded. For this reason, sophisticated interview techniques need to be integrated.

A poorly designed database system may be unreliable, contain redundant and/or inaccurate data, perform poorly, be inflexible, or characterized by any number of other standard "anomalies". We list some common mistakes in structure-oriented database design.

1. The spreadsheet design or the trash table. Database user, frustrated by confusing documentation, give up and put everything into one monster file or table. To avoid this problem, one should understand that each object (table) should correspond to an entity or express a relationship. Example: file on students, instructors and courses.

2. Megaattributes. Attributes are building blocks of database tables. They should contain the smallest meaningful units of information. Example: Name of person with "firstName+LastName". Otherwise the design will lack of flexibility. Example: Sorting on the name is sorting on the first name.

3. No key attributes. Keys are -like keys for a door- a fundamental part of any database design and should guarantee that the database will find only one exact match. They are helpful for maintaining the "entity integrity".

4. Bad keys. Despite caveats about the importance of unique keys, designers select keys that aren’t unique. Example: Phone in Department. This information may change. Two departments can share the same phone... Mostly, it should be assumed that keys should not contain information, but should be arbitrary numeric or alphanumeric codes. Furthermore, be sure to find out whether the DBMS will perform better with numeric rather than alphanumeric keys before deciding on a data type.

5. The ivory tower syndrome. Database design is a miniature version of systems analysis and design. Therefore, the first stage is more talk than action: talking to both management and users to find out what data needs to be stored - and what doesn’t (pointing out problems with, and bottlenecks in the current systems).
6. Do we really need the *kitchen sink*? It is easy to over-model an organization. Do we really need a long domain field for all cases of real world?

7. **No associations.** Associations relate data in different tables. Especially in relational databases, it turns out that the best way to associate the two tables is to include the corresponding attributes in the current relation. This column’s presence may seem redundant, but it’s necessary. This association column is often referred to as a foreign key. Without this association column, you would have a way of relating entities if there are no corresponding relationships.

8. **Wrong associations or too many associations.** Often relationships are overloaded. Associations should not be put in the wrong table. This will cause major problems the road when you try multi-table views and database operations like join and union. 

   *Example: If the relationship CourseEnrollment contains also a link to instructors then this information is redundant. If the relationship StudInstr expresses the link between the instructors and teachers then this information can be extracted already from the database using CourseEnrollment and CourseOffering.*

9. **Bad many-to-many-to... relationships.** A good form will "feed into" more than one table at once, so users won’t face redundant data entry. Another problem with many-to-many-to... relationships is simply the failure to recognize them.

10. **Too much redundancy.** A well-designed database won’t have any redundancy other than required by associations.

   *Example: Students address in Student.*

11. Using the **same attribute** name. This is a tricky one. Designers are often tempted to name their key attributes something like "ID". But multi-entity operations won’t work properly if you use the same attribute name for different data.

   *Example: The name of a department is different from a name of a student.*

12. **No planning for the future** (see also the ivory tower syndrom). This should be also an integral part of selecting hardware and software.

13. Miscellaneous. Watch out for null values, especially in keys, lawless design, poor security.

### 3.1 On Database Design Methodologies

Database design is implicitly based on some assumptions [Hai91]:

- A database must satisfy a limited set of basic requirements used also in software engineering (correctness, time performance, space performance, compatibility with organizational constraints) and special database requirements (availability, reliability, compliance with a data manager, compatibility with semantical constraints). Satisfying each of these problems constitutes a special identifiable design problem.

- The solution of the above mentioned problems is the objective of the design process. Each step of this process produces a piece of specification.

- Independent from the used methodology a database design process requires the solution of these problems. Therefore, the design process consists in carrying out a limited set of processes.

- Each design process is based on specific concepts, techniques and reasonings.

- Each step of the database design process can be expressed as a transformation (adding, removing, changing the specification).
Therefore, for solving the general design task we a theory-based generic model of database design steps. This model can be used also for re-engineering of existing databases. In this case, it suggests to search the description of the database for traces of each specific design process.

A design methodology could be based on monotone reasoning. Each design step is adding certain information. During design the information already modeled is not changed. This does not mean that structures, operations and constraints are not changed. However, the new design step does not abandon already obtained information. Each design step needs to satisfy certain restrictions [Lie85]:

- Each design step is content-preserving, i.e. there exists a query \( q \) such that for each database \( db' \in SAT(S') \) : \( q(db') \in SAT(S) \) for \( S, S' \) and \( Step(S) = S' \).

- Each design step is constraint-preserving, i.e. \( \Sigma' \models \Sigma \) for \( S = (Struc, Ops, \Sigma) \), \( S' = (Struc', Ops', \Sigma') \) and \( Step(S) = S' \).

- Each step is minimally content-preserving, i.e. the design step does not add derived types; there does not exist a query \( q \) such that for each database \( db' \in SAT(S') \) there exists a database \( db \in SAT(S) \) such that \( q(db) = db' \) for \( S, S' \) and \( Step(S) = S' \).

Furthermore, there are some desirable restrictions:

- Each step is minimally constraint-preserving, i.e. \( \Sigma \models (\Sigma' |_{struc}) \) for \( S = (Struc, Ops, \Sigma) \), \( S' = (Struc', Ops', \Sigma') \) and \( Step(S) = S' \).

- Each step is nearly free of path tuple-generating constraints. When a path tuple-generating constraint is present it often requires that the path has to be computed to check whether the constraint holds in the database.

Notice, that inclusion constraints can be maintained in a database using only projections. Path inclusion constraints are still based on joins. The last restriction is often too strong. It can be replaced by the following restriction. This restriction requires an effective management of basic database operations.

- Each step is update simplicity-preserving, i.e. if for \( S \) the complete set of basic update operations is defined without join operations then \( Step(S) \) has this property too.

A schema is lexical-minimal is it uses as few lexical units as possible. For instance, the attribute DepartmentName of the type Department is can be replaced by Name (the corresponding identifier is Department.Name).

It should be mentioned that several known normalization algorithms do not fulfill the above requirements. However, there are 4NF algorithms which preserve all properties.

Other requirements which are often mentioned in the literature are too complex and can lead to conceptual schemata which are not minimal. For instance, the requirement that each type to be inserted into the schema is in BCNF leads to local normalization instead of global normalization.

A methodology needs also rules in which case which concept is going to be applied. In Chapter 2.1. we considered already the object-oriented approach to database design. One rule which could be considered as the basic rule is that independent objects are to be represented by entity sets. Several approaches, especially those based on the original entity-relationship model, state that attribute values can be also represented by additional, artificial entity types, e.g. choosing an entity type representation when we understand that several properties (attributes, relationships, generalizations, subsets, ..) can be associated with the object to be modeled. We represent any property only by attributes. This strictness can be dropped if types are decomposed according to same attribute value (e.g. the attribute sex of a person is used for the horizontal decomposition of the entity set Person). This decomposition is used when we expect that (now or later) some property will be associated to the lower level entity.
types. Nested attributes should be chosen when it is natural to assign a name to it. Each attribute
(nested or atomar) of a type should be as independent as possible from other attributes.

The design process is highly complex. If the design process is considered for one solution of one
specific application task then the process is performed as described above. However, if we develop a
design tool or if we are interested in developing several applications then we need a support for the
variety of design tasks. One solution for mastering this variety is to reuse parts of previous designs
within the current design. This approach could be supported by a database of reusable design objects.
In this case, an intelligent design tool collects during its lifetime also design decisions and support the
designer with general solutions. The tool for itself models the application area.

Then we consider the design process as two complementary activities: the application engineering
and the application development (see also [Gir92]). Application engineering abstracts generic design
information and represents it by general units using a high-level language. During the application
development this abstract information is used for the generation of the specific solution. General units
could be considered as abstract types. They have a structure, a general semantics and operations. They
have a general description which is used for the reification to special solutions. The different kinds of
abstraction (data abstraction (generalization/specialization (Is-Generalization, Is-Specialization), ag-
gregation/decomposition (Is-Part, Is-Decomposed-Into), classification/instantiation (Classified-As, Is-
Instance)), localization abstraction (Has-Design), implementation abstraction (Has-Implementation))
are used for the organization of general units.

General units are used for specific reification in accordance of the given specific application. There
are procedures for
decomposing a description into a set of descriptions (Is-Decomposed-Into),
composing a description from a set of descriptions (Is-Part),
generalizing a description (Is-Generalization),
specializing a description (Is-specialization),
suggesting a design (Has-Design) or implementation (Has-Implementation) alternatives.
Therefore, general units are generic reusable units accombined by an information how to reuse.

The development of a framework for general units is still open. Different approaches like ma-
chine learning and problem solving strategies can be used for this general task. General units can be
considered to be based on a generlization of inheritance.

3.2 Classical Design Methodologies

In software design and in database design we distinguish different design perspectives.

1. The structure-oriented perspective focusses on the structural description of the data. Almost all
known database design approaches are based on the structure-oriented perspective. Sometimes,
the structure-oriented perspective is unified with the semantical perspective. In this cases, the
design of the structure is to be combined with the design of static integrity constraints. The
different strategies based on the structure-oriented perspective are shown in Figure 5.

2. The behavior-oriented perspective (called in [LiB90] integrity-centered; see also [Lip89]) is con-
cerned with the behavior of the database during its lifetime. It can be based on event approaches
[Lip89] or on Petri-net approaches [Obe88, Obe90] and predicate transition systems.

3. The process-oriented perspective is concentrated on the database operating.

The structure-oriented perspective is discussed in a large variety of papers. The process-oriented
perspective can reuse approaches known in software engineering. The behavior-oriented perspective is
a high-level descriptive approach to an integrated specification of structures and operations. Database
design is considered to be a transformation process carried out on integrity constraints. Database
behavior is specified by stepwise reification.
The structure-oriented approach is currently well developed for relational models and was directly
generalized to semantical models like entity-relationship models. It is, however, possible to develop for
semantical models more specific methodologies which are using the specific properties of these models.
One specific methodology is discussed in the next subsection. Generally, a methodology consists of a
strategy, a set of primitives to be used and guidelines on the application of each strategy step and each
primitive. A methodology should be, from one hand side, rigorous, i.e. suggesting a strategy which is
to be used, and, from the other hand side, flexible, i.e. applicable to a variety of applications. These
requirements are contradictory in general. A design tool should be flexible enough to be adaptable to
different design styles and should be rigorous enough to be well-based.

The building block of a methodology is the design strategy. Known strategies are, for example, the
bottom-up and top-down approach. Most of the known strategies are considering only the structural
part of a schema. The strategies are based on different primitives. Most of these primitives can be
considered to be generalizations of the known algebraic operations of the relational model.

3.2.1 Design Primitives

In [BCN92] design primitives were introduced. Although this approach is cleaning up the various
proposals on primitive design steps it is not continuous. It is claimed in [BCN92], for instance, that
top-down design strategies are incomplete. This is mainly due to the used system. Their systems are
incomplete also for other design strategies for several reasons:

1. Only binary entity-relationship diagrams can be generated.

2. It is possible to generate subtypes of entity types keeping the entity type and adding the relation-
   ship types between the original entity type and the subtypes. However, this operation is not
defined for relationship types.

Another problem in the approach of [BCN92] is the graphical representation used for subtypes. We can
use an idea hinted in [Rad92]. We distinguish between structural abstraction and graphical abstraction.
The last is understood as a kind of representational abstraction. In this case, we define a mapping from
the set of possible schemata to the set of graphs. This mapping is called injective if each graph image
corresponds only to one preimage. The mapping is called basic if it is defined as the extension of the
mapping from primitive types (entity, relationship and attribute types) to primitive graphical units
(rectangles, diamonds with arrows, points with lines; may be with names). The structure is context-
sensitive because the corresponding entity types must exist for component types of relationship types.
However, the context sensitivity follows the order of the types. Therefore, the mechanism for defining
a structure are not equal-order-context-sensitive. The mapping mechanism for the presented extended
ER-model is injective and basic. The mapping mechanism used in most ER books is injective but not
basic. The reason for this is that subtypes and their supertype need to be mapped together. If the
mapping mechanism is not basic then during database design the design of the database structure
needs to be combined with the graphical representation. Since in such design languages the graphical
structure is on the equal-order-context-sensitive the graphical structure can change strongly for even
very simple changes. Tools based on such approaches are more complex.

Further, since database design can be considered to be database management on a schema database
we can compare the design primitives with operations in the design database system. In the nested
relational model different operations are used:
projection, selection, partition, join, \( \theta \)-join or the join via a coercion function, union, nest, and unnest.

Additionally, concepts can be added to the schema.
The operations can be seen also as decomposition/composition pairs:
selection / union,
partition / union,
nest / unnest,
projection / product (or join).

We have mentioned already that each database model needs its own primitives according to the constructors used in the definition of types. In HERM we are using the type constructors product (generating tuple-sets) and union (generating clusters or generalizations of subtypes). Therefore, for a complete set of design primitives we need only primitives defining the generation of products (i.e. product and concatenation to components (The last can be expressed via the first operation. However it might be sometimes more convenient to have this operation too.) and the generation of union types. Further, we need operations for initializing the design (in our case, generating basic attribute types and generating basic entity types but not for the generation of relationship types). We define now these general design operations. For that we assume a set \( \mathcal{C} \) of type constructors. Using this set we can define a set of terms and subterms for each term. For instance, if the \( \mathcal{C} \) consists of union \( \cup \) and product \( \times \) then the set \( T \) of terms is defined by the consecutive application of these operators and the subterms are defined by selection and projection operators. Further, we assume \( \bot \in T \) for the empty term.

Composition. Given the constructor \( C \), a new name \( t \) and types \( t_1, \ldots, t_n \). The composition \( \text{compose}(t_1, \ldots, t_n, C, t) \) defines a new type \( t \) using the constructor \( C \).

Decomposition. Given a type \( t \) and terms \( e_0, e_1, \ldots, e_n \) and (possibly empty) names \( N_0, \ldots, N_n \). The decomposition \( \text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n) \) defines new types which are generated by \( e_i \) and are labeled by \( N_i \).

Extension. Given a type \( t = (t_1, \ldots, t_m, \{A_1, \ldots, A_n\}) \) of order \( i \) \((m \geq 0 \ (m = 0 \text{ for entity types}), \ n \geq 0) \), an attribute \( A' \) and a type \( t' \) of order \( j \), \( j < i \) then:
- \( \text{extend}(t, A) \) is the type obtained from \( t \)
  - adding \( A' \) to the attribute set and
- \( \text{extend}(t, t') \) is the type obtained from \( t \)
  - adding \( t' \) to the component sequence.
For our purposes entity types are types of order 0.

Initialization. \( \text{generate}(E) \), \( \text{generate}(A) \) generate an entity type labeled by \( E \) without attributes and an attribute type labeled by \( A \).

3.2.2 The Top-Down Strategy in Database Structure Design

The pure top-down strategy is based on refinement of abstract concepts into more concrete ones. Each step introduce new details in the schema. The design process ends when all structural requirements are represented. According to the above discussed taxonomy of database design operations the following primitive operations can be used during top-down design:

\[
\begin{align*}
\text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n) \\
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \\
\text{generate}(E), \text{generate}(A)
\end{align*}
\]

The advantage of top-down design is the independent consideration of each type during a design step. The designer can analyse one concept ignoring the other concepts. However, top-down design requires from the designer the knowledge on the general picture of the design first. Therefore, complete top-down design can be performed only by experienced designers if the application area is not well-structured.
In [BCN92] a set of primitives is defined (We use here a slight generalization and omit $T_7$ since it can be represented by $T_6,T_8$):

$T_1$ \text{decompose}(E, t_0, e_1, ..., e_n, N_0, N_1, ..., N_n) where $e_1, ..., e_n$ are projection terms, and $e_0 = e_1 \times ... \times e_n \times e_{01}$.

The result of the application of the primitive are new entity types labeled by $N_1, ..., N_n$ and a $n$-ary relationship type $N_0$ on $N_1, ..., N_n$.

$T_2$ \text{decompose}(E, e_0, e_1, e_2, ..., e_n, E, N_1, ..., N_n) where $e_1, ..., e_n$ are selection terms and $e_0 = \text{id}$. The primitive generates $n$ subtypes of the entity type $E$.

$T_3$ \text{decompose}(E, e_1, ..., e_n) where $e_1, ..., e_n$ are selection terms. The primitive generates $n$ subtypes of the entity type $E$ and removes $E$.

$T_4$ \text{decompose}(R, e_1, ..., e_n, N_1, ..., N_n) where $e_1, ..., e_n$ are selection terms and $R$ is an $m$-ary relationship type.

The primitive generates $n$ $m$-ary relationship types and removes $R$.

$T_5$ \text{decompose}(R, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n) where $e_0, e_1, ..., e_n$ are projection terms and $e_i = e_{i1} \times e_0$ for $1 \leq i \leq n$.

The primitive generates an entity type labeled $N_0$ and $n$ new relationship types connected to $N_0$ and $e_i(R)$.

$T_6$ \text{extend}(R, A), \text{extend}(E, A)$ add additional attributes to relationship and entity types.

$T_8$ \text{decompose}(A, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n) where $e_1, ..., e_n$ are atomar terms and $e_0 = e_1 \times ... \times e_n$.

The primitive generates nested attributes via a tuple constructor.

It is easy to see that $T_5$ can be replaced by the double application $T_1; T_1$ of the first primitive. It is obvious, that the set $\{T_1, T_2, T_3, T_4, T_6, T_8\}$ is minimal and the primitives are independent. $T_1$ generate relationship types, $T_2$ generates generalizations, $T_3$ generates disconnected schemes, $T_4$ is needed to generate cycles, $T_6$ is used for the generation of attributes, and $T_8$ generates nested attributes. Notice further that $T_2$ has no counterpart for relationships. $T_4$ is the counterpart for $T_3$.

For the HERM design the primitives should be extended. For instance, the primitive $T_5$ can be extended to the generation of higher-order types:

$T_9$ \text{decompose}(R, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n) where $e_0, e_1, ..., e_n$ are projection terms and $e_0 = e_{01} \times e_1 \times ... \times e_n$.

The primitive $n$ new relationship types on $e_i(R)$ and generates a relationship type labeled $N_0$ on the new relationship types.

This primitive is the relationship counterpart of $T_1$. We can also develop the counterpart of $T_2$.

$T_{10}$ \text{decompose}(R, e_0, e_1, e_2, ..., e_n, E, N_1, ..., N_n) where $e_1, ..., e_n$ are selection terms and $e_0 = \text{id}$. The primitive generates $n$ subtypes of the relationship type $R$.

The graphical representation in HERM uses unary relationship types for $N_i$. In the classical entity-relationship model this primitive is widely applied by copying the original relationship type $n$ times and using additional inclusion constraints. The HERM representation is simpler.

In [BCN92], an example is given for the incompleteness of the proposed operations $\{T_1, ..., T_8\}$. It is demonstrated that the schema with the entity types City, Division, Army, State and the relationship types LocatedIn = (Division, City, $\emptyset$), HeadQuartersOf = (City, Army, $\emptyset$), ProtectedBy = (State, Army, $\emptyset$), HiresFrom = (Division, State, $\emptyset$), Of = (Division, Army, $\emptyset$), In = (City, State, $\emptyset$) can not be generated. This is true due to the following more general fact.
**Proposition 1** Using \( \{T_1, ..., T_8\} \) only planar entity-relationship schemata can be generated.

### 3.2.3 The Bottom-Up Strategy Database Structure Design

Bottom-up design was first developed for the relational model. During bottom-up design each step introduces new concepts not considered before or modify existing ones. During each step the designer checks whether all features of requirements are covered. The designer start from elementary concepts and builds more complex out of them. The advantage of the bottom-up approach is its simplicity. The main disadvantage of the bottom-up approach is that this approach requires restructuring. Therefore, the strategy is not monotone.

The following primitive operations can be used during top-down design:

\[
\text{compose}(t_1, ..., t_n, C, t) \\
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \\
\text{generate}(E), \text{generate}(A)
\]

In [BCN92] the following primitives are defined (We use slight generalizations of the introduced primitives.):

- \( B_1 \) \( \text{generate}(E) \) is used to generate a new entity type.
- \( B_2 \) \( \text{compose}(E_1, ..., E_n, \times, R) \) generates a new relationship type between the entity types.
- \( B_3 \) \( \text{compose}(E_1, ..., E_n, \cup, E) \) generates a new entity type which is the generalization of the previously given entity types.
- \( B_4 \) \( (\text{generate}(A); \text{extend}(R, A)), (\text{generate}(A); \text{extend}(E, A)) \) generate a new attribute and add it to an entity or relationship type.
- \( B_5 \) \( \text{compose}(A_1, ..., A_n, \times, A); \text{extend}(t, A) \) creates a nested attributes and adds it to a type.

Primitive \( B_3 \) needs also to check which attributes are to be shifted to the generalization.

Bottom-up primitives are minimal and complete for the original entity-relationship model.

### 3.2.4 The Inside-Out Strategy in Database Structure Design

The inside-out strategy restricts the bottom-up approach by controlling the order of primitives application. This strategy is still complete. We choose the most important concept first, design it and then proceed by moving as an oil stain does, designing first concepts that are conceptually closer to the already design. The order of refinements is disciplined. The designer navigates from the center type to the more distant ones. It is easy to discover new concepts which are to be designed next. However we loose abstraction capabilities. The global schema is built at the end only.

### 3.2.5 The Mixed Strategy in Database Structure Design

Another controlled approach is the mixed approach. This approach mixes the top-down and the bottom-up approach. First, a skeleton schema is to be designed (using one of the previous approaches). This schema represents the main classes (or units) and their main associations. Then each of the units is refined and later integrated with other designed units. Using the skeleton schema the bottom-up integration of concepts is simpler. Since the complete requirements set is now partitioned the design of each unit is less complex. The success of the strategy depends from the design of the skeleton schema. Therefore, this method is applicable if the application is already well-recognized.
3.2.6 AI Strategies in Database Structure Design

If we compare different strategies then we can relate them to known search strategies in AI. Therefore, other design strategies can be developed based on this analogy.

3.3 Design by Units

In [YaT89, Tha91', Tha89] another new design methodology was developed: Design-by-units. Most of the well-known design methodologies think as in the relational approach. But each of the database models should have its own methodology. It is surprising that most of the proposed models do not have its own design methodology. If the model is getting richer in construct the methodology should be deepen. One of the database models with its own methodology is the ER model. However, there is still a little agreement in which cases objects from the real world should be modelled by attributes, entities or relationships. A part of the problems of view integration is caused by this modelling problem. And this contradicts the belief of experienced database designers. Those assume that the views of an enterprise can be integrated since there is an internal integration in the enterprise. The reason for this mismatch is that methodologies are not supporting abstraction in an efficient manner. The new design methodology can be understood as a step towards a well-integrated methodology. The proposed methodology is truly object-oriented and at the same time also theoretically based what supports the implementability. This methodology support also extensibility since using this methodology an existing design and implementation can be extended without introducing changes to it. It promotes reuse and inheritance as well as behavioral extension. To some extend, this approach is similar to modular design known in software engineering. The orientation is different only. We are first interested in the data representation part and then in the processing part since a part of the processing part is based on generic operations which are defined according to the structure. However, if we consider modularization, parametrization and inheritance to be the kernel concepts of object-oriented design then this design approach can be considered to be complety object-oriented.

This approach has further other advantages: it is easier to detect similarities among design units and to reuse parts of design units in other units: changes to the scheme and to parts of units are directly reflected in all other units which are using the changed. The new methodology supports directly the above discussed distinction between kernel and dependent object. This is especially useful, if abbreviation techniques [Sci91] are used in query forms [YaT89]. It is a recursive method. Each step is based on the above discussed eight design steps. This methodology is not following the classical waterfall model with iterations but rather supporting a high level inside-out-strategy [BCN92]. Experience in utilization of (DB²) has shown that this methodology was the most often choosen for practical design.

Design-by-units

1. basic step.
   Design the types for the independent kernel object classes.

2. Recursion step.
   Repeat until the schema is not changed.
   Either reification :
   1. Refine the units introducing subtypes (in HERM represented by unary relationship types).
   2. Refine the relationship types according to the subtypes.
   or generalization of units:
   1. If there are associations among units then introduce a new unit containing the associated units and the relationship type according to the association.
   2. Add other relationship types if there exist new associations.
Let us illustrate this method (the structural part) using the following medical example. The database covers the following information:

- known diseases (with a list of possible symptoms (estimation of the probability (whenever a patient has the disease, he also has the symptom with a higher acuteness) and a list of possible factors (drugs, drug combination, other diseases) which may aggravate, cause or cure the disease);
- known symptoms (names, magnitude of their intensity/ acuteness);
- known drugs with their components;
- patients (names address, date of birth, medical history (present and past illness (duration, diagnosing physicians, drugs prescribed for them) and reported symptoms (duration of the occurrences, magnitude of intensity/ acuteness, persons who reported or measured the occurrence (patient himself, relatives, medical personnel));
- physicians (area of specialization) (physicians can be also patients);
- other persons (names, addresses).

First we design the basic units and then we relate these basic units like the units Person and Disease in Figures 2 and 3. Then we merge these units obtaining larger, more general units. One relationship type which is to be designed during the merge of the units Person, Disease and Drug is the type Cure. In the last iteration the schema in Figure 4 is designed.

![Diagram of the unit Person](image)

Figure 2: The unit Person

The final result of the design by units is represented in Figure 4.

Therefore, the proposed strategy could be understood as a controlled, second-level strategy in Figure 5.

3.4 Process Design

As already discussed in the previous section, the data design and the process design cannot be separated from each another. We need the process information as well as the structural information. For this
reason the process design and the structural design need to be integrated. We use a dataflow approach [BCN92]. A process is an activity of the information system. The dataflow is an exchange of information between processes. The processes use information from the database and create temporary databases necessary for the process. Since processes use different databases and these databases are usually not fully integrated, interfaces are to be used for view cooperation. Generally, the interface is the description of the cooperation between different users (originator/receiver of the dataflow).

The processing-requirement modeling at the data level has several applications:

1. **Completeness and consistency checking of information requirements**: One requirement in conceptual database design is that the information requirements of a user group are complete and semantically consistent (not conflicting) with the processing requirements. A process model can be used to verify the completeness and consistency of other requirements.

2. **Identification of proper data entries**: Processing requirements can add different temporal and persistent database structures to the schema. In this case the view integration is more complex. One of the difficult tasks in processing modeling is to evaluate whether the designed data structures are appropriate for an effective processing of data. It is known already that relational normalization can contradict effective processing. Sometimes unnormalized relations can be used simpler. For handling this we need a cost model for processes. The cost model can be based on models of complexity for operations and on priority functions for queries and transactions.

Therefore, we need a representational and a graphical language for the representation of processes. For that we use notions known from dataflow diagrams which are represented in figure 6.

Processing requirements can be modeled at the data, query, transaction and program levels. Since the database structure is defined by a graph *queries* (the user's interaction with the database) can be defined by a traversal of the graph or by a graph generated from the schema. We can use a set of anticipated queries to determine the schema with the query optimal behavior. Each query is defined by a graph, its frequency of use and its priority. Furthermore, we can use additional (cardinality) values for estimation of the queries volume like the
- the estimated number $N$ of occurrences associated with a type,
- the average number $P$ of associated elements for a given element,
Figure 4: HERM-Diagram of the Medical Application Database
Figure 5: Structure-Oriented Design Strategies
Using these numbers, one can determine which of the alternatives has a better behavior. For that, each edge in the query graph defines a workload (e.g. the frequency of the traversal via this edge multiplied by a schema value depending on \( N, P, K, H \)). This is used to determine the workload of a query. Then the alternatives define the workload for all queries and can be compared.

A transaction is a (nested) sequence of database operations which transforms consistent databases into consistent ones. They can be described like queries. However, modeling of transactions provides additional information about the relationship and the order of database operations. This information can be used for selecting alternative HERM designs and for the translation to the most efficient logical schema.

In the same manner, an application program can be viewed as a structure of queries and/or transactions that manipulate a database or several databases.

Figure 7 shows a part of the process description for the request of drugs in the hospital. This process description is embedded into the schema graph for modeling of processing complexity. If the drug is available in the departments pharmacy then the physician is informed. If not then the pharmacies of other hospital units are consulted and requested. If the drug is not available in the hospital the central pharmacy is consulted. The central pharmacy use its store or requests the supplier. The drug is then send back to the hospital with an invoice. If the drug is not available the physician is to be informed.

4 HERM and Design Systems \((DB)^2\)

There are several features which are desirable.

Design systems need a sophisticated user interface which are based on a consistent screen representation and interaction paradigms and are well-integrated in the design process. Flexibility and breadth of
Figure 7: Complex Transaction RequestADrugFromPharmacy
coverage influence the usability. The robustness and architecturally and functionally integration eases the use. The system should help track and control the sequence and interrelationships of different versions. One basic desirable feature of design systems is that the tool is well-integrated into the current technology. It should be understood as a part of groupware (where groupware can be understood as computer-based systems that support groups of people and systems engaged in a common task and that provide an interface to a shared environment). Finally, a system should be extensible into various directions.

We discuss now how these requirements are met by two tools.

4.1 The Design System $(DB)^2$

The system $(DB)^2$ (Data Base Design by Beta; $DBDB = (DB)^2$) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. $(DB)^2$ supports database design from requirements analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual design, thus offering a choice between the relational, network, or hierarchical models for the logical design. Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can be used for creating query compilers which are more productive, effective and forceful.

One of the most important advantages of the system is the interactive user-driven input of information. It is used an easily intelligible, comprehensible and understandable “fill in the blanks” input procedure which is required in the literature as the ideal input form.

The tools developed in $(DB)^2$ can be divided into the following groups:

1. Analyzers which produces different types of reports. They include the schema checker, the normalizer, the logical and physical access evaluation tools, and the index selection tools.

2. Transformers produce a new scheme with information content equivalent to the input. The transformers handle scheme translation as described in normalization theory. They should cope with missing data if need be. The large translations among data models depend on a set of smaller transformations such as removing attributes, declaring keys, and create link records.

3. Heuristic, user-dialog-driven tools produce a new scheme not equivalent to the input. They are invoked early in the design process, and make best-guess decisions to quickly produce a usable, editable result. The view analyzer looks for homonyms and synonyms in two schemes. Synonyms, i.e. attributes having approximately the same meaning and domain and used in different relationships, and homonyms, i.e. attributes having the same spelling but a different meaning in different relationships, can be detected by tools and user-driven identified or distinguished.

4. After partial view integration, a translator produces a scheme appropriate for the target data model. The translation to the physical level adds suggested indexes and clustering.

5. After partial view integration, a query definition module is used for the definition of query-forms. These query forms are to be used for the normalization of the translated schemes. The end-user is able to define queries on the HERM level.

6. The scheme and the defined queries are translated into relational, network or hierarchical schemes and queries. In the case of the relational translation there are implemented four different alternatives which can be chosen according to the properties of interrelationships and according to the support of provided by the DBMS.
7. Since \((DB)^2\) can be used bilingual (English/arabic) the tool needs several routines for the adoption of different directions in writing etc.

The general system architecture is represented in Figure 8 where thick lines denote the data flow and thin lines the control flow.

The underlying methodology can be adapted to the three different views of the design process:

1. the structural view is concerned with the description of the structure and of the static semantics and with tasks like integrity;
2. the behavioral view which contains the description of the operations, usual transactions and the behavior of the database;
3. the implementational view which captures the distribution of the data as well as the storage and management of the data.

Normally, in databases these three views are to be handled independently. The behavior can be specified using generic operations like insert and deriving restrictions to the behavior from the semantical information. The implementational view depends from the underlying systems and from the distribution of the database parts. The strict separation of these three views from each another is the reason for
the mismatch between the abstract given database information and the maintenance of the database during the lifetime. The maintenance complexity is one of the most important database design criterias. The underlying principle of decomposition and normalization in relational databases is the simplification of the maintenance. In order to design a database with the full understanding of the actual complexity of the data and the full understanding of the semantics of the data, the database designer should be supported by a system which captures the structural view as well as the behavioral view and which is able to derive from this information and from the implementational view the corresponding implementation of the database. This task was shown to be solvable using the system $(DB)^2$. The database designer can design the structural, semantical, operational and behavioral information on the application area and the system generates in accordance to the DBMS's type and the distribution the corresponding data structures and transactions.

The clear distinction of independent objects from dependent objects is another advantage of the HERM model and the system $(DB)^2$. This simplifies the modeling since we do not need additional constructs for subtyping. Is-A-relationships are represented directly if they are subtyping relationships. The ER model is often criticized for the unconstrained use of types instead of other types [Cod90, NDT88]. The help tool of $(DB)^2$ can be used for the solution of the problem whether a given concept should be represented by a relationship type or by an entity type or by an attribute. Since this decision depends from the application area no general method exist to get this distinction only from the model information. By answering questions on the meaning of the concept the designer can decide to which kind the given concept belongs.

Database design is at present considered as top-down design. This approach is only useful for single-designer support systems. However, in the case the designer can detect only in a restricted manner similarities between the concepts. This is especially useful if the design is made by several designers. In this case the view integration is one of the most difficult problems which has to be solved for an efficient management of the whole database system. This task can not be solved using only methods of top-down design. The window manager approach in $(DB)^2$ can be used to detect similarities. Moreover, designing the process information at the same time adds the possibility to adapt the structure to the process efficiency. For this reason, new normalization algorithms are used in $(DB)^2$ which prohibit normalization if this contradicts the process efficiency.

Figure 1 shows the dependence graph for the design information. The designer can move from one design part to the other design part according to his skills and his understanding of the design process and the application area. This freedom in choosing the personal design methodology is further supported in the system RAD [BOT90] which is discussed in the next subsection by a special user adaption tool.

4.2 The System RAD

At present the system $(DB)^2$ is widely used for database design. There is a large number of active groups which are using the system. However this system was developed as a prototype and was not developed as an application system. Therefore several extensions and improvements are necessary [BOT90] and are to be developed in the RAD system [BOT90]. This tools provide also a better understanding of design task like abstraction (e.g. analysis of concepts), editing (e.g. correctness, consistency, complexity), refinement (e.g. context representation) and transformation (e.g. prototyping, population or test data generation).

The system RAD [BOT90] is intended to become a toolbox for the interactive, object-oriented design of databases, and which is being developed within a cooperation between the universities of Rostock, Aachen and Dresden in Germany. The development of RAD will be carried out in two stages:

In the first, a system will be realized whose goal is to support designers of a database in all stages of the database life cycle, beginning with conceptual design and ending in documentation and provisions and support of redesign after modifications. This will be done interactively meaning that users are
guided through the various design steps, can get explanations about the systems itself and about what they have designed already, are advised about reasonable design choices, and may consult a “reviewer” who critically inspects the result of a design phase and makes suggestions for modifications if necessary.

In the second phase, the system shall be turned into a genuine toolbox, which on the one hand provides a number of methodologies and tools for database design, but on the other can be configured according to designers’ needs into a system that is customized for a specific application.

RAD is based on two basic ideas: First, database design should be a highly interactive process, in which a designer not only uses the system for drawing, say, entity-relationship diagrams on a screen and has them automatically translated into a relational schema; instead, the system should act as a “design workstation” and hence

- provide the user with a reasonable way to describe the syntax, static semantics and behavior of an application, thereby following an object-oriented approach to systems design,
- assist the user in making design choices and in analyzing a (preliminary) design,
- comprise automatic mechanisms for translating a completed design into a specific data model that underlies a given system, and these should not be restricted to the relational model,
- provide means to modify a design even when a corresponding database has been established already.

Second, database design is a task that arises in a variety of applications, like banks, insurance companies or other commercial enterprises, but also in technical domains like CAD or CASE applications; in addition, database design is carried out by people of different backgrounds: specifically trained employees who have a good knowledge of databases, casual users with little a priori knowledge, and people with an “intermediate” degree of expertise. As a consequence, a design system should support

- different levels of user sophistication,
- distinct types of applications and their requirements,
- various hardware systems, ranging from PCs to workstations.

Thus, it does no longer make sense to offer a design system which is perfectly suited for particular types of applications and/or users, but performs poorly for others. The solution, then, is to build a toolbox that is configurable with respect to a variety of aspects.

Basically, the system in Figure 9 is consisting of four major components:

1. HERM+, the interface to the outside world, in particular to the designer and the application he or she wants to design. It will provide a language based on an extension of the entity-relationship model for defining syntactical structures, static semantics and behavior of a given application and will hence support conceptual database design based on an object-oriented methodology.

2. The Support System, a component to support users of the system during the various phases of a design process. It has two subcomponents: The User’s Guide will comprise a tutoring system for guiding the user through object-oriented design in general as well as through HERM+ and the database design process in particular; it will also support him or her in choosing design strategies, and provide examples. The Analyzer will allow a designer to get online feedback regarding design decisions, and critically review the result of a design.

3. The Translator, a component for translating the result of some database design written in the HERM+ language into the language of a specific database system that is to be used in the given application. Specifically, it is intended to have the Translator support a variety of concrete
Figure 9: Architecture of RAD.
data models, including the relational, nested relational, network, hierarchical and complex object models. Additionally, it should contain a component for performing translations between any two of these models.

4. The **Modifier**, a subsystem which can be incorporated for modifying a given, already completed design or for altering one or more design decisions. To this end, it will consist of means to modify syntactical structures, static semantics as well as behavior, will use a **Normalizer** for performing normalization on parts of a design or an entire design according to user needs, and will have a **Redesigner** for generating the input for the Translator in case a design has been modified.

Figure 1 shows the dependence graph for the design information. The designer can move from one design part to the other design part according to his skills and his understanding of the design process and the application area. This freedom in choosing the personal design methodology is further supported in RAD [BOT90] by a special user adaption tool.

Figure 9 shows that RAD is designed as an extension of \((DB)^2\). During the use of \((DB)^2\) we discovered also several disadvantages. Most of the disadvantages are caused by the prototypical architecture, the limited computational power and by the limited reasoning support of the \((DB)^2\) interface. The last requirement is the more important. Database design is at present performed by "experts" who have detailed knowledge of a design methodology as well as the benefit of experience gained from previous designs. Since these experts are called upon to work in many diverse areas they often do not know to much about the specific application. They need a knowledgable user who supplies the application expertise. As a result, the designer often ask questions that appear unnecessary to the user or teach the user on database design. A better approach is the expert system approach. The user is introducing his knowledge on the application area and the system supports him by translating this knowledge to schemata. For that, the system should use "real world" knowledge, should have reasoning capabilities and should learn from experience, as a human designer. Furthermore, the system should explain the corresponding decisions and questions to the user. Therefore, the design system needs a component based on techniques known in AI.

Further, at present it is planned to develop the system RAD for the support of different, more sophisticated design tasks like:

- During database design different versions of the same design should be stored and considered together.
- Therefore, a design tool needs capabilities known from text processing.
- It is useful to reuse parts of other schemata and to modify parts of other schemata.
- The reuse and redesign of schemata should be supported especially in the case of changing environments (other DBMS, other versions of DBMS).
- If the tool is to be used by designers without deep knowledge of database and system theories the design tool should supervise the designer and explain critical decision.
- If a subschema is not well designed or leads to complex operations for maintenance the design system should be able to support the designer during modification and redesign of the schema.

The database design process could be considered to be a **knowledge acquisition process**. The database design process can be viewed as a special kind of cooperation between the designer and the system. The system requests information that might be useful in the design and makes suggestions for schema reductions and other improvements. The designer supplies the requested information and accepts or rejects the suggestions. The designer may also add information not requested and transform the schema in ways other than those suggested by the system. The design system need also component that enables it to learn during design and to learn the designers behavior. Further, the system needs reasoning capabilities and a knowledge base on **commonsense** knowledge (classificatory and generic; industry-specific and organization-specific knowledge). Whenever, the designer adds user-specific and application-specific information the system can generate the relationships to the world being modeled.
Figure 10: The control flow in RAD
translated schemata  
(structure, generalized triggers, transactions)

Figure 11: The data flow in RAD
These requirements were considered for the system RAD. It has the same input paradigm but contains also AI components. The control flow and the data flow of the system RAD is illustrated in Figures 10, 11. These figures are also used to display the parts which are to be developed in different cites: Rostock (boldface type style), Aachen-Gießen (slanted type style), Dresden (normal type style) and common in all three sites (sanserif style).

The system is based on the graphical tool GraphEd [Him90] which was developed in Passau. GraphEd is a powerful interactive SUN-based editor for graphs, using windows and has a special open data interface [Him90']. The graphical representation uses graph grammars.

5 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system (DB)$^2$ was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and systems have the following advantages:

1. The model has a strong theoretical basis [Tha92'].
2. The modeling is more natural and can be applied in a simple manner. Only necessary facts are to be expressed [Tha90, YaT89].
3. The theory is applicable to practical needs [SST91, Tha89, Tha91', YaT89].
4. The results of the design are much simpler than in other approaches [YaT89, Tha92].
5. The model is easy understandable, simple and perceivable [Tha91'] and can be used as a basis for design tools [BOT90, Tha89, Tha92].

References


[Him90'] M. Himsolt, SGraph 3.0. University Passau, Computer Science Department, July 1990.


Strategies and Methodologies of Database Design
Supporting Entity-Relationship Database Design Strategies*

Margita Altus and Bernhard Thalheim

Abstract

Database design strategies have been mainly considered for the relational database model. These strategies can be extended to entity-relationship modeling. Since the ER model has richer constructs, strategies for design on the basis of ER models should be richer as well. This paper develops strategies for extended ER models which can be used in design systems. Database design tools need to be adaptable to a wide range of designers. The system RADD developed in different German university groups has a component which enables the user to choose his design strategy according to his experience and abilities. This paper surveys strategy support of RADD and proposes the design methodology Design-By-Units.

1 Database Design Tools and the Database Design Process

Database design methodologies and tools should facilitate database modeling, effectively support database processing, database redesign and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970's, various tools for database design have been introduced. Most of them, however, are dependent on knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The systems (DB)2 and RADD (Rapid Application and Database Development) developed in groups in Cottbus, Dresden, Hamburg, Kuwait, Münster, and Rostock [6, 24] do not require the user to understand theory, implementation restrictions and programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. These tools are based on an extended entity-relationship model. The entity-relationship model is extended to the Higher-order Entity-Relationship Model (HERM) by adding structural constructs and using integrity constraints and operations. The system RADD has a component which enables the user to choose his design strategy according to his experience and abilities. Different database design methodologies are developed based on the HERM approach. During the last decade several dozens of computer-aided database engineering tools (CASE) have been developed and used in practice1. Most of them are based on the relational approach. In most cases, third or fourth normal forms are the main goal of the design. Some tools can be used to develop also BCNF. The ER models were getting more popular for database design tools during the last 15 years. These tools support a graphical representation of structures. Some of them are based on wrong theoretical decisions and cannot be used for a design of efficient relational structures.

Based on an analysis of systems supporting database design the database design system (DB)2 has been developed [24]. It is supporting a high-level efficient database design. The system is based on an extension of the entity-relationship model for structural design and on a graphical interface. Further, a high-level language for specification of integrity constraints and operations has been developed.

At present, we can distinguish already three generations ([7, 17]).

---


---

*This research has been supported by DFG Th 465/2.
The first generation tools were based on the classical “waterfall” model of software development. Most of them were platforms for a design from scratch. These tools did not support changes during life-cycle of the database. In most cases, they were using the relational database model.

Second generation tools which become available now are designed as complete workbenches for the design support over the complete life-cycle of a database. Such tools use graphic subsystems and support consistency of the design.

Although second generation tools are now putted into practice third generation tools are already under development. There are already proposals how tools can be customized. Third generation tools will be more user-friendly and user-adaptable (using, for instance, user-driven strategies which are influenced by special organizational approaches in enterprises). Users can use strategies which are model-dependent and have a support for reasoning in natural language. They provide a tuning support. Users can use object-oriented development strategies.

At present, there is no tool which supports a complete design methodology. Most tools currently available support a restricted part of the design process in a restrictive manner. There are tools which do not support integrity and/or are mainly graphical interfaces. Further, there are only very few tools which can claim to be a third generation design tool. Beside this restriction there are other limitations:

- Many design tools are restricted to certain DBMS. Since any DBMS has some implementation restrictions a computationally efficient relational design should be different for each of the systems.
- The number of different normal forms can be hyperexponential according to the number of attributes. Most of tools, however, compute only one normal form. Often this normal form computation is dependent from the order of input information (attributes and functional dependencies).

The current development of technology enables the development of components for a third generation database design system. This system needs to satisfy the following requirements:

**Complete specification of structure, semantics and behavior.** The application should be specified by its structure, its set of integrity constraints, its operations and restriction on operations. The relational database model is not expressive enough. The entity-relationship model can be extended to a model which allows specification of database structure, database operations and database semantics and behaviour. In this case, the system requires a sophisticated methodological framework.

**Natural language support.** Since users are better trained in expressing their application on the basis of natural languages the design system should be able to extract structural, semantical and operational specification of an application from sentences stated natural languages. The system has to generate dialogues in order to discuss the context of statements with the user.

**Advanced and powerful user interface.** Database designers are permitted to use graphical, procedural and logical specification of their application.

**Strategy support.** Database designers prefer different strategies according to their skills, abilities and to applications. They need methodological support and guidance for consistent development of database applications. The design system can check whether a design is complete and which design steps are necessary for completing a design.

**DBMS independence.** The design system should support a variety of DBMS. Implementational restrictions of DBMS should be discussed or used during translation of the database specification.

**Optimization of behavior.** The tool can estimate the complexity of the behavior during database operation.

RADD is a third generation tool which satisfy the requirements above [2, 3, 4]. It will be shown in this paper how RADD satisfies the fourth requirement.

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

- to meet all the information (content) requirements of the entire spectrum of users in the given application area;
• to provide a "natural" and easy-to-understand structuring of the information content;
• to conserve the whole semantic information of the designers for a later redesign;
• to achieve all the processing requirements and achieve a high degree of efficiency of processing;
• to achieve the logical independence for query and transaction formulation on this level.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process. A formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

The requirement acquisition is the most difficult step during database design. There are several reasons for that:

The designers start with vague ideas of the orderers. Those may have difficulty putting their ideas into words. It may be difficult to obtain reliable requirements. Some of the participating orderers may tell the designer what they think the designer wants to hear. There can be envisioned a functionality that is not consistent with the objectives that is accomplished from the system. Objectives of different groups may compete or conflict. Organizational interactions may impede among orderers.

Based on an analysis of systems supporting database design we developed a database design system (DB)\(^2\) [22] which is supporting a high-level efficient database design. The system is using an extension of the entity-relationship model for the structural and operational design and a graphical interface. Further, a high-level language for specification of integrity constraints and operations has been developed. RADD is based on the experience of extensive utilization of (DB)\(^2\) in more than 100 different user groups over five years. (DB)\(^2\) has been used in more than thousand very large projects. Using a browser, analysis of the development process in those projects gave us a deeper insight into the design process and the needs of database designers. Designers use in practice strategies which are more refined, complex, and adaptable to the designer and to the application. Further, designers are using different abstraction and representation levels at different stages of the design process. Finally, the consistence of design decisions of team members has to be supported. Based on this experience, RADD has a better user support system especially for the more difficult design steps like design of integrity constraints, design of operations and maintenance, refinement and rebuilding the schema. Database design strategies are adapted to the model and the purposes of database design processes. This paper reports these results. For an automatic adaption of the design system to the chosen user strategy, known database design strategies [7, 9, 11, 12, 13, 14, 15, 16, 21] have been generalized and a general framework has been developed (For preliminary versions see [28] and for a general idea see [26]..). Further, we developed our own database design strategy which can be extended to some extend to object-oriented models [19, 20]. This paper gives an insight into the support system of RADD and the basics of ER database design strategies.

The paper is divided into five parts. In Section 2, the database design process is briefly discussed. We distinguish design directions, control strategies and modularity. Different operations which are used during database design are developed in Section 3. Different classical design strategies are generalized to a framework in Section 4. Sections 5 discusses a simple sample application.

2 Dimensions in Database Design Strategies

Database design has been based in the past on relational design techniques. Two main approaches have been developed: top-down approaches (decomposition) and bottom-up approaches (synthesis). The first strategy begins with a set of basic types and builds new types using decomposition. The second approach begins with a set of descriptions or attributes and then group those in types.

The relational decomposition approach starts with a given schema of relation types and decomposes it to smaller relation types until certain properties such as normal form and losslessness are satisfied. The starting schema can be a universal relation type or a near-normal-form set of types. In the synthesis approach to
designing relational databases, we assume the application is described by a set of attributes and functional dependencies among them. The functional dependency (FD) is the main vehicle in synthesizing relation types out of lists of attributes. In this case, we start with a minimal cover of FDs. Normally, the two strategies are not applied in their pure form. Database design could be performed by mixing the two strategies which is sometimes called mixed strategy.

These two approaches could be considered as approaches with different design directions. Directions depend on the constructors which are used in the database model. The extended ER model of [25] has the following constructors:

**Tuple constructor:** Two or more types can be composed by product. Objects are represented by tuples according to the component types.

**Set constructor:** Objects on a new type are composed of sets of objects on another type.

**Grouping:** Clusters are used for the representation of the union of types.

**Nesting:** Nest operations are used for grouping objects on another type.

**List constructor:** A new type is defined as a list of objects on another type.

The relational model uses only sets of tuples on atomic types. In the extended ER model attributes can be constructed by using all constructors. Entity types are defined by application of a tuple constructor to attributes. Relationship types are defined on the basis of set, tuple and list constructors. Clusters are defined by grouping. For the purpose of this paper we restrict the extended ER model to the first three constructors. The strategy support can be easily extended to all five constructors.

In the relational model the top-down strategy and the bottom-up strategy explicitly use the simplicity of the type constructors. ER models are defined on a more elaborated type system. Thus, each model should possess its design strategy according to the type system used for the definition of the model. Nevertheless, the HERM model allows to use the classical directions.

There are several reasons why the ER model needs its own design strategy.

**Different models:** The modeling approaches are different. The relational model requires flat representation of objects. The ER model concentrates on semantically meaningful units.

**Environments:** In the relational approach each relation type is developed by its own. Later referential integrity is added for associations. The ER model allows to reason on the type and on its environment, e.g. associated type. Thus ER modeling is more sophisticated.

**Complex type constructors:** Due to the construction mechanism for ER types semantics of types can be also inherited, e.g. relationship types inherit semantics from their component types and by the existence constraint that relationships exists only if the corresponding component objects exist.

**Differentiating usage and definition:** The ER model explicitly differentiate usage and definition. Kernel entity type define values. Relationship types use component types with additional characteristics.

**Avoiding abstract types:** ER modeling gives the possibility to avoid abstract types which do not carry any meaning in the application but are developed because of restrictions of representations. Relational modeling and modeling on the basis of binary ER models make use of abstract types. Since in larger applications nobody can maintain correctness of abstract types those should be avoided if possible.

**Behavior enforcement:** Cardinality constraints and other ER constraints are a simple vehicle to model behavior of operations. Integrity enforcement can be made understandable. Thus, behavior can be explicitly modeled.

**Different kinds of types:** ER model differentiates between attribute types, entity types and, additionally, relationship types. This differentiation should be kept during design.

Pure top-down strategy is based on refinement of abstract concepts into more concrete ones. Each step introduce new details in the schema. The design process ends when all requirements are satisfied. According to the above discussed taxonomy of database design operations the following primitive operation can be used during top-down design:

\[ \text{decompose}(C, (e_0, e_1, ..., e_n), (N_0, N_1, ..., N_n)) \]

for a type \( C \), expressions \( e_i \) and new types \( N_i \).
The advantage of top-down design is the independent consideration of each type during a design step. Bottom-up design was first developed for the relational model. During bottom-up design each step introduces new concepts not considered before or modify existing ones. During each step the designer checks whether all features of requirements are covered. The designer start from elementary concepts and builds more complex out of them. The advantage of the bottom-up approach is its simplicity. The main disadvantage of the bottom-up approach is that this approach requires restructuring. The following primitive operation can be used during bottom-up design:

\[
\text{compose}((C_1, \ldots, C_n), e, C_0)
\]

for already defined types \(C_i\) a new type \(C_0\) is constructed by the expression \(e\).

Both strategies use additionally support operations:

\[
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E)
\]

for an entity type \(E\), relationship type \(R\) and an attribute type \(A\)

\[
\text{generate}(E), \text{generate}(A)
\]

for an entity type \(E\) and an attribute type \(A\).

Design strategies can be distinguished also by their control mechanism. One control mechanism rules the step-by-step development of types. Control strategies decides which rules to use in a given situation. The inside-out strategy selects first one central type and then develops the associated type by a discrete neighborhood function. The next type to be developed is a new type with the smallest neighborhood value according to the central type. Another control mechanism exploits another more abstract schema of the database. This mixed strategy is controlled by a sketch of the database schema (the so-called skeleton). Alike in second-order grammars this control mechanism can be generated by another grammar. The skeleton schema is representing the control information of the database design.

Relational database design has been considered mainly for the design of conceptual schemata according to the three-level architecture of a database. The conceptual schema unifies the different views of external users. The database design could also begin with the view design [9]. The views are later composed into the conceptual schema. This approach can be useful for the design of homogeneous view sets. Since view integration is undecidable the applicability of this approach is limited. The modular design strategy Design-by-units extends this approach. For each subschema which could be considered to be a module similar to software technology the interface is defined for each unit. Units can be associated only by their interfaces. We can define also other kinds of scoping rules in database design. This dimension of database design uses the modularity concept. Modularization is based on implementation abstraction and on localization abstraction. Implementation abstraction [18] is to selectively hide information about structures, semantics and the behavior of concepts defined by the previous two abstraction mechanisms. Implementation abstraction is a generalization of encapsulation and scoping. It provides data independence through the implementation, allowing the private portion of a concept to be changed without affecting other concepts which use that concept. Localization abstraction [18] is used to "factor out" repeating or shared patterns of concepts and functionality from individual concepts into a shared database / knowledge base application environment. Naming is the basic mechanisms to achieve localization. Parametrization can be used for abstraction over partial object descriptions.

Another dimension not represented in the figure is the representation form for design results. For instance, the ER model uses rectangles, diamonds as nodes of a graph. The IFO model uses more sophisticated node types and different types of edges. The representation is more compact but also more complex. The graphical representation differs from model to model. The most important issue is, however, that the graphical representation has to be based on a well-founded semantics. Often, especially in the ER literature and for ER-based tools, design is considered as drawing graphs leaving semantic issues aside.

The power of abstraction principles comes from their orthogonality and their generality. All these mechanisms can be combined with more specialized mechanisms, like exception and default rules.

If we compare different strategies then we can relate them to known search strategies in AI. Search strategies are characterized by direction, topology, node representation, selecting rules and heuristic functions. Other design strategies can be developed based on this analogy. We could search forward or backward.

Summarizing, a CASE tool supporting database design needs for strategy support different features:

- **Primitive operations** are used for constructing new types.
BCI = Bottom-up, (conceptual,) inside-out design
BCU = Bottom-up, (conceptual, uncontrolled) design
BCM = Bottom-up, (conceptual,) mixed design
BMI = Bottom-up, modular, inside-out design
BMM = Bottom-up, modular, mixed design
BMU = Bottom-up, modular, (uncontrolled) design
BVM = Bottom-up, view-oriented, mixed design
BVI = Bottom-up, view-oriented, inside-out design
BVU = Bottom-up, view-oriented, (uncontrolled) design
TCI = Top-down, (conceptual,) inside-out design
TCM = Top-down, (conceptual,) mixed design
TCU = Top-down, (conceptual, uncontrolled) design
TMI = Top-down, modular, inside-out design
TMM = Top-down, modular, mixed design
TMU = Top-down, modular, (uncontrolled) design
TVM = Top-down, view-oriented, mixed design
TVI = Top-down, view-oriented, inside-out design
TVU = Top-down, view-oriented, (uncontrolled) design

Fig. 1: Different Structure-Oriented Database Design Strategies
Control operations restrict the generation procedure for design.

Modularity operations keep track on consistency of separately developed subschemes.

Further, the representation form of the corresponding model should be supported. In RADD the graphical editor GraphEd [10] is used for this purpose. Finally, intelligent browsers support consistent design.

3 Design Primitives

A design methodology could be based on monotone reasoning. During design the information already modeled is not changed. This does not mean that structures, operations and constraints are not changed. However, the new design step does not abandon already obtained information. The methodology can be represented by the quadruple

\((M, \Sigma_M, P, T)\)

where \(M\) is the metamodel, \(\Sigma_M\) is the set of metamodel constraints, \(P\) is the set of primitive steps that are available in the methodology, and \(T\) is the transition system.

Each design step is adding certain information. It can be modeled as a triplet

\((s, d, a)\)

where \(s\) is the situation or environment of the step, \(d\) is the decision and \(a\) is the action. Decisions are design operations and strategy tasks. Each design step needs to satisfy certain restrictions [27]: minimality-preserving, content-preserving, constraint-preserving, simplicity preserving and closed. Design steps can be based on scripts and actions. Scripts define the situation and the operation.

Scripts can be now defined for the definition of the environment of the design step. The script consists of the following components:

Schema segment: Views or selections of types define the environment of the design step under consideration.

Types to be changed: Types can be changed, removed or renamed by design steps.

Operations: The operation of the design step is defined by expression from the type constructor algebra.

Types to be introduced: The operation to be applied generates new types with new names.

Restrictions: Since types do not exist in isolation in the schema the environment should be considered as well. Thus, any action of an operations is changing the environment as well.

Concept preservation: The type which is to be changed by the design step can be preserved or be removed. We denote preservation by \(T^+\) and removal by \(T^-\). The constructors generate root type which is either a supertype or a subtype of the remaining new types. This new type can be preserved (denoted by \(S^+\)) or removed (denoted by \(S^-\)) after the design action has been performed.

Kind preservation: Types are either attribute type or entity types or relationship types (of order \(i\)). The new types can be of the same kind as the original type (denoted by \(C^N\)). If the original type has been preserved then its kind can be preserved as well (denoted by \(C^T\)). If the root type is preserved and this type is of the same kind as the original type then we denote it by \(C^S\).

Input/Output specification: A type which is under change can be a component of another type (specified in HERM diagrams by an incoming edge). Further, a type can have components (specified in HERM diagrams by outgoing edges). The incoming edges can be assigned to new types, to the root type or to the original type (we denote this by \(I^N, I^S, I^T\) respectively) if it is possible. Another assignment is possible for outgoing edges \((O^N, O^S, O^T)\).

Thus, a script is defined by a 5-tuple.

We can use general scripts. Since a design system cannot support all possible scripts we need to restrict this variety in such a way that all necessary design actions can be performed. In our strategy model we use only two different directions: top-down and bottom-up. Thus, the main operations are decompose and compose. Using the above defined type constructors pairs of destructors and constructors are defined in figure 2. The arrow denotes the component association.

Thus, we develop now the primitive operations by extending and refining those proposed in [7]. The graphs pictured in figures 3, 4, 5, 6, 7, 8, 9, 10 are used as graph rules for decomposition. For simplicity we assume that original types have only one input and one output type. Further, decomposition is a decomposition into three new types and a root type.
<table>
<thead>
<tr>
<th>Operation as</th>
<th>destructor</th>
<th>constructor</th>
<th>assignment root type new types</th>
</tr>
</thead>
<tbody>
<tr>
<td>project/join-decomp.</td>
<td>projection $\pi$</td>
<td>join $\nu$</td>
<td>$\times_{\pi e_1} \ldots_{\pi e_n}$</td>
</tr>
<tr>
<td>complete decom.</td>
<td>projection $\pi$</td>
<td>product $\times$</td>
<td>$\times_{\pi e_1} \ldots_{\pi e_n}$</td>
</tr>
<tr>
<td>horizont. decom.</td>
<td>selector $\sigma$</td>
<td>union $\cup$</td>
<td>$\cup_{\sigma e_1} \ldots_{\sigma e_n}$</td>
</tr>
<tr>
<td>disjoint horiz. dec.</td>
<td>selector $\sigma$</td>
<td>union $\cup$</td>
<td>$\cup_{\sigma e_1} \ldots_{\sigma e_n}$</td>
</tr>
<tr>
<td>grouping</td>
<td>unnesting $\mu$</td>
<td>nesting $\nu$</td>
<td>$\nu_{\mu e_1} \ldots_{\mu e_n}$</td>
</tr>
<tr>
<td>extension by compon.</td>
<td>projection $\pi$</td>
<td>tuple construction $\circ$</td>
<td>$\circ_{N_1} \ldots_{N_n}$</td>
</tr>
</tbody>
</table>

Fig. 2: Operations using for Decomposition and Composition

$$I_1 \rightarrow C \rightarrow O_1$$

Fig. 3: Script$(S, C, (\lambda, \pi e_1, \pi e_2, \pi e_3), (\perp, N_1, N_2, N_3), (T^-, S^+, C^N \cup C^S, I^S, O^N))$

$$I_1 \rightarrow C \rightarrow O_1$$

Fig. 4: Script$(S, C, (\lambda, \sigma e_1, \sigma e_2, \sigma e_3), (\perp, N_1, N_2, N_3), (T^-, S^+, C^N \cup C^S, I^S, O^N))$

$$I_1 \rightarrow C \rightarrow O_1$$

Fig. 5: Script$(S, C, (id, \pi e_1, \pi e_2, \pi e_3), (N_0, N_1, N_2, N_3), (T^+, S^+, C^N \cup C^S C^T, I^S, O^N))$
Fig. 6: Script($S, C, (id, \sigma_{e_1}, \sigma_{e_2}, \sigma_{e_3}), (N_0, N_1, N_2, N_3), D$) for $D \in \{(T^-, S^+, C^N \lor C^S, I^S, O^N), (T^+, S^-, C^N \lor C^T, I^T, O^N)\}$

Fig. 7: Script($S, C, (id, \sigma_{e_1}, \sigma_{e_2}, \sigma_{e_3}), (N_0, N_1, N_2, N_3), D$) for $D \in \{(T^-, S^+, C^N \lor C^S, I^S, O^N), (T^+, S^-, C^N \lor C^T, I^T, O^N)\}$

Fig. 8: Script($S, C, (id, \pi_{e_1}, \pi_{e_2}, \pi_{e_3}), (N_0, N_1, N_2, N_3), D$) for $D \in \{(T^-, S^+, C^N \lor C^S, I^S, O^N), (T^+, S^-, C^N \lor C^T, I^T, O^N)\}$

Fig. 9: Script($S, C, (id, \sigma_{e_1}, \sigma_{e_2}, \sigma_{e_3}), (N_0, N_1, N_2, N_3), (T^+, S^+, C^N \lor C^T C^S, I^S, O^N)$)

Fig. 10: Script($S, C, (id, \circ(e_1), \circ(e_2), \circ(e_3)), (C, N_1, N_2, N_3), (T^+, S^-, C^T, I^T, O^N)$)

Fig. 11: Script($S, C, (\pi_{e_1}, \pi_{e_2}, \pi_{e_3}), (N_0, N_1, N_2, N_3), (T^-, S^+, C^N, I^S, O^N)$)
[7] defines as a primitive operation further the binarization of relationships. This operation can be represented by the script in figure 11.

The composition scripts for bottom-up operations can be defined in the same manner.

The RADD tool does not require the user to use complex scripts. We developed a framework which is similar to the representation in figure 2. The user can add his own scripts. The application of scripts has the advantage that the system can support each design step and is able to detect inconsistencies to a certain extend. Further, the design history can be recorded. This record enables RADD to analyse the designer's behavior. If a designer chooses his own strategy then this strategy is analysed. If the designer is often using support steps or often correcting the schema then the used strategy is analysed. Since it is not possible to provide designers with all potential strategies RADD stores those histories for later refinement of the strategy support.

The underlying graphical editor GraphEd is based on graph grammars [10]. Since a script can be directly represented by a graph grammar rule, GraphEd has a direct support for implementation of scripts.

It is easy to prove that the proposed set of primitives is complete. Each schema can be generated using this set of primitives and the generate primitives. Also several proper subsets of the set of primitives have this property. We have chosen a set of canonical top-down primitives and a set of canonical bottom-up primitives. The canonical sets are used whenever the designer does not want to develop his own strategy.

4 Composing Design Strategies

Control for Design Strategies

In RADD the control is based on graph grammars. Each graph grammar rule represents primitive operations [5, 23]. For instance, based on the extended entity relationship model the following canonical primitives can be used:

<table>
<thead>
<tr>
<th>Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁</td>
<td>generate(E)</td>
</tr>
<tr>
<td>B₂</td>
<td>generate(A)</td>
</tr>
<tr>
<td>B₃</td>
<td>Script(S, (E₁, ..., Eₙ)∪E, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₄</td>
<td>Script(S, (R₁, ..., Rₙ)∪R, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₅</td>
<td>Script(S, (A₁, ..., Aₙ)∪A, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₆</td>
<td>Script(S, (A₁, ..., Aₙ)×E, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₇</td>
<td>Script(S, (C₁, ..., Cₙ)×R, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₈</td>
<td>Script(S, (A₁, ..., Aₙ)×A, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>B₉</td>
<td>Script(S, A₁, {}, A, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>T₁</td>
<td>Script(S, E, (λ, π₁, π₂, ..., πₙ), (∧, N₁, N₂, N₃), (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₂</td>
<td>Script(S, R, (π₁, π₂, ..., πₙ), N₀, N₁, N₂, N₃, (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₃</td>
<td>Script(S, C, (π₁, π₂, ..., πₙ), N₀, N₁, N₂, N₃, (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₄</td>
<td>Script(S, E, (λ, π₁, π₂, ..., πₙ), (∧, N₁, N₂, N₃), (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₅</td>
<td>Script(S, R, (λ, π₁, π₂, ..., πₙ), (∧, N₁, N₂, N₃), (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₆</td>
<td>Script(S, A, (λ, π₁, π₂, ..., πₙ), (∧, N₁, N₂, N₃), (T⁻, S⁺, Cᴺ, Iˢ, Oᴺ))</td>
</tr>
<tr>
<td>T₇</td>
<td>Script(S, E, (id, π₁, π₂, ..., πₙ), N₀, N₁, N₂, N₃, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>T₈</td>
<td>Script(S, R, (id, π₁, π₂, ..., πₙ), N₀, N₁, N₂, N₃, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>T₉</td>
<td>Script(S, A, (id, π₁, π₂, ..., πₙ), N₀, N₁, N₂, N₃, (T⁺, S⁻, Cᵀ ∨ Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
<tr>
<td>T₁₀</td>
<td>Script(S, A, (id, o₁, o₂, oₑ₃), N₀, N₁, N₂, N₃, (T⁺, S⁻, Cᵀ, Iᵀ, Oᵀ))</td>
</tr>
</tbody>
</table>
Now based on canonical primitives, several strategies can be defined. Strategies are constructive theories, i.e., a strategy can be defined by a construction system:

**Primitive steps**: Each primitive unit is a step for itself.

**Construction rules**: Based on already defined steps new steps can be constructed using connectives.

**Consistency rules**: The construction process is limited by an order, by consistency constraints etc.

So far we defined primitive steps like $B_1$ and $B_2$. Further, the scripts $T_1, ..., T_{12}$, $B_3$, ..., $B_9$ can be used as construction rules. Now we need to define consistency rules. Such rules should be at least Turing computable. Thus, we can define consistency rules by grammars. It has been our aim to find such construction rules for which the consistency rules can be defined by context-free grammars. For this reason, we used more complex construction rules and reduced the complexity of consistency rules.

We demonstrate now that classical design strategies use context-free production systems.

The classical top-down strategy has the following production system.

| $s \rightarrow B_1 t_1$ | Generate start entity types. |
| $t_1 \rightarrow T_1 t_2 \mid T_4 t_1 \mid T_7 t_1$ | Reification of starting entity types. |
| $t_2 \rightarrow \varepsilon \mid T_1 t_2 \mid T_2 t_2 \mid T_3 t_2 \mid T_4 t_2 \mid T_5 t_2 \mid T_7 t_2 \mid T_8 t_2 \mid t_3$ | Reification of entity and relationship types. |
| $t_3 \rightarrow T_1 t_3 \mid T_1 t_4$ | Extending entity and relationship types by attributes. |
| $t_4 \rightarrow \varepsilon \mid T_6 t_4 \mid T_9 t_4 \mid T_{10} t_4 \mid T_{11} t_4$ | Reification of attributes. |

Obvious, this production system is regular. We can derive very simple control systems based on abstract automata.

Based on general ideas on strategies, different modifications can be easily generated and supported by RADD. For example, the modified bottom-up design can be modeled by the following production system:

| $s \rightarrow b_6 b_4$ |
| $b_6 \rightarrow b_6 b_6 \mid B_1 b_7$ | Generate entities. |
| $b_7 \rightarrow T_{12} b_7$ | Extend entities by atomar attributes. |
| $b_8 \rightarrow \varepsilon \mid B_5 b_8 \mid B_8 b_8 \mid b_4$ | Generation of complex attributes, generalization of attributes. |
| $b_4 \rightarrow b_4 b_4 \mid B_3 \mid B_7 \mid B_7 b_5 \mid B_7 b_7$ | Generalization of entities, generating relationship types. |
| $b_5 \rightarrow B_4 \mid B_4 b_5$ | Generalizing relationship types. |

Based on simplicity of the production systems we can now develop an approach to consistency management. Design steps have an impact on applicability of other design steps. Further, some design actions should be blocked until certain adjustments have been performed. From the other side, each design step should be closed. We use two approaches from theory of formal languages [1]:

**Attribute grammars**: Attributes are attached to non-terminals. Their values can be changed by application of other design steps.

**Controlled grammars**: The order of application of rules can be restricted by another grammar. Check points can be enforced by such grammars. Further, mixed strategies which are controlled by a skeleton can be directly expressed by controlled grammars.

Notice that primitives are defined for schemes and for subschemes. In this case modular design is supported as well as design of complete schemes.

Inside-out strategies are modelled by additional controllers. We introduce a function $next$ which can be dynamically changed.
Modular Design by Units

**4.0.1 Design by Units - Structural Design**

In [28, 22] another new recursive design methodology was developed: **Design-by-units**. It promotes extensibility, reuse and inheritance as well as behavioral extension. To some extend, this approach is similar to modular design known in software engineering. The orientation is different only. We are first interested in the data representation part and then in the processing part since a part of the processing part is based on generic operations which are defined according to the structure. However, if we consider modularization, parametrization and inheritance to be the kernel concepts of object-oriented design then this design approach can be considered to be completely object-oriented.

This methodology is not following the classical waterfall model with iterations but rather supporting a high level inside-out-strategy [7]. Experience in utilization of (DB2) has shown that this methodology was the most often chosen for practical design.

**Algorithm Design-by-units**

1. **Basic step.**
   Design the types for the independent kernel object classes.

2. **Recursion step.**
   Repeat until the schema is not changed.
   Eitheroreification:
   - Refine the units introducing subtypes (in HERM represented by unary relationship types).
   - Refine the relationship types according to the subtypes.
   or generalization of units:
   - If there are associations among units then introduce a new unit containing the associated units and the relationship type according to the association.
   - Add other relationship types if there exist new associations.

**4.0.2 Design by Units - Process Design**

Data design and process design can not be separated from each another. We need the process information as well as the structural information. For this reason the process design and the structural design need to be integrated. We use a dataflow approach [7]. One of the difficult tasks in processing modeling is to evaluate whether the designed data structures are appropriate for an effective processing of data. It is known already that relational normalization can contradict effective processing. Sometimes unnormalized relations can be used simpler. For handling this we use a cost model for processes which is based on models of complexity for operations and on priority functions for queries and transactions.

**4.0.3 Modular Object-Oriented Design Based on Top-Down-Direcitions**

Modular object-oriented design is based on a database model which incorporates structural, semantical, and behavioral model. The model developed and used in [19] could be used as one of such. The designer specifies the types, classes and methods of the given application. Similar to the previous strategy, we assume that kernel types and classes are specified. Then the specified scheme is refined by adding information on the database and changing the structure of the database or refining types in the schema. This change can be considered [19] as reification of the scheme and its types. This approach allows the development of generic class libraries for application areas. The types in the library are general enough to be used again after reification according to the specific applications of the given application area. This approach quickens the development of schemes. For modular development of schemes, we specify the type, its interface, its semantics and meaning and its behavior within the database. The reification follows that specification. Therefore we can distinguish the following categories of reification:
Reification by Specialization. Refinement by specialization introduces new subclasses and reorganizes associated relationships. Moreover, it may involve to replace a structure expression such that the new type will be a subtype of the old type and the new implicit and explicit constraints will imply the old ones.

Reification by Splitting. Refinement by splitting also leads to new types. But their type corresponds to parts of an existing type which in turn is replaced by references.

Reification by Completion. Refinement by completion means the definition of new types, constraints, methods, and behavioral constraints.

Reification by Instantiation. The schema may contain undefined parameters, attributes, and types. Refinement by instantiation provides definition for those concepts. The same applies to class parameters. Reification by instantiation may introduce new parameters as well.

5 Sample Application

6 Conclusion

The system RADD [6] which is currently developed is intended to become a toolbox for the interactive, object-oriented design of databases. The solution of RADD is to build a toolbox that is configurable with respect to a variety of aspects, to different database design strategies and to different users.

This paper discusses the strategy support of the design system RADD. RADD is a third generation tool which enables the designer to perform the design of structure, operations, views and semantics of applications and to translate the schema to logical schemata. Further, the schema can be optimized according to the characteristics of the chosen platform.

Designers use different design strategies depending on their skills and abilities, preferences and on the application. A design supporting system needs to know different strategies. RADD uses a generation approach for design strategies. According to the needs of the designer and the requirements of the application different design strategies can be generated. The generation of new strategies is based on a thorough analysis of the design process. There are three dimensions of the design process: directions, control and modularity. Thus, we obtain twelve different strategies which can be used. Further, designers often switch from one strategy to another one. Therefore, design support requires management of design steps which are based on primitive operations.

We propose an extension of the primitive operations of [7] for extended entity relationship models. These operations have an explicit embedding mechanism for replacement in graphs. Such embeddings exist implicitly in [7]. They must be made explicit for utilization in CASE environments. Based on primitives construction rules can be developed. Consistency rules maintain consistency during the design process. Control of design processes is based on production systems. On this basis different design strategies can be derived and effectively supported. Design-by-units is an example of a derived and supported strategy.

References

Towards Theory-Based Database Design Strategies

Bernhard Thalheim
Computer Science Department, Rostock University
Albert-Einstein-Str. 21, Post Box 999, D-1800 Rostock
thalheim @ informatik.uni-rostock.dhp.de

Abstract

Database design methodologies and tools should facilitate database modeling, effectively support database processing, database redesign and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970's, various tools for database design have been introduced. Most of them, however, are dependent on the knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The systems (DB2 and RAD developed in our groups do not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The Entity-Relationship Model is extended to the Higher-order Entity-Relationship Model (HERM) by adding structural constructs and using integrity constraints and operations. Different database design methodologies are developed based on the HERM approach. This paper surveys the strategy support of the design systems and proposes the design methodology Design-By-Units.

1 Database Design

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

• to meet all the information (content) requirements of the entire spectrum of users in the given application area;
• to provide a "natural" and easy-to-understand structuring of the information content;
• to conserve the whole semantic information of the designers for a later redesign;
• to achieve all the processing requirements and achieve a high degree of efficiency of processing;
• to achieve the logical independence for query and transaction formulation on this level.

The most important issue of database design is a high-quality schema within the restriction of the model and the class of DBMS or more general within the restrictions of the modeling paradigms. Low-quality schemes are hard to use, to maintain, and to adjust and tend to corruption.

A schema is called high quality or conceptually adequate if:

1. It describes the concepts of its application naturally.
2. The schema contains no or very little or necessary redundancy.
3. The schema does not impose implementational restrictions.

4. The schema covers as many integrity constraints as necessary for an efficient implementation on different platforms.

5. The scheme is flexible according to later changes.

6. The concept is conceptually-minimal.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process. A formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

In database modeling three different perspectives can be identified. Different models stress only some of those. The structure-oriented, semantic perspective focuses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The process-oriented perspective is concerned with the processes or activities performed in the application area. The behavior-oriented perspective is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The following strategy supports this approach and is an extension of [AbH84],[Che83],[NiH89],[Ris88].

There are several additional requirements to support for database design, e.g.,

- Database design tools need to be adaptable to a wide range of designers. Personal preferences and past experience can make one representation better than another, given an application and a team of people to built it.

- Database design could be viewed as a special knowledge representation process. It should make processing easy and should map onto problems that we know how to deal with.

- The reusability of design decisions is another important feature of modern design tools. Reusable development information includes not only generic code components but also generic system specification like generic schema components and generic requirements.

- A database changes also its structure over its lifecycle. A database design tool should actively support this restructuring. This can be based on a theory of equivalent schemata or subschemata. Redesign is closely related to reverse engineering.

- Database design is based on one or more data models. A large number of conceptual data models have been proposed. While most models have been proposed primarily for stand-alone database management systems and are adapted to implementational restrictions in database systems, it is increasingly apparent that data models will be from one hand side incorporated directly into programming languages and a variety of tools and from the other hand side have to be extended to interoperating environments and multisystem- and multimodel-paradigms.

Based on an analysis of systems supporting database design we have developed a database design system (DB)² [Tha92⁺] which is supporting a high-level efficient database design.

The system is using an extension of the entity-relationship model for the structural design and the graphical interface. Further, a high-level language for specification of integrity constraints and operations has been developed. Constraint declarations include: attribute data types, non-null attributes, attribute
combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated. The design can be translated to different DBMS according to characteristics of the DBMS and can be used for the generation of an interface builder if the design is to be translated to a multi-paradigm and multi-system environment.

At present, we are developing another tool RAD in a joint project in different groups [BOT90]. The system is based on the experience of four year of extensive utilization of (DB)² in more than 100 different user groups. It has a better user support system especially for the more difficult design steps like design of integrity constraints, refinement and rebuilding the schema.

The two systems are using the known and developed database theory in almost every step [EMN89], [Heu89], [Hul89], [NiH89], [Ris88], [Tha91], [Tha91’], [Tha92”]: theory of data models, theory of normalization of relational databases, theory of scheme equivalence. The theory has been extended in order to support bad schemes and manipulation of schemes, to cope with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user, and to deal withh consistency if the same type of information appears in several design phases.

Last but not least, the database design strategies had to be adapted to the model and the purposes of database design processes. This paper will report these results. For an automatic adaption of the design system to the chosen user strategy, the known database design strategies have been generalized. Further, we have developed our own database design strategy using approaches known in computer science.

Object-oriented design techniques provide a basic support to the construction of generic specifications. Modular design and abstraction promote the design of generic classes (later called units) through the specification of the external behavior of units while hiding the internal details. Our modular design methodology addresses the reuse problem by using a high-level abstract object-oriented approach and viewing the database design process as two complementary activities:

- Construction of generic development information to be reused in the development of similar applications in a particular domain.

- Construction of applications with specific requirements by reusing the available generic development information for the application domain.

The paper is divided into five parts. In section 2, the database design process is to be briefly discussed. Properties of database design are presented in section 3. Different classical design strategies are to be reviewed in section 4. Further, we introduce the design by units design process which is a unifies modular and object-oriented database design in section 5. Due to the space limitations, the paper presents a partial example on database design in the appendix. The paper summarizes the experience of intensive utilization of the design tool (DB)² for more than four years by almost 100 different user groups. One outcome of the project is presented in the paper: The theoretically based design strategy.

2 Components of the Database Design Process

The database design process can be described according to the information units which are necessary:

- The requirement description is an informal description of the application and the goals (mission) to be achieved.

- The interface is the description of the organizational structure and the distribution of the database.

- The structure of the database objects is described by different classes of objects and their typing systems.
The semantics describes the static properties of objects and object classes. Thus, the semantics is used for restricting the set of possible databases.

- The views describes the different subschemata for different parts of the application.

- The operations describe how the database should operate.

- Based on the semantical, structural and view information cooperating views can be defined for the different sub-databases.

- Using abstraction query forms describe in general the database information retrieval processes for the user.

- The behavior specifies the dynamic constraints of a database.

- The transactions are used for the specification of database programs.

- Finally, the database modul collects and re-organizes the information.

These units are not independent. The description of views depends from the structural, semantical and interface information. The complete dependency graph is displayed in Figure 1. Based on these information units different design decisions can be generated.

![Dependency Graph](image)

Figure 1: The Information Acquisition in Database Design

The database design activities can be differently organized by different methodologies in accordance to the information systems life cycle. For instance, we can use the commonsense knowledge on platforms first for feasibility studies. Then requirements can be collected, analysed, structured and grouped. Based on this information the design is performed. Further, a prototype is developed. The prototype allows the user to verify that the information system satisfies their needs. This can lead to an additional requirement collection and analysis. If the prototype satisfies the requirements the implementation of the design is performed. Implementation is concerned with the programming of the final, operating version. During the validation and testing process the developer assures that phase of the process is of acceptable quality and is an accurate transformation from the previous phase. Operation starts with the initial loading
of data and terminates when the system eventually becomes obsolete and has to be replaced. During operation, maintenance is required for adopting the system to other requirements (conditions, functions, structures) and for correcting errors that were not detected during validation.

The design process is mainly concerned with the requirement acquisition, requirement analysis and conceptual design. We can distinguish different phases in database design. Based on the waterfall and/or fountain models of software development the classical scenario for structure-driven database design distinguishes between conceptual, logical and physical design. During conceptual design a high-level description of the database (structure, semantics, operations, distribution) is developed using the data and operations requirements. This description is independent from the platforms or the database management systems which are later used. Conceptual design results in a conceptual schema which is kept, for instance, for later database maintenance. During logical design a conceptual schema is translated to a logical schema which describes the structure, operations, semantics and distribution of the database according to a class of database management systems (or according to a single system or, in general, according to a set of paradigms). The logical schema is transformed into the physical schema during physical design. The storage structures, access methods, the organization in secondary memory is developed for a specific DBMS.

Alternatively, the processing-driven database design starts with the processing analysis of requirements which generates a functional schema. This is used during high-level application design for the generation of the application specification which describes the behavior of the database in a high-level abstraction. These specifications are transformed into a detailed program specification during the application program design phase.

Modern, more recent approaches try to unify and to combine both the structure-driven and the processing-driven database design.

3 On Database Design Methodologies

Database design is implicitly based on some assumptions:

- A database must satisfy a limited set of basic requirements used also in software engineering (correctness, time performance, space performance, compatibility with organizational constraints) and special database requirements (availability, reliability, compliance with a data manager, compatibility with semantical constraints). Satisfying each of these problems constitutes a special identifiable design problem.

- The solution of the above mentioned problems is the objective of the design process. Each step of this process produces a piece of specification.

- Independent from the used methodology a database design process requires the solution of these problems. Therefore, the design process consists in carrying out a limited set of processes.

- Each design process is based on specific concepts, techniques and reasonings.

- Each step of the database design process can be expressed as a transformation (adding, removing, changing the specification).

Therefore, for solving the general design task we a theory-based generic model of database design steps. This model can be used also for re-engineering of existing databases. In this case, it suggests to search the description of the database for traces of each specific design process.

A design methodology could be based on monotone reasoning. Each design step is adding certain information. During design the information already modeled is not changed. This does not mean that structures, operations and constraints are not changed. However, the new design step does not abandon already obtained information. Each design step needs to satisfy certain restrictions:
Each design step is content-preserving, i.e. there exists a query \( q \) such that for each database \( db' \in SAT(S') \) : \( q(db') \in SAT(S) \) for \( S, S' \) and \( Step(S) = S' \).

Each design step is constraint-preserving, i.e. \( \Sigma' \models \Sigma \) for \( S = (Struc, Ops, \Sigma), S' = (Struc', Ops', \Sigma') \) and \( Step(S) = S' \).

Each step is minimally content-preserving, i.e. the design step does not add derived types; there does not exist a query \( q \) such that for each database \( db' \in SAT(S') \) there exists a database \( db \in SAT(S) \) such that \( q(db) = db' \) for \( S, S' \) and \( Step(S) = S' \).

Furthermore, there are some desirable restrictions:

Each step is minimally constraint-preserving, i.e. \( \Sigma \models \Sigma' \mid_{Struc} \) for \( S = (Struc, Ops, \Sigma), S' = (Struc', Ops', \Sigma') \) and \( Step(S) = S' \).

Each step is nearly free of path tuple-generating constraints. When a path tuple-generating constraint is present it often requires that the path has to be computed to check whether the constraint holds in the database.

Notice, that inclusion constraints can be maintained in a database using only projections. Path inclusion constraints are still based on joins. The last restriction is often too strong. It can be replaced by the following restriction. This restriction requires an effective management of basic database operations.

Each step is update simplicity-preserving, i.e. if for \( S \) the complete set of basic update operations is defined without join operations then \( Step(S) \) has this property too.

A schema is lexical-minimal if it uses as few lexical units as possible. For instance, the attribute DepartmentName of the type Department is can be replaced by Name (the corresponding identifier is Department.Name).

It should be mentioned that several known normalization algorithms do not fulfill the above requirements. However, there are 4NF algorithms which preserve all properties.

A methodology needs also rules in which case which concept is going to be applied. One rule which could be considered as the basic rule is that independent objects are to be represented by entity sets. Several approaches, especially those based on the original entity-relationship model, state that attribute values can be also represented by additional, artificial entity types, e.g. choosing an entity type representation when we understand that several properties (attributes, relationships, generalizations, subsets, ..) can be associated with the object to be modeled. We represent any property only by attributes. This strictness can be dropped if types are decomposed according to same attribute value (e.g. the attribute sex of a person is used for the horizontal decomposition of the entity set Person). This decomposition is used when we expect that (now or later) some property will be associated to the lower level entity types. Nested attributes should be chosen when it is natural to assign a name to it. Each attribute (nested or atomic) of a type should be as independent as possible from other attributes.

4 Classical Design Methodologies

In software design and in database design we distinguish different design perspectives.

1. The structure-oriented perspective focuses on the structural description of the data. Almost all known database design approaches are based on the structure-oriented perspective. Sometimes, the structure-oriented perspective is unified with the semantical perspective. In this cases, the design of the structure is to be combined with the design of static integrity constraints. The different strategies based on the structure-oriented perspective are shown in Figure 2.

2. The behavior-oriented perspective (called in [LiB90] integrity-centered; see also [Lip89]) is concerned with the behavior of the database during its lifetime. It can be based on event approaches [Lip89] or on Petri-net approaches [Obe88, Obe90] and predicate transition systems.
3. The process-oriented perspective is concentrated on the database operating. The structure-oriented perspective is discussed in a large variety of papers. The process-oriented perspective can reuse approaches known in software engineering. The behavior-oriented perspective is a high-level descriptive approach to an integrated specification of structures and operations. Database design is considered to be a transformation process carried out on integrity constraints. Database behavior is specified by stepwise reification. The structure-oriented approach is currently well developed for relational models and was directly generalized to semantical models like entity-relationship models. It is, however, possible to develop for semantical models more specific methodologies which are using the specific properties of these models. One specific methodology is discussed in the next subsection. Generally, a methodology consists of a strategy, a set of primitives to be used and guidelines on the application of each strategy step and each primitive. A methodology should be, from one hand side, rigorous, i.e. suggesting a strategy which is to be used, and, from the other hand side, flexible, i.e. applicable to a variety of applications. These requirements are contradictory in general. A design tool should be flexible enough to be adaptable to different design styles and should be rigorous enough to be well-based. The building block of a methodology is the design strategy. Known strategies are, for example, the bottom-up and top-down approach. Most of the known strategies are considering only the structural part of a schema. The strategies are based on different primitives. Most of these primitives can be considered to be generalizations of the known algebraic operations of the relational model.

4.1 Design Primitives

In [BCN92] design primitives were introduced. Although this approach is cleaning up the various proposals on primitive design steps it is not continuous. It is claimed in [BCN92], for instance, that top-down design strategies are incomplete. This is mainly due to the used system. Their systems are incomplete also for other design strategies for several reasons:

1. Only binary entity-relationship diagrams can be generated.
2. It is possible to generate subtypes of entity types keeping the entity type and adding the relationship types between the original entity type and the subtypes. However, this operation is not defined for relationship types.

Another problem in the approach of [BCN92] is the graphical representation used for subtypes. We distinguish between structural abstraction and graphical abstraction. The last one is understood as a kind of representational abstraction. In this case, we define a mapping from the set of possible schemata to the set of graphs. This mapping is called injective if each graph image corresponds only to one preimage. The mapping is called basic if it is defined as the extension of the mapping from primitive types (entity, relationship and attribute types) to primitive graphical units (rectangles, diamonds with arrows, points with lines; may be with names). The structure is context-sensitive because the corresponding entity types must exist for component types of relationship types. However, the context sensitivity follows the order of the types. Therefore, the mechanism for defining a structure are not equal-order-context-sensitive. The mapping mechanism for the presented extended ER-model is injective and basic. The mapping mechanism used in most ER books is injective but not basic. The reason for this is that subtypes and their supertype need to be mapped together. If the mapping mechanism is not basic then during database design the design of the database structure needs to be combined with the graphical representation. Since in such design languages the graphical structure is on the equal-order-context-sensitive the graphical structure can change strongly for even very simple changes. Tools based on such approaches are more complex.

Further, since database design can be considered to be database management on a schema database we can compare the design primitives with operations in the design database system. In the nested relational model different operations are used:
projection, selection, partition, join, \( \theta \)-join or the join via a coercion function, union, nest, and unnest. Additionally, concepts can be added to the schema.

The operations can be seen also as decomposition/composition pairs:

- selection / union,
- partition / union,
- nest / unnest,
- projection / product (or join).

We have mentioned already that each database model needs its own primitives according to the constructors used in the definition of types. In HERM we are using the type constructors \texttt{product} (generating tuple-sets) and \texttt{union} (generating clusters or generalizations of subtypes). Therefore, for a complete set of design primitives we need only primitives defining the generation of products (i.e. product and concatenation to components (The last can be expressed via the first operation. However it might be sometimes more convenient to have this operation too.)) and the generation of union types. Further, we need operations for initializing the design (in our case, generating basic attribute types and generating basic entity types but not for the generation of relationship types). We define now these general design operations. For that we assume a set \( C \) of type constructors. Using this set we can define a set of terms and subterms for each term. For instance, if the \( C \) consists of \texttt{union} \( \cup \) and \texttt{product} \( \times \) then the set \( T \) of terms is defined by the consecutive application of these operators and the subterms are defined by selection and projection operators. Further, we assume \( \perp \in T \) for the empty term.

**Composition.** Given the constructor \( C \), a new name \( t \) and types \( t_1, \ldots, t_n \). The composition \( \text{compose}(t_1, \ldots, t_n, C, t) \) defines a new type \( t \) using the constructor \( C \).

**Decomposition.** Given a type \( t \) and terms \( e_0, e_1, \ldots, e_n \) and (possibly empty) names \( N_0, \ldots, N_n \). The decomposition \( \text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n) \) defines new types which are generated by \( e_i \) and are labeled by \( N_i \).

**Extension.** Given a type \( t = (t_1, \ldots, t_m, \{A_1, \ldots, A_n\}) \) of order \( i \) \( (m \geq 0 \ (m = 0 \ for \ entity \ types), \ n \geq 0) \), an attribute \( A' \) and a type \( t' \) of order \( j \), \( j < i \) then

- \( \text{extend}(t, A) \) is the type obtained from \( t \) adding \( A' \) to the attribute set and
- \( \text{extend}(t, t') \) is the type obtained from \( t \) adding \( t' \) to the component sequence.

For our purposes entity types are types of order 0.

**Initialization.** \( \text{generate}(E) \), \( \text{generate}(A) \) generate an entity type labeled by \( E \) without attributes and an attribute type labeled by \( A \).

### 4.2 The Top-Down Strategy in Database Structure Design

The pure top-down strategy is based on refinement of abstract concepts into more concrete ones. Each step introduce new details in the schema. The design process ends when all structural requirements are represented. According to the above discussed taxonomy of database design operations the following primitive operations can be used during top-down design:

\[
\text{compose}(t_1, \ldots, t_n, C, t) \\
\text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n) \\
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \\
\text{generate}(E), \text{generate}(A)
\]

The advantage of top-down design is the independent consideration of each type during a design step. The designer can analyse one concept ignoring the other concepts. However, top-down design requires
from the designer the knowledge on the general picture of the design first. Therefore, complete top-down
design can be performed only by experienced designers if the application area is not well-structured.

In [BCN92] a set of primitives is defined (We use here a slight generalization and omit $T_7$ since it can be
represented by $T_6, T_8$):

$T_1$ \textit{decompose}($E, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n$) where $e_1, ..., e_n$ are projection terms, and $e_0 = e_1 \times ... \times e_n \times e_{01}$.
The result of the application of the primitive are new entity types labeled by $N_1, ..., N_n$ and a $n$-ary
relationship type $N_0$ on $N_1, ..., N_n$.

$T_2$ \textit{decompose}($E, e_0, e_1, e_2, ..., e_n, E, N_1, ..., N_n$) where $e_1, ..., e_n$ are selection terms and $e_0 = id$.
The primitive generates $n$ subtypes of the entity type $E$.

$T_3$ \textit{decompose}($E, e_1, ..., e_n, \bot, N_1, ..., N_n$) where $e_1, ..., e_n$ are selection terms.
The primitive generates $n$ subtypes of the entity type $E$ and removes $E$.

$T_4$ \textit{decompose}($R, e_1, ..., e_n, \bot, N_1, ..., N_n$) where $e_1, ..., e_n$ are selection terms and $R$ is an $m$-ary
relationship type.
The primitive generates $n$ $m$-ary relationship types and removes $R$.

$T_5$ \textit{decompose}($R, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n$) where $e_0, e_1, ..., e_n$ are projection terms and $e_i = e_{i1} \times e_0$
for $1 \leq i \leq n$.
The primitive generates an entity type labeled $N_0$ and $n$ new relationship types connected to $N_0$
and $e_i(R)$.

$T_6$ \textit{extend}($R, A$), \textit{extend}($E, A$) add additional attributes to relationship and entity types.

$T_8$ \textit{decompose}($A, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n$) where $e_1, ..., e_n$ are atomar terms and $e_0 = e_1 \times ... \times e_n$.
The primitive generates nested attributes via a tuple constructor.

It is easy to see that $T_5$ can be replaced by the double application $T_1; T_1$ of the first primitive. It is
obvious, that the set $\{T_1, T_2, T_3, T_4, T_5, T_6, T_8\}$ is minimal and the primitives are independent. $T_1$ generates
relationship types, $T_2$ generates generalizations, $T_3$ generates disconnected schemes, $T_4$ is needed to generate
cycles, $T_6$ is used for the generation of attributes, and $T_8$ generates nested attributes.
Notice further that $T_2$ has no counterpart for relationships. $T_4$ is the counterpart for $T_3$.

For the HERM design the primitives should be extended. For instance, the primitive $T_5$ can be extended
to the generation of higher-order types:

$T_9$ \textit{decompose}($R, e_0, e_1, ..., e_n, N_0, N_1, ..., N_n$) where $e_0, e_1, ..., e_n$ are projection terms and $e_0 = e_{01} \times e_1 \times ... \times e_n$.
The primitive $n$ new relationship types on $e_i(R)$ and generates a relationship type labeled $N_0$ on the new
relationship types.

This primitive is the relationship counterpart of $T_1$. We can also develop the counterpart of $T_2$.

$T_{10}$ \textit{decompose}($R, e_0, e_1, e_2, ..., e_n, E, N_1, ..., N_n$) where $e_1, ..., e_n$ are selection terms and $e_0 = id$.
The primitive generates $n$ subtypes of the relationship type $R$.
The graphical representation in HERM uses unary relationship types for $N_i$. In the classical entity-
relationship model this primitive is widely applied by copying the original relationship type $n$ times and
using additional inclusion constraints. The HERM representation is simpler.

In [BCN92], an example is given for the incompleteness of the proposed operations $\{T_1, ..., T_8\}$. It is
demonstrated that the schema with the entity types $City$, $Division$, $Army$, $State$ and the relationship
types LocatedIn = ( Division, City, θ), HeadQuartersOf = ( City, Army, θ), ProtectedBy = ( State, Army, θ), HiresFrom = ( Division, State, θ), Of = ( Division, Army, θ), In = ( City, State, θ) can not be generated. This is true due to the following more general fact.

**Proposition 1** Using \{T_1, ..., T_8\} only planar entity-relationship schemata can be generated.

### 4.3 The Bottom-Up Strategy Database Structure Design

Bottom-up design was first developed for the relational model. During bottom-up design each step introduces new concepts not considered before or modify existing ones. During each step the designer checks whether all features of requirements are covered. The designer start from elementary concepts and builds more complex out of them. The advantage of the bottom-up approach is its simplicity. The main disadvantage of the bottom-up approach is that this approach requires restructuring. Therefore, the strategy is not monotone.

The following primitive operations can be used during top-down design:

\[
\begin{align*}
&\text{compose}(t_1, \ldots, t_n, C, t) \\
&\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \\
&\text{generate}(E), \text{generate}(A)
\end{align*}
\]

In [BCN92] the following primitives are defined (We use slight generalizations of the introduced primitives.):

- **B_1** \text{generate}(E) is used to generate a new entity type.
- **B_2** \text{compose}(E_1, \ldots, E_n, \times, R) generates a new relationship type between the entity types.
- **B_3** \text{compose}(E_1, \ldots, E_n, \cup, E) generates a new entity type which is the generalization of the previously given entity types.
- **B_4** \text{(generate}(A); \text{extend}(R, A)), \text{(generate}(A); \text{extend}(E, A)) generate a new attribute and add it to an entity or relationship type.
- **B_5** \text{compose}(A_1, \ldots, A_n, \times, A); \text{extend}(t, A) creates a nested attributes and adds it to a type.

Primitive **B_3** needs also to check which attributes are to be shifted to the generalization. Bottom-up primitives are minimal and complete for the original entity-relationship model.

### 4.4 The Inside-Out Strategy in Database Structure Design

The inside-out strategy restricts the bottom-up approach by controlling the order of primitives application. This strategy is still complete. We choose the most important concept first, design it and then proceed by moving as an oil stain does, designing first concepts that are conceptually closed to the already design. The order of refinements is disciplined. The designer navigates from the central type to the more distant ones. It is easy to discover new concepts which are to be designed next. However we loose abstraction capabilities. The global schema is built at the end only.

### 4.5 The Mixed Strategy in Database Structure Design

Another controlled approach is the mixed approach. This approach mixes the top-down and the bottom-up approach. First, a skeleton schema is to be designed (using one of the previous approaches). This schema represents the main classes (or units) and their main associations. Then each of the units is refined and later integrated with other designed units. Using the skeleton schema the bottom-up integration of
concepts is simpler. Since the complete requirements set is now partitioned the design of each unit is less complex. The success of the strategy depends from the design of the skeleton schema. Therefore, this method is applicable if the application is already well-recognized.

5 Design by Units

5.1 Design by Units - Structural Design

In [YaT89, Tha91', Tha89] another new design methodology was developed: Design-by-units. Most of the well-known design methodologies think as in the relational approach. But each of the database models should have its own methodology. It is surprising that most of the proposed models do not have its own design methodology. If the model is getting richer in construct the methodology should be deepen. One of the database models with its own methodology is the ER model. However, there is still a little agreement in which cases objects from the real world should be modelled by attributes, entities or relationships. A part of the problems of view integration is caused by this modelling problem. And this contradicts the belief of experienced database designers. Those assume that the views of an enterprise can be integrated since there is an internal integration in the enterprise. The reason for this mismatch is that methodologies are not supporting abstraction in an efficient manner. The new design methodology can be understood as a step towards a well-integrated methodology. The proposed methodology is truly object-oriented and at the same time also theoretically based what supports the implementability. This methodology support also extensibility since using this methodology an existing design and implementation can be extended without introducing changes to it. It promotes reuse and inheritance as well as behavioral extension. To some extend, this approach is similar to modular design known in software engineering. The orientation is different only. We are first interested in the data representation part and then in the processing part since a part of the processing part is based on generic operations which are defined according to the structure. However, if we consider modularization, parametrization and inheritance to be the kernel concepts of object-oriented design then this design approach can be considered to be completely object-oriented.

This approach has further other advantages: it is easier to detect similarities among design units and to reuse parts of design units in other units; changes to the scheme and to parts of units are directly reflected in all other units which are using the changed. The new methodology supports directly the above discussed distinction between kernel and dependent object. This is especially useful, if abbreviation techniques [Sci91] are used in query forms [YaT89]. It is a recursive method. Each step is based on the above discussed eight design steps. This methodology is not following the classical waterfall model with iterations but rather supporting a high level inside-out-strategy [BCN92]. Experience in utilization of (DB2) has shown that this methodology was the most often choosen for practical design.

Design-by-units

1. basic step.
   Design the types for the independent kernel object classes.

2. Recursion step.
   Repeat until the schema is not changed.
   Either reification:
   - Refine the units introducing subtypes (in HERM represented by unary relationship types).
   - Refine the relationship types according to the subtypes.
   or generalization of units:
   - If there are associations among units then introduce a new unit containing the associated units and the relationship type according to the association.
• Add other relationship types if there exist new associations.

![Structure-Oriented Design Strategies](image)

Figure 2: Structure-Oriented Design Strategies

Therefore, the proposed strategy could be understood as a controlled, second-level strategy in Figure 2.

5.2 Design by Units - Process Design

As already discussed in the previous section, the data design and the process design can not be separated from each another. We need the process information as well as the structural information. For this reason the process design and the structural design need to be integrated. We use a dataflow approach [BCN92]. A process is an activity of the information system. The dataflow is an exchange of information between processes. The processes use information from the database and create temporary databases necessary for the process. Since processes use different databases and these databases are usually not fully integrated, interfaces are to be used for view cooperation. Generally, the interface is the description of the cooperation between different users (originator/receiver of the dataflow).

The processing_requirement modeling at the data level has several applications:

1. Completeness and consistency checking of information requirements: One requirement in conceptual database design is that the information requirements of a user group are complete and semantically consistent (not conflicting) with the processing requirements. A process model can be used to verify the completeness and consistency of other requirements.
2. **Identification of proper data entries:** Processing requirements can add different temporal and persistent database structures to the schema. In this case the view integration is more complex.

One of the difficult tasks in processing modeling is to evaluate whether the designed data structures are appropriate for an effective processing of data. It is known already that relational normalization can contradict effective processing. Sometimes unnormalized relations can be used simpler. For handling this we need a cost model for processes. The cost model can be based on models of complexity for operations and on priority functions for queries and transactions.

Therefore, we need a representational and a graphical language for the representation of processes. We decided to use three different languages, one high-level abstract language which is directly supported by the modular design-by-units-strategy, another one for a more detailed procedural description and another one more dataflow oriented which shows the level of the database directly. For that we use notions known from dataflow diagrams which are represented in figure 3.

Processing requirements can be modeled at the data, query, transaction and program levels.

Since the database structure is defined by a graph queries (the user’s interaction with the database) can be defined by a traversal of the graph or by a graph generated from the schema. We can use a set of anticipated queries to determine the schema with the query optimal behavior. Each query is defined by a graph, its frequency of use and its priority. Furthermore, we can use additional (cardinality) values for estimation of the queries volume like the

- the estimated number $N$ of occurrences associated with a type,
- the average number $P$ of associated elements for a given element,
- the number $K$ of distinct values associated with an attribute and
- the number $H$ of types in the schema.

Using these numbers, one can determine which of the alternatives has a better behavior. For that, each edge in the query graph defines a workload (e.g. the frequency of the traversal via this edge multiplied by a schema value depending on $N, P, K, H$). This is used to determine the workload of a query. Then the alternatives define the workload for all queries and can be compared.

A transaction is a (nested) sequence of database operations which transforms consistent databases into consistent ones. They can be described like queries. However, modeling of transactions provides additional information about the relationship and the order of database operations. This information can be used for selecting alternative HERM designs and for the translation to the most efficient logical schema. In the same manner, an application program can be viewed as a structure of queries and/or transactions that manipulate a database or several databases.
6 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed a theoretical basis for different database design methodologies together with an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system \((DB)^2\) was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and systems have the following advantages:

1. The model has a strong theoretical basis [Tha92].
2. The modeling is more natural and can be applied in a simple manner. Only necessary facts are to be expressed [Tha90, YaT89].
3. The theory is applicable to practical needs [SST91, Tha89, Tha91', YaT89].
4. The results of the design are much simpler than in other approaches [YaT89, Tha92'].
5. The model is easy understandable, simple and perceivable [Tha91'] and can be used as a basis for design tools [BOT90, Tha89, Tha92].

This paper aims to demonstrate how a well founded theory can be used for the development of a concise, understandable and useful approach to the database design. Based on this theory, design strategies can be developed and supported by design systems.

References

[LiB90] U.W. Lipeck and S. Braß, Tools for integrity-centered design of database applications. Internal report of the Computer Science Department, University Dortmund, 1990 (in German).
REFERENCES


[Tha92"'] B. Thalheim, Design with the database design system (DB)$^2$. In Fourth Int. Conference "Putting into practice methods and tools for information system design" (ed. H. Habrias), Nantes, France, 1992, 155 – 174


Appendix
A Structural Design by Units

For the example below we use the extended entity-relationship model HERM. We can specify complex attributes, clusters and also relationship types over relationship types. The last extension makes it possible to represent IsA-relationships directly by unary relationship types what is simplifying the design.

Let us illustrate (the structural part) of the design-by-units-method using the following medical example. We omit the semantical part. The database covers the following information:

- known diseases (with a list of possible symptoms (estimation of the probability (whenever a patient has the disease, he also has the symptom with a higher acuteness) and a list of possible factors (drugs, drug combination, other diseases) which may aggravate, cause or cure the disease);
- known symptoms (names, magnitude of their intensity/acuteness);
- known drugs with their components;
- patients (names address, date of birth, medical history (present and past illness (duration, diagnosing physicians, drugs prescribed for them) and reported symptoms (duration of the occurrences, magnitude of intensity/acuteness, persons who reported or measured the occurrence (patient himself, relatives, medical personnel);
- physicians (area of specialization) (physicians can be also patients);
- other persons (names, addresses).

First we design the basic units and then we relate these basic units like the units PERSON and DISEASE in Figures 4 and 5. Then we merge these units obtaining larger, more general units. One relationship type which is to be designed during the merge of the units PERSON, DISEASE and DRUG is the type Cure. In the last iteration the schema in Figure 6 is designed.

![Diagram](image)

Figure 4: The unit PERSON

The final result of the design by units is represented in Figure 6.
Process Design and Design by Units

Let us extend the previous design by a units Pharmacy and CityPharmacy. Then we can model the complex transaction RequestADrugFromPharmacy. If the drug is available in the departments pharmacy then the physician is informed. If not then the pharmacies of other hospital units are consulted and requested. If the drug is not available in the hospital the central pharmacy is consulted. The central pharmacy use its store or requests the supplier. The drug is then send back to the hospital with an invoice. If the drug is not available the physician is to be informed.

For modeling of processes we have developed different languages which can be considered as a seria of abstract specification languages. A process specification written in algebraic language can be transformed to the process specification in procedural language and can be represented in graphical transaction language.

The specification of the above presented example would lead to the following algebraic process specification:

\[
\text{Request} := \text{get Drug} \\
\text{from DirectPharmacy } \cup \text{ HospitalPharmacy } \cup \text{ CityPharmacy}
\]

where the exclusive ordered union is denoted by \( \cup \).

This leads to the procedural process specification

\[
\text{Request} := \text{if AvailDrugPharmacy then AvailInfo} \\
\text{else if AvailAnotherHospPharmacy then MakeListToPharmacy} \\
\text{else OrderFromCityPharmacy}
\]

The unit Pharmacy has also an active operation

\[
\text{Active MakeOrder} := \text{activation RequestPharmacy } \neq \text{ Empty} \\
\text{if DrugAvailable then ProceedRequestByPharmacy} \\
\text{else OrderFromCityPharmacy}
\]

and the transaction with the complete description

\[
\text{ProceedRequestByPharmacy}(d,\text{doc},\text{ord}) := \\
\text{Update(DrugOthPh,-d); DeleteReqPhar(\text{ord}); AvailInfo(\text{doc}).}
\]

Figure 7 shows a part of the graphical process specification. This process description is embedded into the schema graph for modeling of processing complexity.
Figure 6: HERM-Diagram of the Medical Application Database
Figure 7: Complex Transaction RequestADrugFromPharmacy
Figure 11 shows a part of the graphical process specification. This process description is embedded into the schema graph for modeling of processing complexity.
Towards a Framework for Database Design Strategies

Bernhard Thalheim
Computer Science Institute, Cottbus Technical University, D - 03013 Cottbus
full professor
thalheim @ informatik.tu-cottbus.de

Abstract
Database design methodologies and tools should facilitate database modeling, effectively support database processing, database redesign and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970's, various tools for database design have been introduced. Most of them, however, are dependent on the knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The systems (DB)$^2$ and RAD developed in our groups do not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. These tools are based on an extended entity-relationship model. The entity-relationship model is extended to the Higher-order Entity-Relationship Model (HERM) by adding structural constructs and using integrity constraints and operations. The system RAD has a component which enables the user to choose his design strategy according to his experience and abilities. Different database design methodologies are developed based on the HERM approach. This paper demonstrates how different database design strategies can be developed and supported. This paper surveys further the strategy support of the design system RAD and proposes the design methodology Design-By-Units. The extensions proposed in the paper are generalizing and applying the NIAM approach to database design, especially to entity-relationship database modeling.

1 Database Design
The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

- to meet all the information (content) requirements of the entire spectrum of users in the given application area;
- to provide a "natural” and easy-to-understand structuring of the information content;
- to conserve the whole semantic information of the designers for a later redesign;
- to achieve all the processing requirements and achieve a high degree of efficiency of processing;
- to achieve the logical independence for query and transaction formulation on this level.

The most important issue of database design is a high-quality schema within the restriction of the model and the class of DBMS or more general within the restrictions of the modeling paradigms. Low-quality schemes are hard to use, to maintain, and to adjust and tend to corruption.

A schema is called high quality or conceptually adequate if:

*This research has been supported by DFG Th 465/2.
1. It describes the concepts of its application *naturally*.

2. The schema contains no or very little or necessary *redundancy*.

3. The schema does not impose *implementational restrictions*.

4. The schema covers as many *integrity constraints* as necessary for an efficient implementation on different platforms.

5. The scheme is *flexible* according to later changes.

6. The concept is *conceptually-minimal*.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process. A formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

In database modeling three different perspectives can be identified. Different models stress only some of those. The *structure-oriented, semantic perspective* focusses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The *process-oriented perspective* is concerned with the processes or activities performed in the application area. The *behavior-oriented perspective* is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The following strategy supports this approach and is an extension of [1],[6],[13],[17].

There are several additional requirements to support for database design, e.g.,

- Database design tools need to be adaptable to a wide range of designers.
- Database design could be viewed as a special knowledge representation process.
- The reusability of design decisions is another important feature of modern design tools.
- A database changes also its structure over its lifecycle.
- Database design is based on one or more data models.

Therefore, database design methodologies have to meet the following criteria [7]:

1. The methodology is a step-by-step procedure.
2. Roles and responsibilities have to evaluated carefully.
3. The methodology distinguishes clearly between generic and product-specific design techniques.
4. The methodology supports the generation of data dictionaries.
5. The methodology should be based on database theory.
6. Checkpoints should be established throughout the design process.
7. The semantical and behavioral information should be integrated.
8. The methodology uses simple graphical representation techniques.

Beside these criterias, special design challenges could be addressed:

- If possible, access to data through views is supported.
- Data security is considered throughout the design process.
- Very large databases and rather small databases are distinguished by the design strategy.
• Database evolution is considered during the design process and during implementation.

• Database technology is developing too. The methodology anticipates the technology evolution.

Based on an analysis of systems supporting database design we have developed a database design system (DB)² [23] which is supporting a high-level efficient database design. The system is using an extension of the entity-relationship model for the structural design and the graphical interface. Further, a high-level language for specification of integrity constraints and operations has been developed. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated. The design can be translated to different DBMS according to characteristics of the DBMS and can be used for the generation of an interface builder if the design is to be translated to a multi-paradigm and multi-system environment. At present, we are developing another tool RAD in a joint project in different groups [3]. The system is based on the experience of four year of extensive utilization of (DB)² in more than 100 different user groups. It has a better user support system especially for the more difficult design steps like design of integrity constraints, refinement and rebuilding the schema.

Last but not least, the database design strategies had to be adapted to the model and the purposes of database design processes. This paper will report these results. For an automatic adaption of the design system to the chosen user strategy, the known database design strategies have been generalized. Further, we have developed our own database design strategy using approaches known in computer science.

The paper is divided into five parts. In section 2, the database design process is to be briefly discussed. Properties of database design are presented in section 3. Different classical design strategies are to be reviewed in section 4. Further, we introduce the design by units design process which is a unifies modular and object-oriented database design in section 5. Due to the space limitations, the paper presents a partial example on database design in the appendix. The paper summarizes the experience of intensive utilization of the design tool (DB)² for more than four years by almost 100 different user groups. One outcome of the project is presented in the paper: The theoretically based design strategy.

2 Components of the Database Design Process

2.1 Database Design Units

The database design process can be described according to the information units which are necessary:

• The requirement description is an informal description of the application and the goals (mission) to be achieved.

• The interface is the description of the organizational structure and the distribution of the database.

• The structure of the database objects is described by different classes of objects and their typing systems.

• The semantics describes the static properties of objects and object classes. Thus, the semantics is used for restricting the set of possible databases.

• The views describes the different subschemata for different parts of the application.

• The operations describe how the database should operate.
Based on the semantical, structural and view information, cooperating views can be defined for the different sub-databases.

Using abstraction, query forms describe in general the database information retrieval processes for the user.

The behavior specifies the dynamic constraints of a database.

The transactions are used for the specification of database programs.

Finally, the database module collects and re-organizes the information.

These units are not independent. The description of views depends from the structural, semantical and interface information. The complete dependency graph is displayed in Figure 1. Based on these information units different design decisions can be generated.

The database design activities can be differently organized by different methodologies in accordance to the information systems life cycle. For instance, we can use the commonsense knowledge on platforms first for feasibility studies. Then requirements can be collected, analysed, structured and grouped. Based on this information the design is performed. Further, a prototype is developed. The prototype allows the user to verify that the information system satisfies their needs. This can lead to an additional requirement collection and analysis. If the prototype satisfies the requirements, the implementation of the design is performed. Implementation is concerned with the programming of the final, operating version. During the validation and testing process the developer assures that phase of the process is of acceptable quality and is an accurate transformation from the previous phase. Operation starts with the initial loading of data and terminates when the system eventually becomes obsolete and has to be replaced. During operation, maintenance is required for adopting the system to other requirements (conditions, functions, structures) and for correcting errors that were not detected during validation.

2.2 Phases in Database Design

The design process is mainly concerned with the requirement acquisition, requirement analysis and conceptual design. We can distinguish different phases in database design. Based on the waterfall and/or fountain models of software development, the classical scenario for structure-driven database design distinguishes between conceptual, logical and physical design. During conceptual
design a high-level description of the database (structure, semantics, operations, distribution) is developed using the data and operations requirements. This description is independent from the platforms or the database management systems which are later used. Conceptual design results in a conceptual schema which is kept, for instance, for later database maintenance. During logical design a conceptual schema is translated to a logical schema which describes the structure, operations, semantics and distribution of the database according to a class of database management systems (or according to a single system or, in general, according to a set of paradigms). The logical schema is transformed into the physical schema during physical design. The storage structures, access methods, the organization in secondary memory is developed for a specific DBMS.

Alternatively, the processing-driven database design starts with the processing analysis of requirements which generates a functional schema. This is used during high-level application design for the generation of the application specification which describes the behavior of the database in a high-level abstraction. These specifications are transformed into a detailed program specification during the application program design phase.

Modern, more recent approaches try to unify and to combine both the structure-driven and the processing-driven database design.

2.3 Database Design By Example - Extending the NIAM Approach

Psychological experiments devised to distinguish between the abstraction-based and the example-based approach and support the example-based (or fact-based) approach [16]. Design-by-example is the process of acquiring database information by drawing inferences from examples. Such a process involves design steps of generalizing, transforming, correcting and refining schema representations. This approach is based on the human ability to make accurate generalizations from few scattered examples or to discover patterns in seemingly chaotic collections of facts. This approach is closely related to inductive learning of concepts. In contrast to abstraction-based approaches, examples are the starting information rather than abstract structures. The goal of the example-based approach is to formulate plausible general assertions that formalize the examples, that explain semantics and meaning of the given examples, and that predict further examples to be discussed with the designer. In other words, database design by example attempts to derive a complete and correct description of the database application or of certain parts of database semantics from examples. This approach is supported by tools [2] which are generating plausible examples from the given examples in dependence of the obtained database schema. Therefore, this approach can be considered to be a generalization of the NIAM approach [13] which has been developed partially independently from our approach and is now the common basis for both approaches.

Modern, more recent approaches try to unify and to combine both the structure-driven and the processing-driven database design. Each database concept is associated by a set of operations which are either generically defined or user-defined. Further, each concept is combined with a set of semantical restrictions. Therefore, we can view database concepts as (abstract) data types.

The following methodology supports this approach and is an extension of [1, 6, 13, 17]. We start with meaningful examples (the first step), develop the structure and the semantics of the database under design (the next four steps), generate the operations and model the behavior of the database (the next two steps). Throughout the procedure, checks are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational design can require for performance reasons the redesign of the structure. The eight steps can be defined as follows:

For each designed type (attribute type, entity type and relationship type) the following steps are considered. Then we use these steps also for meta-types (called units later).

1. Transform examples in elementary entity and relationship types, analyze the structure;
2. Graphical representation of types and their associations, population check;
3. Eliminate surplus, derivable types, determine roles;
4. Modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. Normalize the types;
6. Modeling of operators over the types and the behavior;
7. Optimizing types and deriving views;
8. Check the completeness, consistency of the design with the examples.

The design by example approach is useful whenever the design is too complex or requires from the designer high abstraction abilities. This approach is very old. It is known that the Babylonians used examples in order to express mathematical equations. Sometimes, it is much easier to explain certain constraints via examples. Further, using examples the set of possible dependencies to be checked can be reduced even to a comprehensible candidate set. In this case, the designer work on important parts. Then he/she is not enforced to develop the entire constraint set. Since the constraint set is exponential in the number of attributes the design of constraints is one of the most difficult database design tasks. Even worse, the incompleteness of designed constraint sets is the normal case.

Armstrong relations are relations that show both the existence and the nonexistence of constraints for a schema. That is, a constraint holds for the schema if and only if it holds for the Armstrong relation. Armstrong relations exist for a large class of constraints, e.g. for functional dependencies. An Armstrong relation provides also good material for testing the effects of insertions and other updates in the database. Based on the violation property for any dependency which is valid in the schema, updating the Armstrong relation gives a good indication of allowed situation during the lifetime of the relation. Armstrong relations have a drawback when compared with the classical linear representation of a dependency set. For some dependency sets, the minimal size of Armstrong relations is exponential both in the number of attributes and dependencies.

The design-by-example-approach can be helpful. As a rule of thumb, a small number of tuples is necessary to express almost all minimal keys with a small number of attributes. The same is valid for functional dependencies with a small number of attributes in the left side. Therefore, the design of constraints is simplified if the designer has to develop only simple constraints and a design tool helps him/her in reasoning about the remaining set. Two troublesome features of normal database design process must be addressed:

1. Indeterminateness.
2. Uncompleteness. The designer will probably forget to specify certain pieces of semantics.

The indeterminateness reflects the fact that designers are not accustomed to talk about concepts in a way that precisely fits the database design model and design strategy. The incompleteness problem is the problem of how to identify missing or incorrect semantics. The designer, no matter how qualified and thorough he/she may be, is going to forget about certain special circumstances. One of the main assumptions for any design support systems should be the incompleteness assumption. It should be assumed that the database design is incomplete according to the set of constraints.

2.4 NLI-Based Knowledge Acquisition During Database Design

The requirement acquisition is the most difficult step during database design. There are several reasons for that:

The designers start with vague ideas of the orderers. Those may have difficulty putting their ideas into words.

It may be difficult to obtain reliable requirements. Some of the participating orderers may tell the designer what they think the designer wants to hear.

There can be envisioned a functionality that is not consistent with the objectives that is accomplished from the system.

Objectives of different groups may compete or conflict.

Organizational interactions may impede among orderers.
We can distinguish different phases in database design.

During the elicitation phase design information is recognized and drawn out of the designer.

The capturing phase and the transformation phase map design information into external and internal schemata.

Different interview techniques can be used during acquisition of design information [2, 5]. They are used to identify constraints and associations, to structure and refine already-acquired information. Structured interviews force an organization and are goal-oriented. They attempt to remove distortion from designers subjectively and allow better integration of material after the interview. Furthermore, they force the designer to be systematic, and the acquisition tool is able to identify gaps in the semantics which act as a basis for further questions. This process can be combined with open interviews on known constraints which are broad and place few restrictions on the designer. Therefore, in RAD there has been developed an NLI tool to support the designer.

The development of a database system encompasses many knowledge engineering activities in addition to requirement acquisition like accessment of the suitability of the database technology for a prospective application, planning and management of a database project, design of appropriate representation formalism for the information that is acquired, design of the inference and control mechanisms to use the knowledge, implementation and selection of suitable platforms, design, prototyping and implementation of the system, integration of the system with its environment and evaluation of the developed system.

However, the information on which the complete development process is based is acquired during the requirement acquisition process. Therefore, this process could be the bottleneck of the complete design process. If the database design process is supported by tools then the requirement acquisition process is one of the basic processes which needs to be founded. For this reason, sophisticated interview techniques need to be integrated.

3 Properties of Database Design Methodologies

A design methodology could be based on monotone reasoning. Each design step is adding certain information. During design the information already modeled is not changed. This does not mean that structures, operations and constraints are not changed. However, the new design step does not abandon already obtained information. Each design step needs to satisfy certain restrictions:

- Each design step is content-preserving, i.e. there exists a query $q$ such that for each database $db' \in SAT(S')$ : $q(db') \in SAT(S)$ for $S, S'$ and $Step(S) = S'$.

- Each design step is constraint-preserving, i.e. $\Sigma' \models \Sigma$ for $S = (Struc, Ops, \Sigma)$, $S' = (Struc', Ops', \Sigma')$ and $Step(S) = S'$.

- Each step is minimally content-preserving, i.e. the design step does not add derived types; there does not exist a query $q$ such that for each database $db' \in SAT(S')$ there exists a database $db \in SAT(S)$ such that $q(db) = db'$ for $S, S'$ and $Step(S) = S'$.

Furthermore, there are some desirable restrictions:

- Each step is minimally constraint-preserving, i.e. $\Sigma \models \Sigma' \mid_{Struc}$ for $S = (Struc, Ops, \Sigma)$, $S' = (Struc', Ops', \Sigma')$ and $Step(S) = S'$. 
• Each step is nearly free of path tuple-generating constraints. When a path tuple-generating constraint is present it often requires that the path has to be computed to check whether the constraint holds in the database.

Notice, that inclusion constraints can be maintained in a database using only projections. Path inclusion constraints are still based on joins. The last restriction is often too strong. It can be replaced by the following restriction. This restriction requires an effective management of basic database operations.

• Each step is update simplicity-preserving, i.e. if for $S$ the complete set of basic update operations is defined without join operations then $Step(S)$ has this property too.

A schema is lexical-minimal is it uses as few lexical units as possible. For instance, the attribute DepartmentName of the type Department is can be replaced by Name (the corresponding identifier is Department.Name).

It should be mentioned that several known normalization algorithms do not fulfill the above requirements. However, there are 4NF algorithms which preserve all properties.

A methodology needs also rules in which case which concept is going to be applied. One rule which could be considered as the basic rule is that independent objects are to be represented by entity sets.

A database design is similar to database lifetime. Each design step can be modeled by some transaction. The database is designed step by step using one of the given transactions. Each transaction commits or aborts the actions taken since the last last transaction committed. Therefore, the design strategy can be considered to be simple if the design is performed by a sequence of transaction. Often, design strategies are proposed which consists of one complex transaction. Such strategies are prone to errors. Errors and wrong assumptions made at the beginning are corrected only at the end. However, several implications have been derived during design from those assumptions. Therefore, error deletion or revision of assumptions lead in such cases to revision of the entire design process. We need to distinguish between wrong assumptions and errors. Wrong assumptions have several reasons. Errors are detectable at a certain stage of the design procedure. The principle should be that the detection should happen during the most early step.

• A database design step is closed if errors which could have been made during this step must be corrected before this step is going to be completed.

Checkpoints are an instrument to develop design strategies with closed steps.

In order to achieve closed design strategies a set of general design business rules can be developed. Design consistency is maintained by enforcing these rules during database design. Design business rules can fall into the following categories:

Model-specific. Each database model has a certain set of implicit structural constraints and integrity constraints which are satisfied by each schema. Examples of such constraints are the existence of a primary key based on the key uniqueness and the entity integrity.

Schema-specific. Certain models allow or permit the existence of cyclic constructs. If, for instance, derived attributes are designed then a certain set of operations must be designed too. In this case, any change to the schema should not violate the schema-specific constraints.

Application-specific. Organizational policy, laws and general structure of the application affect the design strategy. If, for example, the application under design is highly heterogeneous then the strategy could use that and begin with design of different views. If main requests to the database require computation over different views then the conceptual schema needs to be designed with special care.

The database design methodology developed in [19] is an example of such a strategy driven design business rules. Another example which considers mainly application-specific design rules is discussed in [9]. The rules discussed in [7] are model-specific design rules.
4 On Design Methodologies

4.1 Design Perspectives

In software design and in database design we distinguish different design perspectives.

1. The **structure-oriented perspective** focusses on the structural description of the data. Almost all known database design approaches are based on the structure-oriented perspective. Sometimes, the structure-oriented perspective is unified with the semantical perspective. In this cases, the design of the structure is to be combined with the design of static integrity constraints. The different strategies based on the structure-oriented perspective are shown in Figure 2.

2. The **behavior-oriented perspective** (called in [10] integrity-centered; see also [11]) is concerned with the behavior of the database during its lifetime. It can be based on event approaches [11] or on Petri-net approaches [14, 15] and predicate transition systems.

3. The **process-oriented perspective** is concentrated on the database operating. This perspective has been considered mainly for software design. Several strategies have been developed. Only recently it got attention by the database designers.

The structure-oriented perspective is discussed in a large variety of papers. The process-oriented perspective can reuse approaches known in software engineering. The behavior-oriented perspective is a high-level descriptive approach to an integrated specification of structures and operations. Database design is considered to be a transformation process carried out on integrity constraints. Database behavior is specified by stepwise reification.

The structure-oriented approach is currently well developed for relational models and was directly generalized to semantical models like entity-relationship models. It is, however, possible to develop for semantical models more specific methodologies which are using the specific properties of these models. One specific methodology is discussed in the next subsection. Generally, a methodology consists of a strategy, a set of primitives to be used and guidelines on the application of each strategy step and each primitive. A methodology should be, from one hand side, rigorous, i.e. suggesting a strategy which is to be used, and, from the other hand side, flexible, i.e. applicable to a variety of applications. These requirements are contradictory in general. A design tool should be flexible enough to be adaptable to different design styles and should be rigorous enough to be well-based.

The building block of a methodology is the design strategy. Known strategies are, for example, the bottom-up and top-down approach. Most of the known strategies are considering only the structural part of a schema. The strategies are based on different primitives. Most of these primitives can be considered to be generalizations of the known algebraic operations of the relational model.

4.2 Design Primitives

In [4] design primitives were introduced. We distinguish between structural abstraction and graphical abstraction. The last one is understood as a kind of representational abstraction. In this case, we define a mapping from the set of possible schemata to the set of graphs. This mapping is called injective if each graph image corresponds only to one preimage. The mapping is called basic if it is defined as the extension of the mapping from primitive types (entity, relationship and attribute types) to primitive graphical units (rectangles, diamonds with arrows, points with lines; may be with names). The structure is context-sensitive because the corresponding entity types must exist for component types of relationship types. However, the context sensitivity follows the order of the types. Therefore, the mechanism for defining a structure are not equal-order-context-sensitive. The mapping mechanism for the presented extended ER-model is injective and basic. The mapping mechanism used in most ER books is injective but not basic. The reason for this is that subtypes and their supertype need to be mapped together. If the mapping mechanism is not basic then during database design the design of the database structure needs to be combined with the graphical representation. Since in such design languages the graphical structure is on the equal-order-context-sensitive the graphical structure can change strongly for even very simple changes. Tools based on such approaches are more complex.
Further, since database design can be considered to be database management on a schema database we can compare the design primitives with operations in the design database system. In the nested relational model different operations are used: projection, selection, partition, join, $\theta$-join or the join via a coercion function, union, nest, and unnest.

Additionally, concepts can be added to the schema. The operations can be seen also as decomposition/composition pairs:
- selection / union,
- partition / union,
- nest / unnest,
- projection / product (or join).

We have mentioned already that each database model needs its own primitives according to the constructors used in the definition of types. In HERM we are using the type constructors product (generating tuple-sets) and union (generating clusters or generalizations of subtypes). Therefore, for a complete set of design primitives we need only primitives defining the generation of products (i.e. product and concatenation to components (The last can be expressed via the first operation. However it might be sometimes more convenient to have this operation too.)) and the generation of union types. Further, we need operations for initializing the design (in our case, generating basic attribute types and generating basic entity types but not for the generation of relationship types). We define now these general design operations. For that we assume a set $C$ of type constructors. Using this set we can define a set of terms and subterms for each term. For instance, if the $C$ consists of union $\cup$ and product $\times$ then the set $T$ of terms is defined by the consecutive application of these operators and the subterms are defined by selection and projection operators. Further, we assume $\perp \in T$ for the empty term.

**Composition.** Given the constructor $C$, a new name $t$ and types $t_1, \ldots, t_n$. The composition $\text{compose}(t_1, \ldots, t_n, C, t)$ defines a new type $t$ using the constructor $C$.

**Decomposition.** Given a type $t$ and terms $e_0, e_1, \ldots, e_n$ and (possibly empty) names $N_0, \ldots, N_n$.

The decomposition $\text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n)$ defines new types which are generated by $e_i$ and are labeled by $N_i$.

**Extension.** Given a type $t = (t_1, \ldots, t_m, \{A_1, \ldots, A_n\})$ of order $i$ ($m \geq 0$ ($m = 0$ for entity types), $n \geq 0$), an attribute $A'$ and a type $t'$ of order $j$, $j < i$ then $\text{extend}(t, A)$ is the type obtained from $t$ adding $A'$ to the attribute set and $\text{extent}(t, t')$ is the type obtained from $t$ adding $t'$ to the component sequence.

For our purposes entity types are types of order 0.

**Initialization.** $\text{generate}(E)$, $\text{generate}(A)$ generate an entity type labeled by $E$ without attributes and an attribute type labeled by $A$.

### 4.3 Dimensions in Database Design

Database design has been based in the past on relational design techniques. Two main approaches have been developed: top-down approaches (decomposition) and bottom-up approaches (synthesis). The first strategy begins with a set of basic types and builds new types using decomposition. The second approach begins with a set of descriptions or attributes and then group those in types.

The relational decomposition approach starts with a given schema of relation types and decomposes it to smaller relation types until certain properties such as normal form and losslessness are satisfied. The starting schema can be a universal relation type or a near-normal-form set of types. In the synthesis approach to designing relational databases, we assume the application is described by a set of attributes and functional dependencies among them. The functional dependency (FD) is the main vehicle in synthesizing relation types out of lists of attributes. In this case, we start with a minimal cover of FDs. Normally, the two strategies are not applied in their pure form. Database design could be performed by mixing the two strategies which is sometimes called mixed strategy.
These two approaches could be considered as approaches with different design directions.

Design strategies can be distinguished also by their control mechanism. One control mechanism rules the step-by-step development of types. Control strategies decides which rules to use in a given situation. The inside-out strategy selects first one central type and then develops the associated type by a discrete neighborhood function. The next type to be developed is a new type with the smallest neighborhood value according to the central type. Another control mechanism exploits another more abstract schema of the database. This mixed strategy is controlled by a sketch of the database schema (the so-called skeleton). Alike in second-order grammars this control mechanism can be generated by another grammar. The skeleton schema is representing the control information of the database design.

![Diagram](image)

**Figure 2: Main Dimensions in Structure-Oriented Design**

Relational database design has been considered mainly for the design of conceptual schemata according to the three-level architecture of a database. The conceptual schema unifies the different views of external users. The database design could also begin with the view design [7]. The views are later composed into the conceptual schema. This approach can be useful for the design of homogeneous view sets. Since view integration is undecidable the applicability of this approach is limited. The modular design strategy Design-by-units extends this approach. For each subschema which could be considered to be a module alike in software technology the interface is defined for each unit. Units can be associated only by their units. We can define also other kinds of scoping rules in database design. This dimension of database design uses the modularity concept. Modularization is based on implementation abstraction and on localization abstraction. Implementation abstraction is to selectively hide information about structures, semantics and the behavior of concepts defined by the previous two abstraction mechanisms. Implementation abstraction is a generalization of encapsulation and scoping. It provides data independence through the implementation, allowing the private portion of a concept to be changed without affecting other concepts which use that concept. Localization abstraction to "factor out" repeating or shared patterns of concepts and functionality from individual concepts into a shared database / knowledge base application environment. Naming is the basic mechanisms to achieve localization. Parametrization can be used for abstraction over partial object descriptions.

Another dimension not represented in the figure is the representation form for design results. For instance, the ER model uses rectangles, diamonds as nodes of a graph. The IFO model
uses more sophisticated node types and different types of edges. The representation is more compact but also more complex. The NIAM model uses object nodes for abstract objects and relationship nodes for the representation of associations among objects and relationships nodes. The graphical representation differs from model to model. The most important issue is, however, that the graphical representation has to be based on a well-founded semantics. Often, especially in the ER literature and for ER-based tools, design is considered as drawing graphs leaving semantic issues aside.

The power of abstraction principles comes from their orthogonality and their generality. All these mechanisms can be combined with more specialized mechanisms, like exception and default rules.

If we compare different strategies then we can relate them to known search strategies in AI. Search strategies are characterized by direction, topology, node representation, selecting rules and heuristic functions. Other design strategies can be developed based on this analogy. We could search forward or backward.

5 Directions of Database Design

5.1 The Top-Down Strategy in Database Structure Design

The pure top-down strategy is based on refinement of abstract concepts into more concrete ones. Each step introduce new details in the schema. The design process ends when all structural requirements are represented. According to the above discussed taxonomy of database design operations the following primitive operations can be used during top-down design:

\[
\text{decompose}(t, e_0, e_1, \ldots, e_n, N_0, N_1, \ldots, N_n) \]
\[
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \]
\[
\text{generate}(E), \text{generate}(A) \]

The advantage of top-down design is the independent consideration of each type during a design step. The designer can analyse one concept ignoring the other concepts. However, top-down design requires from the designer the knowledge on the general picture of the design first. Therefore, complete top-down design can be performed only by experienced designers if the application area is not well-structured. Based on this approach, different primitives are developed.

5.2 The Bottom-Up Strategy Database Structure Design

Bottom-up design was first developed for the relational model. During bottom-up design each step introduces new concepts not considered before or modify existing ones. During each step the designer checks whether all features of requirements are covered. The designer start from elementary concepts and builds more complex out of them. The advantage of the bottom-up approach is its simplicity. The main disadvantage of the bottom-up approach is that this approach requires restructuring. Therefore, the strategy is not monotone.

The following primitive operations can be used during top-down design:

\[
\text{compose}(t_1, \ldots, t_n, C, t) \]
\[
\text{extend}(E, A), \text{extend}(R, A), \text{extend}(R, E) \]
\[
\text{generate}(E), \text{generate}(A) \]

6 Control Strategies

6.1 The Inside-Out Strategy in Database Structure Design

The classical inside-out strategy restricts the bottom-up approach by controlling the order of primitives application. This strategy is still complete. We choose the most important concept first, design it and then proceed by moving as an oil stain does, designing first concepts that are
conceptually closed to the already design. The order of refinements is disciplined. The designer navigates from the central type to the more distant ones. It is easy to discover new concepts which are to be designed next. However we lose abstraction capabilities. The global schema is built at the end only.

This strategy can be generalized now as follows:

Given a set of concepts \( C \) to be designed and a neighborhood function \( F \) on \( C \), i.e.

\[
F : C \times C \rightarrow \text{NatNumber}.
\]

This function is used for the design agenda. In this case we use the following algorithm for a given central concept \( c \):

**Algorithm**  
**Bottom-up version**

1. \( \text{Agenda} := \{c\} \); \( \text{Designed} := \emptyset \).
2. Repeat until \( \text{Agenda} = \emptyset \)
   (a) Repeat until \( \text{Agenda} = \emptyset \)
      i. Choose \( c \in \text{Agenda} \)
      ii. Design \( c \) using the bottom-up strategy
      iii. \( \text{Agenda} := \text{Agenda} \setminus \{c\} \)
           \( \text{Designed} := \text{Designed} \cup \{c\} \)
   (b) \( m := \min\{F(c, c') \mid c \in C \setminus \text{Designed}, c' \in \text{Designed}\} \)
       \( \text{Agenda} := \{c \in C \setminus \text{Designed} \mid F(c, c') = m \text{ for some } c' \in \text{Designed}\} \)

For the top-down variant of the algorithm, step 2.(a).ii is used for the design in top-down fashion. This algorithm can be refined. For instance, if the set of concepts is unknown at the beginning then a substep is added to step 2 which looks for extensions of \( C \) and of \( F \) in each iteration.

6.2 The Mixed Strategy in Database Structure Design

Another controlled approach is the mixed approach. This approach mixes the top-down and the bottom-up approach. First, a skeleton schema is to be designed (using one of the previous approaches). This schema represents the main classes (or units) and their main associations. Then each of the units is refined and later integrated with other designed units. Using the skeleton schema the bottom-up integration of concepts is simpler. Since the complete requirements set is now partitioned the design of each unit is less complex. The success of the strategy depends from the design of the skeleton schema. Therefore, this method is applicable if the application is already well-recognized.

7 Modular Design by Units

7.1 Design by Units - Structural Design

In [24, 21, 20] another new design methodology was developed: **Design-by-units**. Most of the well-known design methodologies think as in the relational approach. But each of the database models should have its own methodology. It is surprising that most of the proposed models do not have its own design methodology. If the model is getting richer in construct the methodology should be deepen. The proposed methodology support also extensibility since using this methodology an existing design and implementation can be extended without introducing changes to it. It promotes reuse and inheritance as well as behavioral extension. To some extend, this approach is similar to modular design known in software engineering. The orientation is different only. We are first interested in the data representation part and then in the processing part since a part of the processing part is based on generic operations which are defined according to the structure. However, if we consider modularization, parametrization and inheritance to be the kernel concepts of object-oriented design then this design approach can be considered to be complete.
object-oriented. This approach has further other advantages: it is easier to detect similarities among design units and to reuse parts of design units in other units; changes to the scheme and to parts of units are directly reflected in all other units which are using the changed. The new methodology supports directly the above discussed distinction between kernel and dependent object. This is especially useful, if abbreviation techniques [18] are used in query forms [24]. It is a recursive method. Each step is based on the above discussed eight design steps. This methodology is not following the classical waterfall model with iterations but rather supporting a high level inside-out-strategy [4]. Experience in utilization of (DB²) has shown that this methodology was the most often choosen for practical design.

**Design-by-units**

1. **basic step.**
   Design the types for the independent kernel object classes.

2. **Recursion step.**
   Repeat until the schema is not changed.
   Either reification :
   - Refine the units introducing subtypes (in HERM represented by unary relationship types).
   - Refine the relationship types according to the subtypes.
   or generalization of units:
   - If there are associations among units then introduce a new unit containing the associated units and the relationship type according to the association.
   - Add other relationship types if there exist new associations.

### 7.2 Design by Units - Process Design

As already discussed in the previous section, the data design and the process design can not be separated from each another. We need the process information as well as the structural information. For this reason the process design and the structural design need to be integrated. We use a dataflow approach [4]. A process is an activity of the information system. The dataflow is an exchange of information between processes. The processes use information from the database and create temporary databases necessary for the process. Since processes use different databases and these databases are usually not fully integrated, interfaces are to be used for view cooperation. Generally, the interface is the description of the cooperation between different users (originator/receiver of the dataflow).

One of the difficult tasks in processing modeling is to evaluate whether the designed data structures are appropriate for an effective processing of data. It is known already that relational normalization can contradict effective processing. Sometimes unnormalized relations can be used simpler. For handling this we need a cost model for processes. The cost model can be based on models of complexity for operations and on priority functions for queries and transactions. Therefore, we need a representational and a graphical language for the representation of processes. We decided to use three different languages, one high-level abstract language which is directly supported by the modular design-by-units-strategy, another one for a more detailed procedural description and and another more dataflow oriented which shows the level of the database directly.

A *transaction* is a (nested) sequence of database operations which transforms consistent databases into consistent ones. They can be described like queries. However, modeling of transactions provides additional information about the relationship and the order of database operations. In the same manner, an application program can be viewed as a structure of queries and/or transactions that manipulate a database or several databases.
8 Derived Design Strategies

The experience of (DB)^2 utilization shows that different database designers use also very different design strategies. The system RAD supports this variety of design strategies. For each user, a specific database design strategy can be developed which is derived from abilities and experience of the specific user. This chapter demonstrates two derived strategies. The variety of possible design strategy is much larger.

8.1 Relational Database Design Combining Mixed and Bottom-Up Design

We show now that a closed relational database design strategy can be derived which is based on entity-relationship modeling.

**Mixed strategy on user views.** The user view represents the data requirements of a single user. The skeleton under development contains only the most important aspects of data requirements.

**Bottom-up design: A - Identification.** Now we add detail information to the developed user views.

**Bottom-up design: B - Characterization.** Each object of a certain type has normally beside its identification also characterized properties.

**Integrity control: A - Validation.** The normalization of relational schemes has been generalized to ER schemes. Normalization is a theory that addresses analysis and decomposition of data structures into a new scheme that has more desirable properties.

**Bottom-up design: C - Domains.** Now constraints on valid values that attributes may assume are added.

**Integrity control: B - Behavior.** The specification of the database is used for the generation of the behavior of the database.

**Bottom-up design: D - Combination and Integration.** Each view developed so far models the information requirements for one special application. They are now comprehensive, correct, and unambiguous. Now these views should be integrated.

**Integrity control: C - Stability and Extension.** The database is changing also its structure during its lifetime. This needs to be supported. Scheme evolution is similar to view integration.

The next steps will be only mentioned. For those, we can develop complete descriptions in the same manner.

**Translation and Validation**

**Optimization: A - Access.**

**Optimization: B - Indexes.**

**Optimization: C - Controlled redundancy.**

**Reconsideration and redefinition**

**Coping with challenges: A - View updates.**

**Coping with challenges: B - Security.**

**Coping with challenges: C - Very large databases.**

**Coping with challenges: D - Access change.**

**Coping with challenges: E - Future technology.**
8.2 Modular Object-Oriented Design Based on Top-Down-Directions

Modular object-oriented design is based on a database model which incorporates structural, semantical, and behavioral model. The model developed and used in [19] could be used as one of such. The designer specifies the types, classes and methods of the given application. Similar to the previous strategy, we assume that kernel types and classes are specified. Then the specified scheme is refined by adding information on the database and changing the structure of the database or refining types in the schema. This change can be considered [19] as reification of the scheme and its types. This approach allows the development of generic class libraries for application areas. The types in the library are general enough to be used again after reification according to the specific applications of the given application area. This approach quickens the development of schemes. For modular development of schemes, we specify the type, its interface, its semantics and meaning and its behavior within the database. The reification follows that specification. Therefore we can distinguish the following categories of reification:

Reification by Specialization. Refinement by specialization introduces new subclasses and reorganizes associated relationships. Moreover, it may involve to replace a structure expression such that the new type will be a subtype of the old type and the new implicit and explicit constraints will imply the old ones.

Reification by Splitting. Refinement by splitting also leads to new types. But their type corresponds to parts of an existing type which in turn is replaced by references.

Reification by Completion. Refinement by completion means the definition of new types, constraints, methods, and behavioral constraints.

Reification by Instantiation. The schema may contain undefined parameters, attributes, and types. Refinement by instantiation provides definition for those concepts. The same applies to class parameters. Reification by instantiation may introduce new parameters as well.

In all these cases, additional changes to methods are required.

9 Computer-Aided Database Design by RAD

At present, we can distinguish already three generations of database design tools [4].

- The first generation tools were based on the classical “waterfall” model of software development: requirement analysis, conceptual design, logical design, testing and maintenance. Most of them were platforms for a design from scratch. These tools did not support changes during the life-cycle of the database. In most cases, they were using the relational database model.

- Second generation tools which become available now are designed as complete workbenches for the design support over the complete life-cycle of a database. Such tools use graphic subsystems and support consistency of the design. Some of them help the user to encounter which information is entered several times and/or which is inconsistent. Further, some systems generate design documentations for the complete design process. Most of the workbenches are adaptable to different platforms and can generate different translations for a given design.

- Although second generation tools are now putted into practice third generation tools are already under development. There are already proposals how tools can be customized. Third generation tools will be more user-friendly and user-adaptable (using, for instance, user-driven strategies which are influenced by special organizational approaches in enterprizes). Users can use strategies which are model-dependent and have a support for reasoning in natural language. They provide a tuning support. Users can use object-oriented development strategies.

At present, there is no tool which completely supports a complete design methodology. Most tools available support a restricted part of the design process in a restrictive manner. There are tools which do not support integrity and/or are mainly graphical interfaces. The reason for this situation is that before a strategy and a methodology can be supported by a tool set it first needs to be well specified, theoretically based and extensively tested.
9.1 The Design System ($DB^2$)

The system ($DB^2$) (Data Base Design by Beta; $DBDB = (DB^2)$ [23]) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. ($DB^2$) supports database design from requirements analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual design, thus offering a choice between the relational, network, or hierarchical models for the logical design. Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can be used for creating query compilers which are more productive, effective and forceful.

The tool box uses an easily intelligible, comprehensible and understandable "fill in the blanks" input procedure which is required in the literature as the ideal input form.

The system can be used for the support over the complete lifecycle of database systems. Most known methodologies are not well adapted to the lifecycle because the design information is not used after the design process. Design design is a far more iterative process as captured in the straight forward lifecycle model of these methodologies. Using ($DB^2$) during the whole system development the complete design information is usable for restructuring the database. This makes it possible to restructure the database and during restructuring to recompile in accordance to the old and new structures the programs which are specified in the HERM algebra.

At present the system ($DB^2$) is widely used for database design. There is a large number of active groups which are using the system. However this system was developed as a prototype and was not developed as an application system. Therefore several extensions and improvements are necessary [3] and are to be developed in the RAD system [3]. This tools provide also a better understanding of design task like abstraction (e.g. analysis of concepts), editing (e.g. correctness, consistency, complexity), refinement (e.g. context representation) and transformation (e.g. prototyping, population or test data generation).

Most of the disadvantages are caused by the prototypical architecture, the limited computational power and by the limited reasoning support of the ($DB^2$) interface. The last requirement is the most important. Database design is at present performed by "experts" who have detailed knowledge of a design methodology as well as the benefit of experience gained from previous designs. Since these experts are called upon to work in many diverse areas they often do not know to much about the specific application. They need a knowledgable user who supplies the application expertise. As a result, the designer often ask questions that appear unnecessary to the user or teach the user on database design. A better approach is the expert system approach. The user is introducing his knowledge on the application area and the system supports him by translating this knowledge to schemata. For that, the system should use "real world" knowledge, should have reasoning capabilities and should learn from experience, as a human designer. Furthermore, the system should explain the corresponding decisions and questions to the user. Therefore, the design system needs a component based on techniques known in AI.

9.2 Extending ($DB^2$) to RAD

The system RAD [3] is intended to become a toolbox for the interactive, object-oriented design of databases, and which is being developed within a cooperation between the universities of Rostock, Aachen and Dresden in Germany. The solution of RAD is to build a toolbox that is configurable with respect to a variety of aspects, to different database design strategies and to different users.

Basically, the system in Figure 3 is consisting of four major components:

1. HERM*-Editor, the interface to the outside world, in particular to the designer and the application he or she wants to design. It will provide a language based on an extension of the entity-relationship model for defining syntactical structures, static semantics and behavior of a given application and will hence support conceptual database design based on an object-oriented methodology.

2. The Support System, a component to support users of the system during the various phases of a design process. It has two subcomponents: The User’s Guide will comprise a tutoring
Figure 3: Architecture of RAD
system for guiding the user through object-oriented design in general as well as through HERM+ and the database design process in particular; it will also support him or her in choosing design strategies, and provide examples. The Analyzer will allow a designer to get online feedback regarding design decisions, and critically review the result of a design.

3. The Translator, a component for translating the result of some database design written in the HERM+ language into the language of a specific database system that is to be used in the given application. Specifically, it is intended to have the Translator support a variety of concrete data models, including the relational, nested relational, network, hierarchical and complex object models. Additionally, it should contain a component for performing translations between any two of these models.

4. The Modifier, a subsystem which can be incorporated for modifying a given, already completed design or for altering one or more design decisions. To this end, it will consist of means to modify syntactical structures, static semantics as well as behavior, will use a Normalizer for performing normalization on parts of a design or an entire design according to user needs, and will have a Redesigner for generating the input for the Translator in case a design has been modified.

Figure 1 shows the dependence graph for the design information. The designer can move from one design part to the other design part according to his skills and his understanding of the design process and the application area. This freedom in choosing the personal design methodology is further supported in RAD [3] by a special user adaption tool.

Further, at present it is planned to develop the system RAD for the support of different, more sophisticated design tasks like:
During database design different versions of the same design should be stored and considered together. Therefore, a design tool needs capabilities known from text processing. It is useful to reuse parts of other schemata and to modify parts of other schemata. The reuse and redesign of schemata should be supported especially in the case of changing environments (other DBMS, other versions of DBMS). If the tool is to be used by designers without deep knowledge of database and system theories the design tool should supervise the designer and explain critical decisions. If a subschema is not well designed or leads to complex operations for maintenance the design system should be able to support the designer during modification and redesign of the schema.

The database design process could be considered to be a knowledge acquisition process. The database design process can be viewed as a special kind of cooperation between the designer and the system. The system requests information that might be useful in the design and makes suggestions for schema reductions and other improvements. The designer supplies the requested information and accepts or rejects the suggestions. The designer may also add information not requested and transform the schema in ways other than those suggested by the system. The design system needs also component that enables it to learn during design and to learn the designers behavior. Further, the system needs reasoning capabilities and a knowledge base on commonsense knowledge (classificatory and generic; industry-specific and organization-specific knowledge). Whenever, the designer adds user-specific and application-specific information the system can generate the relationships to the world being modeled.

The system is based on the graphical tool GraphEd [8] which is a powerful interactive SUN-based editor for graphs, using windows and has a special open data interface. The graphical representation uses graph grammars.
References

Das Abstraktionsschichtenmodell für den integrierten Entwurf von Dialogen, Sichten, Funktionen und Schemata von Datenbankanwendungen

Bernhard Thalheim*  
Inst. für Informatik, Brandenburgische Technische Universität Cottbus  
P.O.Box 101344, D - 03013 Cottbus

Zusammenfassung


Wir entwickeln nun eine Datenbankentwurfsmethodik, die die Spezifikation aller Teile schrittweise miteinander integriert und frei von Brüchen ist. Damit kann die Struktur, Funktionalität und Semantik konsistent in Übereinstimmung mit den Aufgaben der Datenbank und bei Berücksichtigung der Oberflächen entworfen werden. Die Methodik basiert auf einem Abstraktionsschichtenmodell.

1 Der Datenbankentwurf


Ziel des Entwurfes ist im wesentlichen eine Abbildung der (dynamischen) Handlungen der Realität wie in Bild 1 auf Ereignisse in der Datenbank. Damit besitzt die Spezifikation der Handlungen eine größere Bedeutung als im klassischen Entwurf angenommen. Nach [Bis95] sollen Änderungen aufgrund von Handlungen sich in Datenbankanwendungen in stati-
B. Thalheim: Integrierter Entwurf von Datenbankanwendungen mit dem Abstraktionsschichtenmodell

Reale Welt

<table>
<thead>
<tr>
<th>Akteur (mit Organis.-modell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rolle</td>
</tr>
<tr>
<td>Aktivität (rekursiv definiert)</td>
</tr>
<tr>
<td>Ziel (sequentiell definiert)</td>
</tr>
<tr>
<td>(Ist,Soll)</td>
</tr>
<tr>
<td>dient der Erfüllung</td>
</tr>
</tbody>
</table>

Computer-arbeitsplatz

<table>
<thead>
<tr>
<th>Prozeß (rekursiv definiert)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rolle</td>
</tr>
<tr>
<td>Ereignis, Transaktion</td>
</tr>
<tr>
<td>Benutzer (mit Org.-modell)</td>
</tr>
<tr>
<td>Zustandsänderung (sequentiell definiert)</td>
</tr>
<tr>
<td>(db_i, db_i+1)</td>
</tr>
<tr>
<td>dient der Erfüllung</td>
</tr>
</tbody>
</table>

Modelierung

<table>
<thead>
<tr>
<th>Klassisches Herangehen</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dynamische)</td>
</tr>
<tr>
<td>Geschäftsverläufe</td>
</tr>
<tr>
<td>Lokale Datenbanken</td>
</tr>
<tr>
<td>(dynamische)</td>
</tr>
<tr>
<td>Gesamtplanen</td>
</tr>
<tr>
<td>Prozesse</td>
</tr>
</tbody>
</table>

Benutzerorientiertes Herangehen

<table>
<thead>
<tr>
<th>Akteur in einer Benutzerrolle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
</tr>
<tr>
<td>(Dialogoberflächen mit Funktionen)</td>
</tr>
<tr>
<td>statische Absperrorung</td>
</tr>
<tr>
<td>Datenbank(schema)</td>
</tr>
<tr>
<td>statischer Zustand</td>
</tr>
<tr>
<td>Log-Dateien</td>
</tr>
<tr>
<td>dynamischer Zustand</td>
</tr>
</tbody>
</table>

Bild 1: Geschäftsverläufe werden auf Ereignisse abgebildet

Bild 2: Der Zustandsänderungsprozeß durch Aktionen von Akteuren (Benutzern)

z.B. objektorientierte Modelle eine Lösung zu finden. Die Literatur zum Datenbankentwurf ist von einer Reihe von Problemen, ungelösten Fragestellungen und inadäquaten Lösungen geprägt. Dazu zählen insbesondere die folgenden Problemkreise:

2. Ein Schema kann für eine Gruppe von Anwendern adäquat sein, für eine andere aber nicht.
4. In der Literatur werden Modetrends verfolgt, die einer Solidität bei der Entwicklung von Modellen entgegenstehen.

2 State-of-the-art von Entwurfsmethoden mit Sprachbrücken

tiefgehende Fachkenntnis des Anwendungsbereiches.


Diese Liste ist nur ein Teil einer umfangreichen Feh- lerliste.

Der klassische Entwurf einer Datenbankanwendung ist von einer Reihe von Brüchen gekennzeichnet.


**Struktur-/Semantikbruch:** Datenintensive Anwendungen zeichnen sich meist durch eine komplexere Struk- tur aus. Die statische Semantik wird entweder intuitiv durch die angewandten Konstrukte ver- standen oder erfordert wie im relationalen Fall tiefgründige Kenntnis der mathematischen Lo- gik. Damit wird aber entweder die Konsistenz in der Spezifikation willkürlich oder nicht mehr nachvollziehbar.


**Workflowbruch:** Geschäftsprozesse können analog zu langandauernden Transaktionen im Ablauf un- terbrochen werden, auf anderen Geschäftsprozes- sen basieren und unterschiedliche Granularität besitzen. Damit entsteht ein komplexes Ausfüh- rungsmodell, das von einem Normalentwickler nicht mehr überschaubar wird.


Diese Brüche entstehen durch

- unterschiedliche Spezifikationssprachen und
- unterschiedliche Semantik und Bedeutung der einzelnen Sprachkonstrukte.

Außerdem impliziert sie eine **Nichtberücksichtigung der Bedürfnisse des Endnutzers**.

Wir entwickeln eine **konsistente, allen Bereichen ge- rechtwerdende** Methode zur Modellierung zu entwickeln. Diese Methode ist *nicht einfach*, weil auch für den Entwurfsprozeß keine universelle und zugleich einfache Weltformel existieren wird. Dabei können wir die Existenz von geeigneten Modellen zur Spezifikation der Struktur (z.B. das HERM) [Tha93], der Sichten [Mai83, MaR92, Sch96], der Prozesse [Tha97] und der Dialoge [Mei95, Vale82] ausgeben und diese auch wie in [Tha97] demonstriert im Datenbankentwurf inte- griert verwenden.

### 3 Das Abstraktionsschichtenmodell

Allgemein bekannt - jedoch nicht ausgeführt - ist die unterschiedliche Granularität der Typen, die in einem Entwurfsprozeß entstehen. Im Entwurfsprozeß sind verschiedene Abstraktionsebenen konsistent zu verwalten. Wir erhalten damit eine Beziehung von

Da die Einzelmethoden bereits z.T. umfassend ausgearbeitet wurden, ihre Integration aber kaum in der Literatur behandelt wurde, wird nun eine Integration dieser Einzelmethodiken vorgestellt. Wir haben im Verlauf der Diskussionen herausgestellt, daß der Entwurf einer Datenbank im wesentlichen aus vier Teilen besteht:

Die Spezifikation des Datenbankschemas zur Darstellung der Struktur und der statischen Semantik.

Die Spezifikation der Funktionen zur Darstellung der Prozesse, die für eine Datenbank zugelassen sind, und der dynamischen Semantik.

Die Spezifikation der Dialoge zur Darstellung der Arbeit der Benutzer mit der Datenbank, d.h. zur Darstellung der benutzbaren Szenen und zur Darstellung der Oberflächen. Die Szenen und die Oberflächen basieren auf den Prozessen und den Sichten aus dem strukturellen Entwurf.

Die Spezifikation der Sichten auf die Datenbank für die Bearbeitung der externen Schemata.

Wir erhalten mit diesen Anforderungen die Aufgabe, die Struktur, die Funktionen, die Dialoge und die Sichten auf eine Datenbank im Zusammenhang zu entwerfen. Vereinfachend ist dabei, daß die Dialoge auf den Sichten und den Prozessen aufsetzen und daß die Sichten in das Schema einbindbar sein sollen. Die Prozesse werden damit über diese Einbindung in das Schema auf für die Sichten benutzbar. Damit erhalten wir ein Entwurfsviereck bestehend aus der Datenspezifikation, der Funktionspezifikation, der Sichtenspezifikation und der Dialogspezifikation.

Damit ist eine Spezifikation einer Datenbankanwendung sich wie im Abstraktion-Spiralmodell auf unterschiedlichen Abstraktionsstufen darstellbar. Wir unterscheiden wie in Bild 4

- die Motivationsschicht zur Spezifikation der Ziele, der Aufgaben und der Motivation der Datenbankanwendung,
- die Geschäftsprozeßschicht zur Spezifikation der Geschäftsprozesse, der Ereignisse, zur Grobdarstellung der unterlegten Datenstrukturen und zur Darstellung der Anwendungsstory,
- die Aktionschicht zur Spezifikation der Handlungen, der Detailstruktur der Daten im Sinne eines Vorentwurfs, zur Darstellung eines Sichtenskeletts und zur Darstellung von Szenarien zu den einzelnen Anwendungsstories,
- die konzeptionelle Schicht zur Darstellung der Prozesse, des konzeptionellen Schemas, der konzeptionellen Sichten und der Dialoge in zusammenhängender Form, sowie
- die Implementationschicht zur Spezifikation der Programme, der physischen und logischen Schemata, der externen Sichten und zur Darstellung der Insznierung.

Das Abstraktionsschichtenmodell erlaubt einen Datenbankentwurf im Zusammenhang. Wir können ein schichtenorientiertes Vorgehensmodell anwenden ebenso wie ein Modell, das sich zuerst auf eine der Komponenten orientiert.


4 Codesign von Datenbankanwendungen


B. Thalheim: Integrierter Entwurf von Datenbankanwendungen mit dem Abstraktionsschichtenmodell


Da das erweiterte ER-Modell benutzt wurde zur Spezifikation der Struktur und der Sichten, existieren die generischen Funktionen *insert, delete, update* für jeden entwickelten Typen. Außerdem lassen sich die generischen Funktionen effektiv berechnen. Da-

**Bild 5: Das Sichten-Struktur-Prozess-(Dialog-)Codesign-Modell mit Abstraktionsschichten (Datensicht)**

- Motivationsschicht: Motive, Ideen, Aufgaben, Workflow
- Vorstudie: Externe Schemata, Diagramme-HRM, Grobentwurf, Herausschnitt, HERM-Programme, Grobes HERM-Schema
- Geschäftsprozessschicht: Vorentwurf, Sichten, -skelett, HERM-Schema, Grobes HERM-Schema, Handlungen
- Feinstudie: Konzeptioneller Entwurf, HERM-Sichten, Phys./Log. Datenentwurf, Programme
- Aktionsentwurf: Konzeptioneller Entwurf, HERM, Entwurf
- Grobentwurf: Konzeptioneller Entwurf, HERM-Sichten, Phys./Log. Datenentwurf, Programme
- Sichtenintegration und -kooperation: Integrierte DBMS, Data Dictionary

mit ist eine Abbildung der Algebren der verschiedenen Schichten effektiv berechenbar.

**Literatur**


[Sch96] B. Schewe, Kooperative Softwareentwicklung, Deutscher UniversitätsVerlag, Wiesbaden, 1996


Human-Computer Interaction in Heterogeneous and Dynamic Environments: Requirements and Conceptual Modelling

Jana Lewerenz*

Abstract

*Supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).
# Contents

1 Introduction 4

2 Requirements for Interaction 6
   2.1 Scenarios to Identify Potential for Change 9
   2.2 Factors Influencing Interaction 11
   2.3 Components of Interaction 14
   2.4 Requirements for Interaction in Heterogeneous and Dynamic Environments 16

3 Existing Approaches 17
   3.1 Architectures for Interactive Systems 17
   3.2 Prototypes 21

4 The ACE Framework 25
   4.1 Overview of the ACE Framework 25
   4.2 The Interaction Model 27
   4.3 The Evaluation 37
   4.4 The Presentation 38

5 The CoDesign Prototype 41
   5.1 Introduction into the CoDesign Project 41
   5.2 Using The Driver Concept for Interaction Presentation 44

6 Summary 46

A A Syntax for the Conceptual Modelling of Interaction 47

References 52
List of Figures

1  Start Page of Cottbus.de ............................... 7
2  City Walk of Cottbus.de ................................. 8
3  Factors in Interaction .................................... 12
4  Components of Interaction ............................... 14
5  The Seeheim Model ...................................... 18
6  The Model-View-Controller Paradigm .................... 19
7  The PAC Architecture .................................... 20
8  The ACE Framework ..................................... 26
9  Meta Model for the Abstract Interaction Specification . 28
10 Example for Interaction Data ........................... 29
11 Connecting Processes .................................... 32
12 Example of an Interaction Specification ................. 36
13 Some Visual Variables ................................... 39
14 Architecture of the CoDesign Prototype ................ 43
15 A Tcl/Tk Interface Created by the CoDesign Prototype . 46

List of Tables

1  Prototypes and the Requirements for Interaction .......... 24
2  Translations for Graphical Interfaces ..................... 40
3  Translations for Language Interfaces ..................... 42
1 Introduction

Interaction is one of the most prevailing concepts in every-day behaviour. It involves several interaction partners who mutually or reciprocally act or influence each other [Col91]. Interaction basically results in the widening of one’s ‘scope of power’, enabling a person to enact activities previously out of his/her reach, to exploit external sources of knowledge, etc.

Human-computer interaction (HC interaction) refers to the interaction between a computer application system and its human users. It is to provide human users with access to the systems functionality and data, e.g., it can be deployed to control the system behaviour and/or to retrieve, store and manipulate data.

What this work is about is the conceptual modelling of HC interaction. Efficient HC interaction has been identified as one of the key elements for user satisfaction and the acceptance of a system [CL99, Dow91]. What efficient HC interaction is, however, is difficult to decide and even more difficult to put into practice. It depends on the characteristics of the user to be addressed, on user behaviour and purpose, on the interaction medium and the interaction channel available and on stylistic defaults. In this work we identify also factors in interaction, discuss their influences and provide a framework that enables designers to define and achieve a high degree of flexibility in HC interaction.

User satisfaction and heterogeneity. Simply spoken, user satisfaction means that users can do what they want and how they want it. To be honest, user satisfaction is not directly critical for every application. In cases where users are simply required to work with a particular system as it is (e.g., office automation) or where no competition between system exists, designers can afford not to bother about subtleties of user preferences, etc. This is not the kind of applications we are interested in, though.

We are interested in applications where user satisfaction is critical. Of course, this can also be the case when managers are worried about the overall happiness of their employees and require high usability for their office automation software too. But above all, user satisfaction is critical whenever customer numbers are essential and a choice between systems exists [FL99]. Common examples are e-commerce applications, city and tourist information systems or any systems that are financed by advertisement.

User satisfaction is achieved by

- providing the required functionality,
- offering relevant and complete information (data) and
enabling appropriate and efficient interaction.

In this work we concentrate on the last of these aspects, namely interaction. Interaction needs to be *appropriate* in so far as it must correspond to the user's needs, requirements and expectations in terms of personality and purpose as well as technical equipment; it needs to be *efficient* in so far as it must allow the user to achieve his/her goals as simply and quickly as possible. Appropriateness and efficiency of interaction highly depend on the interaction partner to be addressed, on the media and interaction channels available and on constraints defined by stylistic defaults (style guides)—in short: on the *interaction environment*.

This is where the aspect of heterogeneous and dynamic interaction environments comes in and makes everything even more difficult. *Heterogeneous* environments are characterised by a vast amount of

- different interaction media and channels, e.g., graphical and textual web browsers, palm tops, speech interfaces, etc.,
- different users with varying expertise, interests, preferences, possibly physical handicaps, etc., and
- different stylistic defaults that transport corporate identities, etc.

To make matters worse, today's information landscape is *dynamic*, i.e., it is changing rapidly and it is close to impossible to predict interaction media and user profiles that will be prevalent in 5 years. Even corporate identities change—due to fusions of companies, image changes, market transitions, etc.

**Meeting the challenges.** Summing up we have to ascertain that interaction is a critical aspect for many applications but its appropriateness and efficiency is dependent on the interaction environment in question. The existence of heterogeneous and dynamic environments, however, poses serious challenges to the *productivity* of interaction design and implementation. Therefore, interaction designers need the possibility to provide interaction mechanisms that are flexible enough to address several interaction environments and that can moreover be easily adapted to changes of interaction environments in the future.

This work presents such a framework. The framework structures the process of interaction design into several phases. It also allows designers to device an abstract, i.e., platform-independent, interaction specification, to determine relevant factors in interaction and to define their eventual influence on the different components of interaction.
Structure of this article. The remainder of this article is divided into four parts:

Section 2 defines the requirements of interaction. It first uses a small case study to point out how the interface design and the dialog structure of an existing city information system could (and should) be customised for different interaction situations. We then derive a classification of factors influencing interaction and roughly define the components of interaction that need to be considered for a customisation.

Section 3 describes how the problems of customised interaction have been tried to address in existing approaches. It points out valuable ideas and starting points as well as deficiencies of different system architectures and prototypes.

Section 4 then presents the ACE framework for the modelling and generation of flexible HC interaction in a productive way. This framework incorporates the phases and components of interaction design. These are described and illustrated and the use of the framework by application and interface designers is discussed.

Section 5 describes the CoDesign prototype which currently realises one of the phases defined by the framework, namely the presentation. In the CoDesign prototype, presentation is achieved by an automatic translation of the abstract interaction specification into concrete constructs of a particular target interface system. The translation is based on a structural and semantical description of the information participating in interaction. It also considers characteristics of the user and stylistic defaults.

2 Requirements for Interaction

In this section, we will examine the web pages of the city information of Cottbus. In a number of scenarios, we will discuss how the existing design could or should be adapted in the presence of other factors. For that purpose, we will explore the impact of different interaction media, channels, user characteristics and behaviour and stylistic defaults.

Based on this discussion, we then provide a classification of influencing factors and define the components of interaction that are possibly subject to adaptations.

The city information system of Cottbus. Figures 1 and 2 show some web pages of the Cottbus city information system. The start page (Figure 1) provides entry points for different user classes: tourists (Kultur & Tourismus), citizens (Bürger) and business people (Wirtschaft). It also
offers shortcuts to a news section, a calendar of events and a site map. The colour code (green for tourists, red for citizens and blue for business people) is used throughout the system.

![Start Page of Cottbus.de](image)

**Figure 1: Start Page of Cottbus.de**

Figure 2 shows a web page that offers an online ‘city walk’ through downtown Cottbus. It contains a number of photographs of prominent buildings and accompanying explanations. We can see the use of the colour code (green for tourist information) as well as the basic layout of the web pages: The left part of each page provides information on the current position and some basic shortcuts: one step back in the navigation hierarchy, back to the start page and back to the contents of the respective main category (in this case to
the tourist category). The top part offers navigation to other subcategories. This layout is used throughout the system.

Figure 2: City Walk of Cottbus.de

The city information system presented is only one of currently three online systems. Competition between systems is high. User numbers are essential in order to justify their funding and to raise new fundings. Other information providers provide access to city information via TV set top boxes and videotext.
2.1 Scenarios to Identify Potential for Change

We have pointed out that user satisfaction is the key to successfulness for many application systems. We have also pointed out that more and more application systems have to cope with the existence of heterogeneous and dynamic environments. In these environments a large variety of users, technical equipments, preferences and style guides exist—each posing different requirements at interaction.

In the following, let us take the perspectives of only some of these users. Interface designers will be well aware of such situations and the problems raised. It is important, however, that application experts too are sensitised for these issues. A few of the following situations are intuitive, others I have encountered myself or they have been pointed out by colleagues and friends.

Scenario 1 – I don’t have a computer but I have videotext! At my friend’s I had a look at cottbus.de and I really liked the kind of information offered there. At home, however, I don’t have a computer. All I have is a TV set with videotext. On videotext, there is city information too, but that is not the kind I’m interested in. If other providers use videotext, why can’t you?

I understand that web browsers and videotext are two very different things and that interfaces which work on the one system don’t work on the other but have to be re-implemented. The content is all there, though. Couldn’t you just sort of translate this content for different media?

Scenario 2 – I live at the other end of the world! I’m a fan of old technology and I have heard about Cottbus housing a number of old mills. I really want to visit this city and I want to find out some information about it beforehand. However, I live far away, bandwidth is low and it’s really painful to put up with such long loading times. I usually switch off automatic image loading but I can’t even enter your site without being required to load the navigation map on the start page.

Of course, images make a web site more attractive but it’s difficult for me to afford this. Couldn’t you offer a choice between image-based and text-based interaction?

Scenario 3 – I’m 50 years younger than my great-aunt! Why do I get the same information as she does? I’m really not interested in concerts with stars from the fifties. And my great-aunt—although I really love her—uses the offering of any punk rock concert to complain once again about the decadence of modern times.
I know that you use this huge database to extract and provide information. But I also know that many information providers use user profiles to collect information about their customers. Couldn’t you use such kind of information to customise the content of your offerings?

**Scenario 4 – I’m red-green colour-blind!** I was told that your site uses a colour code to enable better orientation within the site. This is a great idea but unfortunately you use red and green and I can’t distinguish these colours very well. Occasionally, when you cross-reference between categories, I get confused and don’t know anymore where I am.

I guess these colours are probably hard-coded in your design and might have been chosen because of some aesthetical considerations. But they are just colours and are of no use to me. Couldn’t you substitute these colours by others?

**Scenario 5 – I don’t want to browse but to book!** I have visited cottbus.de many times before and I’m very familiar with the content provided. Sometimes all I want is to book tickets for a show but every time this is the case I have to navigate through the predetermined structure of your site. This is well as long as I’m looking for information but now I’m in a hurry.

Okay, this is the way you have designed the web site to be used and it will be the usual way in many cases. But it’s not what I want all the time. Couldn’t you give me the possibility to use your site in different ways?

**Scenario 6 – I’m the boss and I want to change the layout!** We have decided to change the public image of our city and anyway, I’m tired of the old layout. No more green because of parks but silver and grey because of technology!

Don’t tell me how much effort this will involve. It’s just a couple of colours and pictures and navigation structures. Make it feel different!

- Make it feel different!
- Couldn’t you give me the possibility to use your site in different ways?
- Couldn’t you substitute these colours by others?
- Couldn’t you use user profiles to customise the content of your offerings?
• Couldn’t you offer a choice between image-based and text-based interaction?

• Couldn’t you just sort of translate this content for different media?

Couldn't you do many more things to customise interaction, to satisfy users and to increase user loyalty? The above examples are only a tiny fraction of possible requirements. They illustrate, however, the complexity of interaction and hint at the amount of effort required to cater to all these different requirements that might not even be known at the time of development.

Requirements for interaction. In order to be effective, interaction needs to be flexible and to consider as many aspects of the interaction environment as is possible and feasible. Moreover, it has to do this in a productive way. It is certainly not productive to manually design a number of different interaction mechanisms because this would involve enormous resources and could still be rendered obsolete as soon as new user types are discovered or new interface systems or styles are to be introduced.

What is indeed required is the possibility

• to automatically provide customised interaction and

• to find solutions for keeping maintenance costs low.

In the remainder of this section we will classify the factors influencing interaction as well as the components of interaction that can be subject to customisations. In Section 3 we will then present a number of existing approaches to these problems and point out their advantages and shortcomings. Section 4 introduces a model that can be used to provide flexible interaction in a productive way.

2.2 Factors Influencing Interaction

In the previous section, we roughly suggested four groups of factors that can influence the way interaction is performed:

• user characteristics,

• usage behaviour,

• technical equipment and
• stylistic guidelines.

We will now develop a more detailed picture of these factors. Figure 3 shows an overview.

![Interaction Diagram]

**Figure 3: Factors in Interaction**

**Social factors.** The social factors in interaction include the ‘permanent’ characteristics of the interaction partner and more general cultural conventions. *Characteristics* of the interaction partner are numerous [Alt94, DMKS93]. Following are only some common properties:

• interests, age, social status,

• level of expertise (with computers in general and the application in particular),

• known languages,

• cognitive abilities,

• physical handicaps, etc.

Cultural *conventions* are something like protocols that have to be followed in interaction. Often they originate from the interaction partner’s geographical and educational background. Conventions determine expectations in terms of interaction structures (e.g., greetings are more or less essential in different cultures), terminology, use of metaphors, etc.
Usage factors. In addition to ‘permanent’ user characteristics, interaction partners also differ in the way they use a system in a particular session. The goal of interaction captures the purpose of the current visit to a system, e.g., browsing, buying, contact, etc. The history of interaction describes what has happened so far, which information has already been exchanged, which navigation patterns the user has exposed, etc.

Technical factors. Technical factors refer to the equipment and the infrastructure available. The interaction media describe the interface system to be addressed, e.g., graphical or text-based web browsers, speech interfaces, cellular phones, etc.; they also describe auxiliary software like plug-ins, etc. The interaction channel can be characterised by its capacity, i.e., bandwidth, by its reliability, etc.

Stylistic factors. Stylistic factors determine the appearance of interaction, often by constituting certain style guides. Style guides can be used to convey corporate identities or simply to propagate a consistent look and feel. Concrete style guides depend on the interaction medium in question. In graphical environments they can control the arrangement of interface elements, colours, icons, fonts, etc. In natural language interaction they can determine the verboseness of utterances, the terminology to be used, etc. It is obvious that stylistic factors may interfere with user requirements as is for instance the case in the colour code used by the Cottbus web site that excludes red-green colour-blind users.

Gathering and managing influencing information. How information on users, their behaviours, media and styles is gathered and how information on any of the influential factors is managed, is actually beyond the scope of this article. A few comments, however, seem to be appropriate.

Information can be gathered by direct or indirect methods. Direct methods are to ask questions, consult environment variables of the computer system or check user settings. Indirect methods are based on observation and reasoning on these observations, e.g., observing a low typing speed and inferring that the user is not familiar with the use of computers. Indirectly gathered information is less reliable but can still be useful.

A common approach to store and manage information on users are stereotypes—a paradigm that has been developed in the seventies [Ric79] but is still widely used [CCC96]. Stereotypes are a quantitative model [DMKS93] because they consist of a explicit collection of properties and their values. Qualitative models are, e.g., overlay techniques where the properties
of users are represented as diversions from a certain standard user. Modelling techniques that are employed for user characteristics can also be used for capturing and managing information on user behaviour, equipment and styles.

### 2.3 Components of Interaction

The scenarios presented in Section 2.1 cannot only be used to motivate a number of aspects that influence interaction. They also help to identify the components of interaction that can be customised. We suggest the following (cf. Figure 4):

- the content of interaction,
- the flow of interaction and
- the form of interaction.

![Interaction Diagram](image)

**Figure 4: Components of Interaction**

The **content of interaction**. The content of interaction consists of the information to be exchanged, i.e., information to be provided to the user and information to be extracted from the user. It is defined by the type of information (e.g., a certain data class or view definition), the amount (e.g., the building of subtypes or vertical or horizontal selections) and the values (e.g., a concrete data object or the result of a view).

The content of interaction can be influenced by the interests of users, the interaction history and the interaction medium and channel (e.g., the substitution of pictures by text or the splitting of information into smaller units for small-sized screens).
**The flow of interaction.** The flow of interaction describes the sequence in which this information is exchanged. When information is split, the interaction flow also involves the ordering and control of the resulting information components.

The flow of interaction can above all depend on the interests and goals of users but also on cultural conventions, the user’s cognitive abilities, etc.

**The form of interaction.** The form of interaction concerns the appearance of the exchanged information, the constructs and properties used for their physical representation and their arrangement.

In the first place, interaction form is dependent on the constructs (e.g., graphical widgets) and properties (e.g., monochrome vs. colour displays) that are available on a particular interaction medium. In addition, it is influenced by the defined style guides, by existing cultural conventions and by personal preferences of the user.

**Considering customisation.** It must be the aim of interaction designers to provide a customisation of all of these interaction components. The most sophisticated form of interaction is of no use when an inappropriate amount of content is chosen and the user is overwhelmed by unwanted or even irrelevant information, or when an interaction flow is used which the user is unable to follow.

Let us look at one of the scenarios presented in Section 2.1 again. Let us think about the changes necessary to satisfy the user in *Scenario 1 – I don’t have a computer but I have videotext!*

- First of all, videotext uses different constructs than, e.g., a graphical browser. For the display of text for the Cottbus city walk (cf. Figure 2) in a graphical browser we could use the following HTML fragment:

  \[
  \langle P \rangle \langle B \rangle \text{Altmarkt}: \langle B \rangle \langle /P \rangle \\
  \langle P \rangle \text{Einst wichtigster Handelsplatz} \\
  \ldots \\
  \text{in dem historischen Ambiente untergebracht}. \langle /P \rangle
  \]

  For the display on videotext the generation of a vts file containing a number of text or pseudographic characters and control characters is required:

  \$1d504\text{Altmarkt}: \ldots \quad (1) \\
  \$1d50C\text{Einst wichtigster Handelsplatz} \ldots \quad (2)\text{\textsuperscript{1}}

\textsuperscript{1}The control characters—here at the start of each line—are used to change background
Obviously, different translations are required for representing the same information content.
⇒ Customisation of interaction form

- Videotext and HTML have different requirements regarding screen space. In Figure 2 we can see that a long text is shown which can be browsed using a scroll bar on the right-hand side of the screen. This is not possible in videotext because here space limitations are much stricter. Information content has to be broken down into smaller components each fitting onto one screen and navigation between these components must be enabled.

What is important is that relations between text parts must still be retained. E.g., in the city walk we first visit the Old Square (Altmarkt) and then pay attention to individual sites around that square, e.g., a fountain (Marktbrunnen) and a pharmacy museum (Apothekenmuseum). Besides the fact that this main-components relation is not very obvious on the web page either, on videotext such a structure should result in a hierarchical menu tree instead of a simple sequence.
⇒ Customisation of interaction content and flow

- Another difference between graphical web browsers and videotext is the impossibility to produce anything else but extremely iconographic pictures on videotext. Consequently, on videotext the photographs shown in Figure 2 should be substituted by text that explains some of the architectural features of the shown buildings, etc.
⇒ Customisation of interaction content

2.4 Requirements for Interaction in Heterogeneous and Dynamic Environments

We can now sum up the requirements that have to be met for an efficient human-computer interaction in heterogeneous and dynamic environments, i.e., environments that have to consider a large number of influencing factors and that are subject to unpredictable changes of these factors in the future.

Model all components of interaction. It is essential that all components of interaction, i.e., content, flow and form, are explicitly modelled in such a way as to offer guidance in later stages of detailed design and implementation. Note that an automatic generation of interfaces on the basis of such models is often desirable.

and font colours, etc.
Allow for a customisation of these components. Different user requirements in term of personal characteristics as well as technical equipment call for a customisation of interaction. Ideally, customisation is to be enabled for all interaction components.

Support modifications and extensions. The fact that interaction environments change and that it is difficult to predict future user profiles, equipments and style preference makes it important to consider an easy re-design of interaction. Above all it must be considered unproductive to manually design and implement interaction in its entirety. On the one hand, this is a costly process; on the other hand, consistency of interaction is difficult to maintain.

In the following section we will explore how existing approaches have tried to achieve customisation of interaction. Our own proposition is then put forth in Section 4.

3 Existing Approaches

Human-computer interaction is a decisive criterion for the successfullness of application systems. This has already been recognised and has been acknowledged by much research which has resulted in the development of a number of interactive system architectures and prototypes. We will explore some of the most prominent examples of these achievements.

3.1 Architectures for Interactive Systems

Some of the best-known interactive system architectures are the Seeheim model, MVC and PAC. We will present and discuss these in the following.

The Seeheim model. The Seeheim model (cf. Figure 5) was proposed on a workshop in 1985 [PH85] and can be considered as the first widely accepted interactive system architecture. It describes the constituents logically required to realise the run-time part of a user interface management system (UIMS).

- The presentation component is responsible for the immediate interaction with the user. It describes the HC interface on a lexical level by specifying the interface objects available. By means of these interface objects, the presentation component provides output to the user and accepts input from the user.
The 

- **dialogue control** implements the communication between the presentation component and the application interface. On a syntactical level, it describes the structure of interaction and the behaviour of the interaction objects.

- The **application interface** establishes the link between the HC interface component and the application system this giving access to the semantical level, i.e., the functionality of the application. It captures the purpose of an interaction within the actual application context and passes interaction requests from either side to the other.

- The **feedback** is a direct pipe from the application interface to the presentation component. It can be used to bypass the dialogue control when large volumes of output data are to be passed to the presentation component or when direct semantical feedback from the application system is required.

The Seeheim model identified interaction as a self-standing component of an application system and enabled designers to explicitly decide which information to exchange in which order and in which presentation.

For our purposes, however, the Seeheim model does not go far enough since the main part of the interaction semantics is still hidden within the application system. The fact that the feedback component was introduced to bypass the essential part of the model already suggests that the breaking of this architecture is indeed required in some cases.

Our main criticism is that—although it appears principally possible to define different behaviours and presentations for individual application objects—the decision of which information objects are when exchanged is hidden entirely in the application system itself. What we need is a model of interaction that is able to span all components, i.e., interaction content and flow and form.
The Model-View-Controller-Paradigm. The Model-View-Controller (MVC) paradigm has been developed for the Smalltalk programming environment [KP88]. It is illustrated in Figure 6.

![Diagram of the Model-View-Controller Paradigm]

**Figure 6: The Model-View-Controller Paradigm**

- The *model* is an object or a collection of objects that represent the application domain of the HC interface. It is abstract and is used to maintain object instances.

- The *view* is responsible for the display of object instances to the user. It is either notified if changes to the object instances occur or it checks the model to detect changes.

- The *controller* manages the input from the user. It notifies the model and the view of any changes to the object instance that result from the user’s actions.

The MVC paradigm quite elegantly implements the distinction of application concerns (the model) and interface concerns (the view) without completely disintegrating the design aspects of overall behaviour and interactive behaviour.

MVC explicitly allows to connect one model with several views so that different presentations of the same object instance are possible. This enables an adaption of the interaction form, but still not of the interaction content and flow. Moreover, it is difficult (just as is the case for the Seeheim model) to productively support an extension of interaction environments. In such a case, new views have to be defined for each application object which is a lengthy and costly process and moreover prone to the introduction of inconsistencies in presentations.
The PAC architecture. PAC has been developed as an object-oriented model for dialog design [Cou87]. Its main constituents are presentation, abstraction and control as shown in Figure 7.

- The *abstraction* defines the semantics of the application's interactive behaviour. It corresponds to the model component of the MVC paradigm and is used to maintain instances of application objects.

- The *presentation* describes the interactive behaviour of the application as it is perceived by the user, i.e., provides presentations of object instances to the user and allows for manipulations on those objects.

- The *control* ensures consistency between the abstraction and the presentation components. It notifies either part if changes have occurred in the other. The control can also maintain contextual knowledge like the history of user input, explanatory information, customisation rules, etc.

In PAC, the application semantics and actual presentations are strictly separated. It is possible to change either the abstraction or the presentation without affecting the other, e.g., a change in the internal structure of an object does not need to have an effect on its presentation and presentations can be changed while the abstraction is not. The PAC architecture appears very similar to the MVC paradigm. It vitally differs, however, from MVC since it unites the provision of information to and the elicitation of information from the user in the presentation component whereas MVC splits these aspects into the view and the control components. Moreover, consistency between the abstraction and the presentation is in PAC guaranteed by the control component whereas in MVC it is left to the programmer.

Although there is no explicit reference to the possibility of assigning several presentations to one abstraction, PAC appears to be able to cater for such a situation so that it is at least theoretically possible to provide different presentations in dependence on the current interaction environment. As with MVC, however, a customisation of the flow and the contents of interaction is
not explicitly considered as part of the interaction semantics. These aspects have to be dealt with in the overall system functionality. The handling of dynamics in the interaction environment, i.e., the introduction of a new interface paradigm to be addressed, etc. involves an approach similar to that with MVC: New presentations have to be defined for each object requiring interaction. As this is mainly done manually, consistency of presentations is difficult to maintain.

3.2 Prototypes

In the following we are in particular interested in approaches that have tried to take factors of the interaction environment into account in order to customise interaction.

**NOODL.** The NOODL language adopts a unifying view on application (database) objects by associating data objects and interface objects [MKB95]. Data objects describe the structure of database objects and possible operations on them; interface objects then provide features relating to the interactive behaviour and visual representation of these objects, e.g., shape, position, colour, etc.

Interfaces are generated on the basis of these features. Triggers can modify values of interface object properties so that a customisation of interfaces at system run-time is possible.

The NOODL approach has been taken a step further in the provision of a framework for visual interfaces to databases [MKB96]. In this framework not only heterogeneity with regard to the user but also with regard to the database is considered. In this framework, a visualisation component is responsible for providing information to the user. It distinguishes between the aspects of contents to be presented (referent), of symbols used for visualisation (metaphor) and of positioning different interface components (layout). The interaction component deals with the acceptance of user input and comprises the aspects of activities to be performed (intention), of mechanisms offered for performing these activities (medium) and of results of activities on any component of the entire system (effect).

**Modelling of components:** The NOODL language (and its extending framework) captures well the content of interaction. It neglects, however, interaction flow. Interaction form is specified with regard to graphical interfaces; modelling in terms of shape, colour, etc., excludes other interaction paradigms like natural language interfaces and even command line interfaces.
**Customisation:** A customisation of interaction content and flow is not considered. The form of interaction can be changed thanks to the use of triggers that modify interface object properties at system run time.

**Modifications and extensions:** Changes in the interaction environment are not anticipated. Even new requirements with regard to the interaction form involve the redefinition of the respective triggers. It is obvious that this can only be achieved by means of a re-engineering and re-design of the entire interaction component.

**Catarci et al.** Multiparadigmatic access to databases is the theme of the work of Catarci et al. [CCM$^+$96, CCC$^+$96]. In their approach, the underlying database is modelled by a unifying Graph Model [CT95]. Application objects involved in interaction are then associated with different visualisations (e.g., diagrammatic, iconic and virtual reality).

These visualisations are generated by algorithms on the basis of the graph model describing the database, i.e., the nodes of the graph and the constraints associated to the graph are mapped onto visual presentations and relations between them. Depending on their characteristics, users are offered one of these interaction styles or can select one according to their own tastes.

**Modelling of components:** Catarci et al. model the interaction content by means of a graph model. The modelling of interaction flow is only considered in terms of query formulation. Information on the interaction form is partly expressed by means of the graph structure, to a large degree it is indirectly implied in the translation algorithms (which 'know' that such and such a node is to be presented in such and such a way).

**Customisation:** Customising interaction content and flow is not possible. The form of interaction can be customised in so far as different visualisation algorithms are defined.

**Modifications and extensions:** Changes in the interaction environment can be responded to by modifying existing translation algorithms or defining new ones. The ease of use of re-defining algorithms depends on their modularity, i.e., the degree to which they separate their knowledge on what to represent how from the application of this knowledge. If programmers have ensured that it is relatively easy to exchange individual mappings (e.g., to exchange icons, phrases, colours, arrangements, etc.), then a good support of modifications and extensions of the interaction environment is possible.
Díaz et al. The work of Díaz et al. concerns the support of dynamic displays using active rules [DJPQ94]. Interaction content is modelled by means of objects (and object classes, respectively). Objects are then associated with representations.

The particularity of this approach is that active rules are used to notify the presentation component of modifications of concrete objects and also of modifications of object classes (e.g., attribute extensions, etc.). The presentation component can then customise the respective visualisation of the object concerned, i.e., it can modify values and/or the visualisation components themselves. To the degree where structural modifications of object classes can be foreseen, these modifications can be realised independently from the concrete application.

Modelling of components: Objects are used to model the content of interaction. What is not considered is the overall flow of interaction. The form of interaction is specified using translation mechanisms that map object structures onto representations. Just as is the case in the work of Catarci et al., these translations implicitly contain much knowledge on the actual meaning of objects.

Customisation: A customisation of contents and form of interaction is principally possible. It is not clear, however, which kinds of events are supposed to trigger such customisations. User requirements are not explicitly considered so that eventually the management of user requirements is still left to the core application system.

Modifications and extensions: Similarly to Catarci et al., the feasibility of considering changes in the interaction environment depends on the quality of the algorithms. If they implement a distinction between mapping rules and the application of these mappings, then the redefinition of these algorithms can be relatively easy.

Iconographer. The Iconographer system has been developed for the rapid prototyping of iconic representations [GWD90, DW91]. Although it is concerned with specifying the form of interaction only and assumes the specification of interaction contents and flow within the underlying application system, it offers an interesting and flexible method of controlling visual presentations.

In the Iconographer system, we define attributes of application objects on the one hand and attributes of presentations on the other hand. Moreover, mappings between these attributes are defined in an explicit way. E.g., if
an object file is linked to a particular icon, the attribute size can be used to control the ‘thickness’ of that icon.

What is basically done, is that type mappings are specified. The Iconographer system then manages type compatibility and executes these type mappings. The Iconographer system has later been extended to the Representor system that among other things provides more efficient means to specify the mappings between attributes and presentations [Gra97].

Modelling of components: Only the modelling of interaction form is considered. Content and flow are left to the underlying application system.

Customisation: Customisation of interaction form is possible. The triggering of changes is, however, also the task of the application system.

Modifications and extensions: The advantage of the Iconographer system is that the mappings are made explicit. A change of individual mappings or the definition of new mappings is, therefore, easily possible.

Summing up this section, Table 1 shows an overview of how the presented prototypes meet the requirements that were identified for interaction in heterogeneous and dynamic environments before.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Modelling components</th>
<th>Customising components</th>
<th>Modifications and extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cont.</td>
<td>Flow</td>
<td>Form</td>
</tr>
<tr>
<td>NOODL</td>
<td>+</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>Catarci</td>
<td>+</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Diaz</td>
<td>+</td>
<td>-</td>
<td>o</td>
</tr>
<tr>
<td>Iconogr.</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 1: Prototypes and the Requirements for Interaction

---

1Well solved
2Solved, but not perfect
3Not solved
4Depends on quality of defined translations


4 The ACE Framework

The following section describes the ACE framework. The ACE framework provides an organisation of the process of interaction development by defining phases as well as components of these phases.

In Section 4.1 we present an overview of the framework. A model for the conceptual modelling of interaction within this framework is proposed in Section 4.2. Sections 4.3 and 4.4 then describe how such a conceptual interaction specification can be transformed into a physical presentation.

4.1 Overview of the ACE Framework

The motivation for the development of the ACE framework was to provide a framework that allows the modelling of human-computer in such a way that it can be used for the generation of different materialisations. Materialisation will differ depending on factors of the environment as identified in Section 2.2: social factors, usage factors, technical factors, stylistic factors.

The ACE framework incorporates ideas that have also been applied in existing approaches—such as a separation of content and form and the definition of mappings between these components. The contribution of the ACE framework is that

- all components of interaction are considered,
- the independence of the conceptual model from the interaction environment is supported and
- tasks for the participation of different experts (application designers, interface designers, programming experts) are defined.

Figure 8 shows a graphical illustration of the ACE framework. It is organised along the two dimensions of abstractivity and scope.

The abstractivity dimension. The abstractivity dimension describes the phases of interaction development. It presents a 'stepping forward' from an abstract model of what to interact about over a selection of which part to realise in a specific situation to the determination of how to present this part. These three steps correspond to the phases of interaction modelling, evaluation and presentation which are further described in Sections 4.2, 4.3 and 4.4. During the process of interaction materialisation, information on the particular interaction environment is constantly incorporated to allow for a concretisation of the abstract specification.
The abstractivity dimension also illustrates a move from application-related concerns to presentation-related concerns, thus already suggesting the involvement of different experts. The abstract interaction model and the evaluation can be worked on by application designers in conjunction with interface experts. The presentation can be conferred to interface experts and programming experts alone.

The scope dimension. The scope dimension captures the components of interaction. The abstract interaction model as well as the evaluation phase have to explicitly deal with interaction content, flow and form. In the presentation phase these components are then combined into a particular interface which is set in a navigation structure (interaction flow), a set of constructs for content presentation and a set of representational (i.e., visual, auditive, tactile, etc.) properties and arrangements.

The principles of ACE. The ACE framework is based on three principles:

Abstraction means that the possibility for an abstract description of interaction is provided that does not pay attention to the technical details of the eventual physical presentation. Abstraction ensures flexibility and maintainability of design.

Abstraction is not an absolute notion. Interaction developers can determine the degree of abstraction in dependence of the particular application context. Less abstraction is achieved by reducing completeness.
Completeness means that in principle all components of interaction are considered during all steps of interaction development. Completeness supports flexibility of design.

Again, completeness is not an absolute notion. The less properties of the individual components are captured in the abstract design, the more presentational aspects must or can be fixed. The result is less complexity of and more control on the design but at the expense of flexibility for user requirements.

Explicitness means that mappings between abstract specification constructs and concrete presentation constructs should not be hidden inside translation mechanisms but should be openly specified so that easy manipulation, re-definition, etc., are possible. Explicitness ensures maintainability of design.

Realising explicitness lies to a large extent in the responsibility of programming experts. It is a decisive criterion for the quality of the translation mechanisms.

We will now in more detail describe the individual phases of interaction modelling, evaluation and presentation.

4.2 The Interaction Model

The interaction model describes the content, flow and form of interaction on an abstract level. It captures all possible and relevant properties of these components—together with conditions that refer to different user requirements and that can later be used to select appropriate parts of the specification. The specification of the abstract interaction model is in the responsibility of application designers who have to determine the interaction semantics.

Figure 9 shows a meta model\(^2\) we have developed for the abstract conceptual modelling of interaction. Its main elements are data, processes and characteristics relations. Data model the interaction content, processes the interaction flow, characteristics and relations the interaction form. A concrete syntax is provided in Appendix A.

Interaction content: Data. Data can be seen as the fundamental entities of interaction specifications. They are used to describe information that is to be provided to and to be extracted from the user; it also describes

\(^2\)The meta model is described in HERM notation [Tha99].
Figure 9: Meta Model for the Abstract Interaction Specification
data that plays a role in interaction-related data processing (computational processes).

In the syntax given in Appendix A data are modelled by types. Types are either basic or complex. Complex types are constructed out of basic types or already existing complex types. A null type also exists. Each node in a type hierarchy has an identifier and a label; the identifier is required to allow for further referencing to types by processes, characteristics and relations.

The following example shows the definition of a type \texttt{Hotel Reservation} in the given syntax (Identifiers are omitted.); a more easily readable graphical notion is provided in Figure 10.

\texttt{Hotel Reservation:}
\texttt{RECORD (Customer:}
\texttt{ \hspace{2em} RECORD (Name: RECORD (First\_Name: STRING,}
\texttt{ \hspace{7em} Last\_Name: STRING),}
\texttt{ \hspace{12em} Address: RECORD (...),}
\texttt{ \hspace{17em} Phone: NAT),}
\texttt{ \hspace{2em} Rooms:}
\texttt{ \hspace{3em} SET (Rooms: RECORD (Number: NAT,}
\texttt{ \hspace{8em} Price: REAL,}
\texttt{ \hspace{13em} Period: RECORD (Begin: DATE,}
\texttt{ \hspace{18em} End: DATE))})

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure10.png}
\caption{Example for Interaction Data}
\end{figure}
**Interaction flow: Processes.** Processes describe the flow of interaction. They connect types or subtypes (i.e., nodes within the type hierarchy) with each other.

Each process gets assigned a list of actors. In simple cases, actors might correspond to user identities. More often, however, a more complex actor model is required that is able to capture substitutions (partially or completely substituting users, e.g., during their holidays), alternatives (e.g., collections of users with equal rights) and hierarchies.

Moreover, each process has incoming data (input) and outgoing data (output). Either of both can be the null type. Processes can be either interactive or computational processes.

- *Interactive processes* are points where the application system requires the user to input data, to confirm data, to select data, etc., or where data is simply to be provided to the user.

- *Computational processes* are used to process data for the support of interactive processes.

A computational process connecting data is easy to interpret: it consumes its input data and produces some output data. In the case of missing input data (null type) it simply retrieves data from the database; in the case of missing output data, it simply stores data.

An interactive process works according to the same principle, but here the ‘processing unit’ is the user. Input data of an interactive process is consumed by the user, i.e., it is provided to the user by the system; output data is produced by the user—typically on the basis of previously provided input data—and is given back to the system. Missing input data means that the user only enters data; missing output data means that the user only consumes data without producing an output\(^3\).

In Figure 11 we see some examples of interactive processes. The process in the top right corner describes that the user produces (fills in) a hotel reservation form. He/she does this on the basis of existing input like suggestions concerning the time period and the hotel. This input information might have been computed by the system or it might originate from previous interactive activities where the user has already selected a hotel or has announced the intended time of stay.

According to the provided syntax, the input (and output) data of processes are specified in the following schema:

---

\(^3\) The terminology introduced by this interpretations which will be used in the following, is reversed to the common terminology of input and output.
INPUT (CLUSTER (⟨conditions⟩)):
  COMP (⟨conditions⟩): ⟨data⟩,
  ..., 
  CLUSTER (⟨conditions⟩):
  COMP (⟨conditions⟩): ⟨data⟩,
  ..., 
  ...

This schema is similar to a disjunctive normal form. Clusters (CLUSTER) are alternatives and each cluster consists of several components (COMP). Components, however, are not compulsory but can be excluded and included, respectively, by conditions. The selection of a particular cluster is also controlled by conditions.

Processes communicate via their participating data: If several processes use data with the same identifier three situations are possible:

- The data functions as output data in one process and as input data in the other process. The processes are then connected to each other as shown in Figure 11.

- The data functions as input data for all processes. It is then either copied and provided to all processes or it is provided to one process only based on the states of the processes, the value of the data, etc.\(^4\)

- The data functions as output data for all processes. Then all processes write this data. The data is either overwritten or the individual entries are collected\(^5\).

**Interaction form: Characteristics and Relations.** We have emphasised before that we want to model interaction form in a way that abstracts from particular interaction paradigms as, e.g., graphical interfaces. Several approaches [CCC\(^+\)96, JWZ93] have already refrained from explicitly defining presentation objects for individual application objects; instead they use translations that are based on the structural description of data and map this structure onto different presentations. The drawback of this method is

\(^4\)Such association semantics can be specified but are beyond the scope of this paper. The interested reader is referred to the work of Clauß et al. [CLS99].

\(^5\)For details, we would again like to refer to the work of Clauß et al. [CLS99].
that—since it is based on data structure only—it is not capable of capturing more subtle semantics like different priorities of data, groupings, etc.

We have established a number of concepts that can be used for an abstract semantical description of data [Lew99]. In the following we distinguish between several types of data characteristics and relations. These are assigned to data with regard to the role they play in a particular interactive process. Characteristics as well as relations can be controlled by conditions referring to different user requirements.

Data characteristics are either qualitative, quantitative, usage-related or functional.

- **Qualitative characteristics:** Data can be assigned to specific application-dependent concept categories such as, e.g., time, place, customer, etc. Principally, data types themselves can also be considered as such concept categories. Qualitative characteristics then serve as an additional categorisation.

- **Quantitative characteristics:** Data can be scaled, i.e., their correspondence to different application-independent and application-dependent criteria can be measured.
Application-independent criteria are relevance and emphasis. Relevance determines how urgently data has to be made available (visibly/audibly/...) or to be extracted. If data is made available, emphasis then controls how prominently that is done.

Application-dependent criteria are introduced according to the application domain in question. Possible criteria are age, duration, expensiveness, etc.

Quantitative characteristics can be measured absolutely (e.g., on a scale from 0 to 100) or in dependence on each other (e.g., data A has a higher emphasis than or an emphasis three times as high as data B).

- **Usage characteristics:** Data has to fulfill certain functions which are mainly information and reaction. These two correspond to the role of data as either input or output data for interactions and do not need to be explicitly specified.

Some input data, however, is not meant as mere information but fulfills a more specific role, namely the invocation of some activity on the user’s side. This is achieved by assigning the characteristic alert.

- **Functional characteristics:** As part of the interaction semantics, data retrieved from the underlying database can be customised to the user to the addressed and/or to the current interaction state. This is actually not a matter of interaction form but already serves the purpose of content adaption. Functional characteristics are included here for classification reasons. They make it possible to keep a large amount of in fact interaction-related content adptions out of the process and data modelling components.

Firstly, by means of the manipulation characteristic, data can be ‘re-calculated’, e.g., students can be automatically offered discounts, system administrators can be given starting times minus one hour to make sure they are at least somewhat in time, etc. Secondly, by means of the selection characteristic, data can be selected, e.g., only high-priced products are displayed for wealthy customers, etc. [FL99].

For some characteristics it is moreover possible to specify the two properties of priority and visibility. These properties are applicable for qualitative, quantitative application-dependent and usage characteristics (indirect characteristics).

Basically, they correspond to the quantitative characteristics of relevance and emphasis: The priority property determines how important it is to make a
particular characteristic apparent. If a characteristic is made apparent, then the visibility property describes how prominently this is done. It is obvious now that both properties are not applicable for functional characteristics and that there is little point in applying them to the relevance and emphasis characteristics themselves (indirect characteristics).

Data do not only have there own particular meaning and scaling but they also take part in relations between each other. We distinguish between rhetorical and organisational relations.

- **Rhetorical relations**: Connections between data that enable an orientation along the logical structure of interaction are called rhetorical relations.

  - *Narrative relations* are motivated by research on inner-textual structures [MT87]. They describe the inner structure of data that the system provides to the user. Typical examples are elaboration, context, help or summary. Narrative relations are not restricted to single interactive processes but can also be used to initiate further contributions from the system. Help data or summaries may, e.g., not be displayed in the first place but may be provided only on the user’s request or after an analysis of the user’s behaviour. In any case, the logical connection between the ‘core’ data and narratively related data must be appropriately conveyed.

  - *Conversational relations* are established between input and output data of interactive processes, i.e., between data the user is provided and data the user is expected to give. Motivation for such relations has been drawn from conversational analysis [Lev83] and the study of dialog acts [ABF97]. The main use of conversational relations is the direct support of the user’s input. Typical examples are suggestion, default, choice and again help. In the cases of suggestion and default the user is suggested a potential input value or a set of potential input values and can accept that suggestion, choose a suggested value or insert a new value, respectively. In the case of the choice relation, a discourse obligation (integrity constraint) is expressed since the user is confined to select a value from a given set.

- **Organisational relations**: The arrangement of data can be determined by organisational relations which can be either application-dependent or independent.
Application-dependent are relations like ownership and time of departure. They capture a certain ordination between data and exactly specify the type of relation between them.

More universal and less concerned with semantic subtleties of the respective application domain are application-independent relations as, e.g., grouping and sequence. Instead of exactly stating the kind of relation between data, they simply specify that several data ‘belong’ together or that they are to be provided or elicited in a particular order. Organisational application-independent relations are also termed hierarchical relations since they can be specified in a recursive fashion.

Similarly to characteristic, priority and visibility properties can be assigned to all types of relations.

In Figure 12 we show an abridged example of an abstract interaction specification in a graphical notation. The example corresponds to a situation where the user is given the possibility to fill in a request form Request form while browsing the hotel section Hotels of an information system. First of all, help information Help is provided. The individual components of this data type are linked to the components of the request form by a help relation. Moreover, the components of the request form are associated with emphasis characteristics (E), relevance characteristics (R), default relations and elaboration relations (Elab). In the case of the emphasis characteristic of the price component we have illustrated the use of conditions: For student users emphasis is set to a high value, for other others to a lower value. We also have included a computational process computation that is responsible for the calculation of price categories which are then used as suggestions for the price component of the request form. Again, different categories can be calculated based on the status of the user (the respective conditions have been omitted here).

Customisation of interaction. We have emphasised that the possibility to customise interaction is essential for its successfulness. In the following, we describe how the foundations for customisability are laid in the abstract interaction specification.

Interaction content:

- Different types of data can be connected to interactive processes by the specification of different clusters.
• Within clusters the amount of data can be modified by including/excluding different components. This corresponds to the building of subtypes.

• Values of data can be customised by the use of functional characteristics that either ‘re-calculate’ values or select values.

Interaction flow:

• The use of different data cluster also influences the flow of interaction since the data making up different clusters can again be connected to different processes.

Interaction form:

• Conditions for characteristics and relations determine their applicability for certain interaction situations.

The conditions that control input data and output data as well as characteristics and relations refer to the factors of interaction that were identified in Section 2.2. At this point we want to distinguish these factors into interaction-related and presentation-related factors:

36
• **Interaction-related factors** are those aspects that affect the organisation of interaction, i.e., the abstract picture we device of what is to be communicated. These are usage factors (goal and history) as well as some cultural conventions and characteristics of the interaction partner (e.g., conventions concerning the structure of interaction, user interests and background knowledge).

• **Presentation-related factors** play a role in deciding on the eventual appearance of interaction. These are technical, stylistic and some social factors that concern presentational metaphors and preferences/requirements of the user.

Due to the abstractivity principle of the ACE framework it is recommended that the abstract interaction specification includes only interaction-related factors. This ensures the independence of the model from the technical details of different interaction platforms. Presentation-related factors should be considered in the later phase of interaction presentation. The current computer technology makes exceptions necessary, though. E.g., since automatic translations between pictures and textual description or vice versa are not yet possible, such alternatives have to be described in the interaction specification itself.

The development of the interaction specification is done by application design experts. They are not at all concerned with presentational issues but only decide on the interaction semantics of the application.

### 4.3 The Evaluation

After the abstract interaction model has been developed, it can be used for an automatic generation of interfaces—either directly or via a prototyping phase. The automatic interface generation is carried out by the evaluation and presentation phases.

The evaluation phase is initiated whenever an interactive process is invoked. Processes can be invoked (a) on explicit user request, (b) automatically as soon as all input data are available or (c) automatically if some output data is required [CLS99]. The evaluation then performs the following tasks:

• It analyses the **current interaction situation**, i.e., it determines the user and tries to retrieve, elicit or infer information on his/her cultural background, interest, preferences, intentions, etc. Similarly, the technical characteristics can be determined.
- It then checks the conditions specified in the abstract interaction model and selects a set of relevant data (and by means of data also transitions between data that eventually form the interaction flow) as well as characteristics and relations.

- It also performs a data instantiation by initiating the retrieval of the required input data from the database and applying the appropriate relevant functional characteristics. In database-backed services, the instantiation will naturally be carried out by the application of view definitions to the underlying database [LTS99].

The result of the evaluation phase is a set if instantiated data to be provided to the user and/or a set of data (types) to be extracted from the user along with relevant characteristics and relations.

4.4 The Presentation

After the evaluation, the selected data, characteristics and relations are presented, i.e., the values of the input data are made available to the user and mechanisms for the insertion of the output data are provided.

Within the ACE framework the use of translations as has been done in many other approaches is recommended as the most appropriate. Such translations map the constructs of the selected interaction specification onto constructs and their properties of the targeted interface system. The presentation component is a combined effort of interface designers and programming experts: Interface designers decide on how to best convey the constructs of the interaction model; programming experts then put these decisions into practice. Individual translations should be extracted into external style guides (stylistic factors) as much as possible to facilitate maintainability (principle of explicitness). In that case, style guides are developed by interface designers and translation algorithms are implemented by programming experts.

Note that the quality and the degree of maintainability depends on this separation of concerns. Although such a technique can not be enforced by any methodology, the ACE framework facilitates such an approach because the explicitness of semantical data properties and relations makes such explicit mappings at all possible.

Customisation of presentations is achieved in the following way:

- **Technical factors** are considered by the mappings themselves, i.e., they determine which target constructs are available for these mappings.
- **Stylistic factors** control the mapping, i.e., they define the nature of mappings, restrict possible mappings, etc.

- **Social factors** can be used to select between alternatives that have been left open by the mappings.

In Section 5 we will present a prototype we are currently implementing that performs such a translation process. In the following, however, we will discuss some more general consideration, namely the relation between source constructs and target constructs, using the examples of a graphical interface and a language based interface. We will identify target construct and target construct properties that can be effected by translations and will also give a few examples of the influence of user characteristics.

A **graphical interface**. Let us have a look at the general components of graphical interfaces. Firstly, there are a number of widgets like text fields, various buttons, sliders, etc. Secondly, these widgets have properties like size, colour, arrangement, etc.

Based on a classification of visual variables [MS95] (cf. Figure 13), we can give the following classification of interface concepts:

- **shapes**, i.e., widgets available,
- **size**,
- **value**, i.e., ‘darkness’ or intensity,
- **hue**, i.e., colour,
- **orientation**, i.e., display angle,
- **texture**, i.e., surface appearance and
- **position**, either absolute or relative to other elements.

Table 2 gives a few examples of how the constructs of the interaction specification can be mapped onto these variables. In cases where several alternatives exist, user characteristics can be used to choose between them.
<table>
<thead>
<tr>
<th>Constructs</th>
<th>Targeted variables</th>
<th>Social factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td><em>Shape:</em> widget selection, e.g., STRING onto text boxes, etc.</td>
<td><em>User:</em> preferences for particular widgets</td>
</tr>
<tr>
<td>Qualitative characteristics</td>
<td><em>Shape:</em> icons, labels</td>
<td><em>User, conventions:</em> contexts for metaphors, terminology</td>
</tr>
<tr>
<td>Quantitative characteristics</td>
<td><em>Size, hue, value, position:</em> gradations of these properties in order to convey differences and scalings</td>
<td><em>User:</em> capabilities for visual distinction (in case of visual impairment), preferences; <em>User, conventions:</em> common metaphors</td>
</tr>
<tr>
<td>Usage characteristics</td>
<td><em>Hue:</em> dominant colours; <em>Value, position:</em> dominance; <em>Shape:</em> icons</td>
<td><em>User:</em> capabilities for visual discriminations; <em>User, conventions:</em> common metaphors</td>
</tr>
<tr>
<td>Narrative relations</td>
<td><em>Position:</em> arrangement; <em>Shape:</em> labels, icons, navigation via links, buttons, etc.</td>
<td><em>User:</em> preferences; <em>User, conventions:</em> common metaphors; <em>Also channel:</em> Avoidance of additional navigation in bad infrastructure</td>
</tr>
<tr>
<td>Conversational relations</td>
<td><em>Shape:</em> widgets that can combine input and output, e.g., checklists</td>
<td><em>User:</em> preferences for particular widgets</td>
</tr>
<tr>
<td>Organisational relations</td>
<td><em>Position:</em> arrangement; <em>Hue, value, shape, size:</em> equality of properties for groupings, gradations for sequences</td>
<td><em>User:</em> capabilities for visual distinction, preferences; <em>User, conventions:</em> common metaphors</td>
</tr>
</tbody>
</table>

Table 2: Translations for Graphical Interfaces
A (spoken) language interface. Let us next look at a spoken language interface and the possibilities offered here. First of all, natural speech can be characterised by a lot of voice facets; not all of these are, however, suitable for use in a language interface. An actor that surprises us by switching between different accents, speeds, voice pitches, etc., might be entertaining, but in a language interface this would be absolutely confusing. We will, therefore, only consider the following properties:

- **lexis**, i.e., the terminology available,
- **position**, i.e., the position of speech elements within sentences and texts,
- **volume**,  
- **melody**, e.g., the typical distinction between statements and questions by raising the voice at the end of questions, and  
- **pitch**, e.g., male vs. female voices, etc.

Lexis will be the most widely used property of language interfaces since many concepts that can be conveyed by colours, font sizes, etc., in graphical interfaces must be paraphrased in language interfaces.

Table 3 now shows correspondencies between constructs of the abstract interaction specification and these voice characteristics. Some examples of the influence of social factors are also given.

5 The CoDesign Prototype

The following section presents the CoDesign prototype—a system we are currently implementing and that is used for the automatic presentation of interaction. We will first give an overview of the CoDesign project in general and will then describe in more details the driver concept of the prototype and how it is used for automatic interface generation.
<table>
<thead>
<tr>
<th>Constructs</th>
<th>Targeted variables</th>
<th>Social factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td><em>Lexis:</em> Natural language possesses only one ‘widget’ type, namely ‘word’. So all types are mapped onto words, phrases, etc.</td>
<td><em>User:</em> prefered or known terminology as a subset of the terminology available</td>
</tr>
<tr>
<td>Qualitative</td>
<td><em>Lexis:</em> words, phrases</td>
<td><em>User, conventions:</em> contexts for the selection of terminology</td>
</tr>
<tr>
<td>characteristics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantitative</td>
<td><em>Lexis:</em> explicit paraphrasing; <em>Position:</em> order in enumerations</td>
<td><em>User, conventions:</em> terminology, common metaphors</td>
</tr>
<tr>
<td>characteristics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usage characteristics</td>
<td><em>Lexis:</em> explicit paraphrasing; <em>Melody:</em> distinction between input and output, <em>Pitch:</em> different voice pitch for warnings</td>
<td><em>User, conventions:</em> terminology</td>
</tr>
<tr>
<td>Narrative relations</td>
<td><em>Lexis:</em> explicit paraphrasing, provide means for follow-up questions; <em>Position:</em> order</td>
<td><em>User:</em> preferences (follow-up question or not); <em>User, conventions:</em> common metaphors</td>
</tr>
<tr>
<td>Conversational</td>
<td><em>Lexis:</em> explicit paraphrasing, pre-formulation of answers, etc.</td>
<td><em>User, conventions:</em> preferences, terminology</td>
</tr>
<tr>
<td>relations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organisational</td>
<td><em>Position:</em> avoid splitting of groups in speech output, order</td>
<td><em>User, conventions:</em> preferences, terminology</td>
</tr>
<tr>
<td>relations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Translations for Language Interfaces
5.1 Introduction into the CoDesign Project

The CoDesign philosophy has been developed to enable an integrated design of different components of an information system [Tha97, CT97, CLT97]. These components include:

- the database schema,
- the view schema,
- the behaviour (system functionality) and
- the dialogs (interfaces).

The CoDesign philosophy had been motivated by experiences from application design which have shown that a separate design of the individual components suffers from several gaps between specification languages [CT97] and that it possibly leads to non-optimal, inconsistent and incomplete designs.

**The CoDesign prototype.** With the CoDesign prototype we are currently implementing a tool for an integrated design of information systems and for their direct execution. The architecture of the prototype is shown in Figure 14.

The core of the prototype is the system control which reads the application specification, checks for the executability of processes and starts their execution. The process execution lies then in the responsibility of the process driver. In the current version of the prototype, the process driver is based on term rewriting.

Processes are either computational processes or converters (virtual drivers). Converters are responsible for the transformation of data storage processes or interactive processes to the respective target system, i.e., a specific data storage system on the one hand or a specific interface system on the other hand. Low-level drivers then perform the actual low-level communication with the actual target system. The architecture also provides for the incorporation of external applications to source out some of the system functionality.

The transformation of storage and interactive processes is controlled by database (DB) and user interface (UI) specifics; in the case of UI specifics, these contain the influential factors on interaction as already described. Together with the application specification, these specifics are produced with the help of tools. Design tools have not yet been implemented in the prototype.
5.2 Using The Driver Concept for Interaction Presentation

The CoDesign prototype is intended for many more tasks than interface generation. We have, however, found it convenient to use for a testing of our ideas and previous conceptualisations.

The CoDesign prototype as it has worked so far. The CoDesign prototype has not yet implemented design tools. The interaction specification must, therefore, still be hand-coded as the UPROC part of the application specification in a rather awkward format, namely term algebras according to the following grammar:

\[
\begin{align*}
\langle \text{prog} \rangle & : = \text{PROG LEFT } \langle \text{def} \rangle \text{ RIGHT} \\
\langle \text{def} \rangle & : = \langle \text{definition} \rangle \\
& \quad | \langle \text{definition} \rangle \ \text{DIVIDER} \ \langle \text{def} \rangle \\
\langle \text{definition} \rangle & : = \text{PROC LEFT } \langle \text{procname} \rangle \ \text{LEFT RIGHT DIVIDER} \\
& \quad \langle \text{tersgra} \rangle \ \text{LEFT RIGHT DIVIDER} \\
& \quad \langle \text{input} \rangle \ \text{LEFT RIGHT DIVIDER} \\
& \quad \langle \text{output} \rangle \ \text{LEFT RIGHT RIGHT}
\end{align*}
\]
The grammar given above allows only for a structural description of the data participating in interactive processes. These structural descriptions are then converted by a hierarchy of drivers into concrete interface constructs of particular target interfaces. A low-level driver finally interprets the output of this driver hierarchy and uses it to control the interface system in question. In this way, we have created UNIX command line interfaces and Tcl/Tk interfaces.

The CoDesign prototype as it is currently being modified. The syntax given in Appendix A has been developed on the basis of the grammar specified above. It refines data and process specifications and additionally introduces the concepts of characteristics and relations.

The processing of these newly introduced concepts is realised for the Tcl/Tk driver that had been developed before. The driver is still heavily under construction. The example given in Figure 15 illustrates its current abilities.

The example corresponds to the interaction specification given in Figure 12 on Page 36. Besides the generation of interface elements based on the structural description of the interaction data, the Tcl/Tk driver realises the following characteristics and relations:
Figure 15: A Tcl/Tk Interface Created by the CoDesign Prototype

- **suggestion** relations: by the simulation of a combo box construct (which does not exist in Tcl/Tk; clicking on the down arrow next to the entry field opens a list that can be selected from);

- **default** relations: by the use of default values in the respective entry fields;

- **relevance** characteristics: as seen in the distinction between the hotel field and the price field. The higher relevance of price is tried to convey by the explicit provision of a suggested value which is not done for the hotel field.

The capabilities of the Tcl/Tk driver are constantly improved. The development of further translations for other interface systems is also planned.

### 6 Summary

Efficient interaction is a decisive criterion for the successfulness of many information system applications. Especially in heterogeneous and dynamic environments methods for interaction development are required that allow
for easy customisation and maintainability. We have identified the following requirements for interaction in such environments:

- the modelling of all components of interaction, i.e., interaction content, flow and form,

- the possibility to achieve a customisation of these components to different social, usage, technical and stylistic factors and

- the possibility to enable the extension and modification of the interaction specification in the case of changes in the interaction environment.

We have found that existing approaches to the development of interaction are not able to meet all of these requirements. Common drawbacks are missing opportunities for the modelling (and thus the customisation) of the interaction flow and the lack of a throughout modelling of semantical characteristics of and relations between data that would enable a better source for the generation of appropriate presentations.

We have, therefore, developed the ACE framework which provides means for a complete modelling of all components of interaction, i.e., content, flow and form, in an abstract manner. Above all we have introduced a set of concepts for describing presentation-related semantics of the data participating in interaction. The abstract interaction specification is completed by a number of conditions. These can be used in the phase of automatic interface generation to select those parts of the specification that are of relevance in the current interaction situation. Translations that map the constructs of the abstract specification onto concrete target interface constructs and properties are then used for the eventual presentation of interaction.

The ACE framework is able to fulfill the requirements for interaction in heterogeneous and dynamic interaction environments since

- it provides a set of constructs for the complete modelling of all interaction components,

- it supports an abstract, i.e., platform-independent, modelling of interaction and instead uses conditions and mappings to translate abstract interaction specifications into presentations customised to different user requirements,

- by the explicit provision of semantical data characteristics and relations, it encourages the use of style guides that explicitly define mappings between specification constructs and target interface constructs and do not hide these mappings within complex translation algorithms.
We are currently implementing a prototype that realises an environment for the integrated development of information systems and their execution. This prototype is used for the testing of our ideas and conceptualisations. After realising an automatic interface generation based on structural aspects of the interaction data (as has been done by many other approaches before), we are currently—successfully—extending the prototype to the inclusion of data characteristics and relations.

Thanks! I would like to thank my supervisor Prof. Thalheim, the Co-Design project team and the members of the Graduiertenkolleg ‘Distributed Information Systems’ for their support and many motivating and clarifying discussions.
A Syntax for the Conceptual Modelling of Interaction

\[
\begin{align*}
\langle \text{hci} \rangle & ::= \text{HCI} \ (\langle \text{data} \rangle), \\
& \phantom{::=} \text{INTERACTION} \ (\langle \text{interaction} \rangle), \\
& \phantom{::=} \text{COMPUTATION} \ (\langle \text{computation} \rangle), \\
& \phantom{::=} \text{CHARACTERISTICS} \ (\langle \text{characteristics} \rangle), \\
& \phantom{::=} \text{RELATIONS} \ (\langle \text{relations} \rangle), \\
\langle \text{data} \rangle & ::= \langle \text{data} \rangle, \langle \text{data} \rangle \\
\langle \text{interaction} \rangle & ::= \langle \text{interaction} \rangle, \langle \text{interaction} \rangle \\
\langle \text{computation} \rangle & ::= \langle \text{computation} \rangle, \langle \text{computation} \rangle \\
\langle \text{characteristics} \rangle & ::= \langle \text{characteristic} \rangle, \langle \text{characteristics} \rangle \\
\langle \text{relations} \rangle & ::= \langle \text{relation} \rangle, \langle \text{relations} \rangle \\
\langle \text{data} \rangle & ::= \langle \text{id} \rangle: \langle \text{name} \rangle: \langle \text{type} \rangle \\
& \phantom{::=} \epsilon \\
\langle \text{id} \rangle & ::= D\#\langle \text{list} \rangle \\
\langle \text{list} \rangle & ::= \langle \text{number} \rangle \\
& \phantom{::=} \langle \text{number} \rangle, \langle \text{list} \rangle \\
\langle \text{name} \rangle & ::= \langle \text{word} \rangle \\
\langle \text{type} \rangle & ::= \langle \text{basic} \rangle \\
& \phantom{::=} \langle \text{complex} \rangle \\
\langle \text{data} \rangle & ::= \langle \text{data} \rangle, \langle \text{data} \rangle \\
& \phantom{::=} \langle \text{data} \rangle, \langle \text{data} \rangle \\
\langle \text{basic} \rangle & ::= \text{STRING} | \text{CHAR} | \text{NAT} | \text{REAL} | \text{BOOL} | \text{DATE} \\
\langle \text{complex} \rangle & ::= \text{SET} (\langle \text{data} \rangle) \\
& \phantom{::=} \text{LIST} (\langle \text{data} \rangle) \\
& \phantom{::=} \text{RECORD} (\langle \text{data} \rangle) \\
\langle \text{interaction} \rangle & ::= \langle \text{id} \rangle: \text{IN} (\langle \text{input} \rangle), \text{OUT} (\langle \text{output} \rangle), \\
& \phantom{::=} \text{ACT} (\langle \text{actors} \rangle) \\
\langle \text{id} \rangle & ::= I\#\langle \text{number} \rangle \\
\langle \text{computation} \rangle & ::= \langle \text{id} \rangle: \langle \text{input} \rangle, \langle \text{output} \rangle, \langle \text{actors} \rangle \\
\langle \text{id} \rangle & ::= C\#\langle \text{number} \rangle
\end{align*}
\]
(input) ::= ((cluster_list))
(output) ::= ((cluster_list))
(cluster_list) ::= CLUSTER ((conditions)) (comp_list)
    | CLUSTER ((conditions)) (comp_list), (cluster_list)
(comp_list) ::= COMP ((conditions)) (did_list)
    | COMP ((conditions)) (did_list), (comp_list)
(did_list) ::= (d_id)
    | (d_id), (did_list)
(actors) ::= (a_list) ? (conditions)
    | (a_list) ? (conditions), actors
(a_list) ::= (a_id)
    | (a_id), (a_list)
(a_id) ::= A#(number)

(characteristic) ::= (dir_char)
    | (indir_char)
(dir_char) ::= (ch_id): (d_id), (dch_type), (quantity),
              (context), (conditions)
(indir_char) ::= (ch_id): (d_id), (ich_type), (quantity),
              (context), (conditions),
              (priority), (visibility)
(ch_id) ::= CH#(number)
(ch_type) ::= (dch_type)
    | (ich_type)
(dch_type) ::= MANIP
    | SELECT
    | RELEV
    | EMPH
(ich_type) ::= ALERT
    | APPL:(word)
(quantity) ::= (ch_equation)
    | (ch_range)
    | (ch_comparison)
(ch_equation) ::= (che_value)
    | ((ch_equation) (f_op) (ch_equation))
(che_value) ::= SELF
    | (d_id)
    | (ch_id):(ch_type)
    | (word)
    | (number)
\[ \begin{align*}
(\text{model}) & : (\text{attribute}) \\
(\text{ch\_range}) & ::= (\text{r\_op}) (\text{chr\_value\_list}) \\
(\text{r\_op}) & ::= \text{MIN} \\
& \quad | \text{MAX} \\
& \quad | \text{AVG} \\
(\text{chr\_value\_list}) & ::= (\text{chr\_value}) \\
(\text{chr\_value}) & ::= (\text{d\_id}) \\
& \quad | (\text{ch\_id}): (\text{ch\_type}) \\
& \quad | (\text{model}): (\text{attribute}) \\
(\text{ch\_comparison}) & ::= (\text{c\_op}) (\text{chc\_value}) \\
(\text{c\_op}) & ::= \text{LESS} \\
& \quad | \text{GREATER} \\
& \quad | \text{EQUAL} \\
(\text{chc\_value}) & ::= (\text{ch\_id}): (\text{ch\_type}) \\
& \quad | (\text{model}): (\text{attribute}) \\
(\text{relation}) & ::= (\text{flat\_rel}) \\
& \quad | (\text{hier\_rel}) \\
(\text{flat\_rel}) & ::= (\text{rel\_id}): \text{SOURCE} (\text{d\_id}), \text{TARGET} (\text{d\_id}), \\
& \quad (\text{flat\_type}), (\text{context}), (\text{conditions}), \\
& \quad (\text{priority}), (\text{visibility}) \\
(\text{hier\_rel}) & ::= (\text{hrel\_id}): \text{SOURCE} (\text{part}), \text{TARGET} (\text{part}), \\
& \quad (\text{hier\_type}), (\text{context}), (\text{conditions}), \\
& \quad (\text{priority}), (\text{visibility}) \\
(\text{rel\_id}) & ::= \text{R\#}(\text{number}) \\
(\text{hrel\_id}) & ::= \text{HR\#}(\text{number}) \\
(\text{part}) & ::= (\text{d\_id}) \\
& \quad | (\text{hrel\_id}) \\
(\text{flat\_type}) & ::= \text{ELAB} \\
& \quad | \text{SUMM} \\
& \quad | \text{CONTEXT} \\
& \quad | \text{HELP} \\
& \quad | \text{SUGG} \\
& \quad | \text{CHOICE} \\
& \quad | \text{DEFAULT} \\
& \quad | \text{APPL}:(\text{word}) \\
(\text{hier\_type}) & ::= \text{GROUP} \\
& \quad | \text{SECU}
\end{align*} \]
\[\begin{align*}
\langle \text{context} \rangle & := \langle \text{i_id} \rangle \\
& \quad \mid \epsilon \\
\langle \text{priority} \rangle & := \text{P} (\langle \text{quantity} \rangle, \langle \text{conditions} \rangle) \\
\langle \text{visibility} \rangle & := \text{V} (\langle \text{quantity} \rangle, \langle \text{conditions} \rangle) \\
\langle \text{conditions} \rangle & := \langle \text{condition} \rangle \\
& \quad \mid \text{NOT} (\langle \text{conditions} \rangle) \\
& \quad \mid \text{AND} (\langle \text{conditions} \rangle, \langle \text{conditions} \rangle) \\
& \quad \mid \text{OR} (\langle \text{conditions} \rangle, \langle \text{conditions} \rangle) \\
\langle \text{condition} \rangle & := \langle \text{criterion} \rangle \langle \text{e_op} \rangle \langle \text{equation} \rangle \\
& \quad \mid \langle \text{criterion} \rangle \langle \text{s_op} \rangle (\langle \text{enumeration} \rangle) \\
& \quad \mid \langle \text{criterion} \rangle \langle \text{s_op} \rangle [\langle \text{interval} \rangle] \\
& \quad \mid \epsilon \\
\langle \text{criterion} \rangle & := \langle \text{d_id} \rangle \\
& \quad \mid \langle \text{model} \rangle : \langle \text{attribute} \rangle \\
\langle \text{model} \rangle & := \text{SOCIAL\_MODEL} \\
& \quad \mid \text{USAGE\_MODEL} \\
& \quad \mid \text{TECHNICAL\_MODEL} \\
& \quad \mid \text{STYLISTIC\_MODEL} \\
\langle \text{attribute} \rangle & := \langle \text{word} \rangle \\
\langle \text{equation} \rangle & := \langle \text{m_value} \rangle \\
& \quad \mid (\langle \text{equation} \rangle \langle \text{f_op} \rangle \langle \text{equation} \rangle) \\
\langle \text{enumeration} \rangle & := \langle \text{m_value} \rangle \\
& \quad := \langle \text{m_value} \rangle, \langle \text{enumeration} \rangle \\
\langle \text{interval} \rangle & := \langle \text{value} \rangle, \langle \text{value} \rangle \\
\langle \text{value} \rangle & := \langle \text{word} \rangle \\
& \quad \mid \langle \text{number} \rangle \\
\langle \text{m_value} \rangle & := \langle \text{word} \rangle \\
& \quad \mid \langle \text{number} \rangle \\
& \quad \mid \langle \text{d_id} \rangle \\
& \quad \mid \langle \text{model} \rangle : \langle \text{attribute} \rangle \\
\langle \text{e_op} \rangle & := = \mid > \mid < \\
\langle \text{s_op} \rangle & := \text{IN} \\
\langle \text{f_op} \rangle & := + \mid - \mid * \mid / \\
\langle \text{word} \rangle & := \langle \text{letter} \rangle \\
& \quad \mid (\langle \text{letter} \rangle) \langle \text{word} \rangle \\
\langle \text{letter} \rangle & := \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \\
& \quad \mid \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{r} \mid \text{s} \mid \text{t} \mid \text{u} \mid \text{v} \mid \text{w} \mid \text{x} \mid \text{y} \mid \text{z} \\
\end{align*}\]
\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
A & B & C & D & E & F & G & H & I & J & K & L \\
N & O & P & Q & R & S & T & U & V & W & X & Y & Z \\
\hline
\end{array}
\]

\[
\begin{array}{l}
\langle \text{number} \rangle ::= \langle \text{digit} \rangle \\
\quad | \langle \text{digit} \rangle \langle \text{number} \rangle \\
\langle \text{digit} \rangle ::= 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9 \\
\end{array}
\]
References


Natural Language Descriptions Mapped To Database Schemata
Applying a Natural Language Dialogue Tool for Designing Databases

Edith Buchholz, Heiko Cyriaks, Antje Düsterhöft, Holger Mehlán, Bernhard Thalheim

University of Rostock, Department of Computer Science
email: duest.uni-rostock.de
University of Cottbus, Department of Computer Science
email: thalheim.tu-cottbus.de

Abstract

The purpose of this paper is to present a Dialogue Tool which has to be applied in order to obtain the structure, semantics and behaviour of a database. We want to show how the knowledge of the designer can be gathered and how syntax and semantics of the natural language input can be analysed or interpreted and then transferred on to a special EER model. The transformation will yield a skeleton design including structural and semantic constructs as well as behaviour.

Resume

1 Introduction

1.1 Present state of the art and aims

The quality of database design is a decisive factor for the efficiency of a database application. A database designer has to use a high level of abstraction for mapping his real-world application onto an entity relationship model. He has to learn the model
and the constraints to use it. The bottleneck during the design process is the high
degree of abstraction and the abstract language that has traditionally been used for
this purpose. There are not many users who can learn the database design formalisms
easily, but most users are able to describe in their native language the entities that
will form the basic elements of the prospective database, how to administer them and
the processes they will have to undergo. There is a growing demand for a natural
language input or, at least, for a natural language interface that will allow the user
to present his informal knowledge in a natural language. But ways have to be found
to handle ambiguity, fuzziness and redundancy of natural language.

We have implemented a knowledge-based Dialogue design Tool (German) for getting
a skeleton design. The Dialogue Tool interacts with other components of our database
design system RADD [ThAABDS94] to form a complex tool. The designer describes
the structure of an application in German as a first step in the design process. The
specification and formalisation of semantic constraints are one of the most complex
problem for the designer. Within natural language sentences the designer uses se-
mantic constraints intuitively. For this reason we focus on extracting comprehensive
semantic information about the domain from natural language utterances. In case
of ambiguity within the RADD system the designer can validate the results of the
natural language analysis by means of other tools (e.g. a graphical editor, a seman-
tics checker). The skeleton design with the semantic constraints is also the basis for
further semantic checks, e.g. for key candidates, and it will restrict the search areas
in the checking process [AlbBDT95]. The data dictionary which contains the actually
design is usable in all tools.

The Dialogue Tool is currently implemented in PROLOG using SUN workstations.

1.2 Overview of this paper

The Introduction shows which methodologies and tools are existing that deal with
natural language supporting database design process and database queries. Starting
from this platform our approach and its aims are outlined.

In Section 2 we illustrate by example the potential of natural language which can be
exploited for database design.

Section 3 shows the prototype implementation of the Dialogue Tool. The Dialogue
together with the knowledge base will be used for drawing to the designer’s attention
special facts resulting from the syntactic, the semantic and the pragmatic analyses.
The system makes suggestions for completing the design applying the knowledge base.

Section 4 deals with the process of transforming knowledge gained from natural lan-
guage sentences into EER diagrams. Results of syntactic and semantic analyses of a
natural language sentence and common heuristic rules, together with the knowledge
base, are used for designing the skeleton and extracting information on the semantics
as well as information on behaviour of the prospective database.

Section 5 presents conclusions and considers aspects of future work.
1.3 Background

Knowledge base. [StoG88], [GolS91] give an expert systems approach to creating user views. Choosing semi-natural language the so called VCS (view creating system) engages the user in a dialogue about the information requirements using a knowledge base. [BouG83], [FloPR85] show expert systems for database design. The systems interact with the user to ask for missing information. [Tau90] presents expert systems for conceptual and logical database design which were developed as integral parts of a next generation CASE tool. In [MetML93] natural language techniques of semantic cases and conceptual graphs are used by several functions of schema manipulations.

Natural language in database design. [Che83] shows the close relationship between English sentence structure and ER diagrams. Within the system INCOD [AtzBLV83] the user can describe the application using a design language. The structure of the prospective database is developed step by step. The paper illustrates how the insertion of new EER models can be controlled to be consistent. [ColGS83] present a dialogue system for the analysis of natural language to extract knowledge from requirements description of data, transactions and events. Natural language as input is used in the project ANNAPURA [Eic84], [EicL85] for requirement specification. The system also uses questions to clarify the designer input. [RolP92] presents a Requirement Engineering support environment that generates the conceptual specification from a description of the problem area provided through natural language statements. [TjoB93] have described the development of database design and database queries using natural language (German). The main idea of the paper is that syntactic structures of the natural language can be semi-automatically transformed into EER structures using rules and heuristics. A so-called DMG (Data Model Generator) was introduced to process sentences in order to get a requirement specification.

[Ort93] and [Ort94] studies the technical language (German) which is used in working situations in order to develop a standardised enterprise language for software developement. Within these studies the relationship between natural language and elements of an object-oriented model will be analysed.

Semantics. In [TseCY92] a methodology is presented which maps natural language of example constructs into relational algebras through ER representations. The natural language conjunctives ‘and’ and ‘or’ are mathematically analysed. Recently computational linguistics has produced a series of outstanding contributions to automatic processing of natural language (e.g. in the field of semantics [Pin93]).

We maintain that natural language descriptions of the requirements of a prospective database are so complex that by skilful interaction with the user, a tool can extract not only the structure but the semantics of the database as well. We have chosen a Dialogue Tool because we are convinced that for most applications descriptions in formal complex texts are not available. Furthermore, using texts and asking the designer questions only when the system does not understand them will not stimulate him to extract knowledge of the domain.
2 The potential of natural language for database design

Let us consider a person who has used a library at least once in his or her life. To this person the sentence

Der Benutzer entleiht ein Buch mit einem Leihschein.

(The user borrows a book with a borrowing-slip.)

automatically conveys a large amount of information. Because of language as well as world knowledge the person knows that

- user and book are two partners in an act
- a certain relationship exists between these two, expressed by the verb
- the relationship is the change of possession
- the change of 'ownership' is only temporarily (in contrast to buying or presenting)
- the user is not a specific one, in fact it can be any Person that fulfils the conditions going along with this business
- a book does not mean one book, it can be two or three and more books
- a book does not necessarily have to be a book, it can be a picturebook, a journal, a magazine, an atlas
- in 'book with a borrowing-slip' with does not express possession or part of like in 'a book with pictures', it expresses instrumentality (by means of)
- the 'borrowing-slip' is not just a slip of paper but has the function of a contract
- borrowing means that the book will have to be returned
- a book is borrowed at some place (library)
- this sentence does not include details about user or book, it is a general lending process.

This complex information which a person associates when reading the sentence will be elicited by the Dialogue Tool in various steps. These steps have to comprise

1. a linguistic analysis of the natural language for syntax and semantics
2. a set of transformation rules for extracting the database structure
3. rules and heuristics for handling world knowledge
4. classification rules for modelling the behaviour

In the following chapters we will illustrate the basic transformations to achieve this aim.

As the prototype tool is based on syntactic, semantic and world knowledge we had to choose a domain as the background of the prospective database and as the object of the domain knowledge base. We decided to take the field 'library - its tasks and processes'. As a method of obtaining the linguistic corpus we decided to carry out a number of interviews with librarians as well as library users (cf. [BucD95]).
For readability reasons we show the following examples in German (the implementation language) and in English. The results of the analysis are given in English only.

3 A Dialogue Tool for acquiring knowledge

3.1 Dialogue model

For the acquisition of designer knowledge we chose a moderated dialogue. A moderated dialogue can be seen as a question-answer system. The Dialogue Tool reacts appropriately to every input sentence. It asks for additional input if it finds gaps in the presentation of database design information. These questions are frames which will be updated in the dialogue process. The designer can form the answers in natural language sentences. Figure 1 shows the structure of the analysis of a natural language input.

Within the Dialogue the results of the syntactic, semantic and pragmatic analyses will be used for controlling the Dialogue. This means, if an incomplete designer input is received a question will be initiated. Inputs are incomplete if either the results of the linguistic analysis are not complete or the design model generated so far is incomplete.

We distinguish three types of questions:

1. CQ: content question (e.g. 'Are there any more details about the application?')
2. LQ: linguistic clarification questions (e.g. 'How is the act -borrow- done?')
3. PQ: pragmatic clarification questions (e.g. 'How is -book- characterized?')

In the following sections we will show in which concrete case these questions will be initiated.
3.2 Syntactic analysis

The designer input into the dialogue tool is first submitted to a syntax analyser.

**Grammar and lexicon.** We have developed a special phrase structure grammar which uses the ID/LP format (Immediate Dependence/ Linear Precedence [Gaz85]) and includes meta rules. The grammar formalism describes a part of the German language based on analyses of user-input (c.f. also [Meh95]). The grammar analyses main and subsidiary clauses, relative clauses, prepositional clauses as well as all basic verb phrases (except extended infinitive clauses phrases). The underlying lexicon contains lexeme and morphological rules.

As a method of obtaining the linguistic corpus we carried out a number of interviews with librarians as well as library users. These talks in most cases had the nature of dialogues with lengthy monologue parts in them. The talks were tape-recorded and afterwards transcribed. The corpus obtained consists of more than 12,000 lexical units. The information gained from it was sustained by referring to the two-volume 'Lexikon des Bibliothekswesens'.

**Parser.** We have implemented a special parser which uses the grammar as well as the lexicon and transforms natural language utterances into syntax trees. A set of syntax trees is produced as a result of the syntactic analysis. All categories can have syntactic features e.g. genus or case.

3.3 Semantic analysis

Interpreting the semantics of the designer input we use the two-stage linguistic model which inserts a semantic level between the syntax level and the conceptual level (EER model).

**Meaning of a sentence.** To identify the meaning of sentences we have used the model of semantic roles. The units in a sentence or an utterance are seen to fulfil certain roles. Our role concept is mainly based on Jackendoff’s hypothesis [Jac83] and consists of the following roles which refer to the objects partaking in the action: Cause, Theme, Result/Goal, Source, Locative, Temporal, Mode, Voice/Aspect. The roles of a sentence are used to clarify linguistically completeness and to support the extraction of the design.

**Verbs.** Verbs form a central part in defining the meaning of sentences and the relationships between parts of sentences. Basically they describe actions, processes and states. We have tried to find a classification of verb semantics [BucD95] that can be applied to all verbs in the German language (and probably other languages as well). Our aim was to keep the number of classes small and fairly general but large enough to identify their function in a sentence correctly. This classification is, at this stage, independent of the domain to be analysed.

The following example shows the semantic roles of the sentence ’Der Benutzer entleiht ein Buch mit einem Leihschein.’ (The user borrows a book with a borrowing-slip.)

**Example:**

- The user borrows a book with a borrowing-slip.
- verb type: verb of movement (borrow)
- Cause (subject): the user
- Theme (object): a book
Nouns and determiners. Nouns describe objects of the real world. For database design purposes we are not interested in the nature of these objects. But we are interested in the relations which the objects have and in quantifying these relations. For analysing the determiners of the nouns we need a noun classification. We adapted the classification of [Bis91] as a first step and distinguish three types of objects: abstract, local and temporal objects. Each type can also express individuality, continuity and uniqueness (see [Cyr95]). For analysing the determiners we use the model theoretic description ([BarC81]).

Nouns and determiners are used to define cardinalities as well as inclusions and exclusions dependencies shown in Section 4.

3.4 Pragmatic interpretation

The pragmatic interpretation is part of the Dialogue Tool. The aim of the pragmatic interpretation is the mapping of the natural language input onto EER model structures using the results of the syntactic and semantic analyses. A basic feature of the pragmatics is the domain model which defines a frame for every specific application and supports the acquisition of semantics and behaviour of the prospective database. Using the results of the linguistic analysis a common domain model is gradually developed. During the design process this model can be updated. Section 4 will show how common rules are used for making general assumptions about how information gained from natural sentences is related to entities, relationships, sets, keys and other EER structures.

4 Transforming knowledge gained from natural language sentences into EER diagrams

4.1 Assumptions concerning the EER model

Due to readability reasons a section of the EER model is used in this paper to illustrate the process of transformation. (The other constructs of the underlying EER model can be modelled in the same fashion.)

The following structural constructs of EER models are the basis of our natural language transformation.

- entity(ENAME): describes an entity with the name ENAME
- relship(RNAME,ENAME1,[ENAME2]): describes a relationship RNAME between entity ENAME1 and the list of entities (ENAME2 - describes an set of according entities);
  the is-a-classification will be described as relship(is-a,ENAME1,[ENAME2]) where a ENAME2 is a ENAME1
"attre(EName,AName): the entity EName has an attribute AName"

"attrr(RName,AName): the relationship RName has an attribute AName"

Semantic constructs are key candidates, cardinalities as well as inclusion and exclusion dependencies.

- keycand(EName/RName,AName): the attribute AName is a key candidate of the entity EName or the relationship RName
- cardcand(NR,RName,EName,MinCard,MaxCard): the relationship RName has a cardinalities MinCard:MaxCard corresponding to the entity EName
- inclcand(EName1,EName2): describes an inclusion dependency of two entities (EName1 and EName2) where a EName1 includes a EName2
- exclcand([EName]): describes a list of excluded entities EName

We assume that names of entities and relationships are unambiguous. We define the model description as the representation of the presented design of the prospective database. The model description is an ordered set of facts.

4.2 Obtaining a skeleton design of the prospective database

The aim of the pragmatic interpretation of a natural language input is firstly obtaining a skeleton structure and secondly getting information on semantics of the prospective database. In this section we will illustrate how we can extract a skeleton design by using world knowledge and heuristics. This general interpretation is domain independent. We have chosen the inside-out approach [Tha93] for getting a design. That means we concentrate firstly on the main elements of the design and then we try to find details of the elements. The additional questions (pragmatical questions- PQ) for clarifications are put to the user in the same order.

The structural pragmatic interpretation consists of three steps. First the syntactic and semantic results of a natural language input are analysed and interpreted. Then this interpretation will be inserted into the actual model description, after which the new model description will be checked in order to find pragmatic incompleteness. If the model description is not complete additional questions (PQ) will be initiated. The following sections illustrate these steps.

4.2.1 Interpreting a natural language utterance

Heuristics. The transformation of the structure of natural language sentences into HERM model elements is a process which is based on heuristic assumptions. ([TjoB93] illustrate a large number of such heuristics in an informal way.) These heuristics illustrate the close relationship between words/phrases and model elements. We have defined and formalised such heuristics using contextfree and contextsensitive rules. Figure 2 shows some of these rules. (The ‘$’ indicates an element which is needed. An special question will be initiated in order to obtain this element.)
Attribute Grammar. The result of the syntactic analysis is a set of syntax trees of
the natural language input. We see a syntax tree as a tuple structure and the set of
possible syntax trees as a language. Now we handle the transformation as a compiler
process. A tuple structure can be analysed by an attribute grammar. So, the termi-
nals are the linguistic categories e.g. N, DET, VP and the various words. The words
match with variables. The grammar rules including the nonterminals are constructed
in order to analyse the tuple structure. The features of the attribute grammar can
match with linguistic features e.g. genus or semantic roles. The heuristics are mapped
onto the grammar rules and integrated into semantic rules of the attribute grammar.
The semantic rules build the interpretation of the natural language input.
The following example illustrates the transformation. First the general tuple struc-
ture of a syntax tree is shown. For readability reasons we do not illustrate the syntax
tree with features, e.g. genus (c.f. figure 3). Features are lists of entries. They
are analysed in the same way as words. Secondly the grammar rules are described.
The upper letters refer to nonterminals. The terminals are in brackets. The ‘$’ is
used for marking variables. These variables match with words in the form of lexicon-
entries. Semantic rules are included into the grammar. ‘assert(X)’ asserts a fact X
to the interpretation. ‘complete-model’ completes the interpretation by establishing
a relation between the entity and the relationship. At last the interpretation of the
sentence is given.

Example: tuple structure:
\[
\text{syntaxtree}(s \langle np\,(\text{det}(\text{the}),\text{noun}(\text{user})), \notag \\
v\,(\text{borrow}), \notag \\
 np\,(\text{det}(\text{a}),\text{noun}(\text{book}))), \notag \\
pp\,(\text{prep}(\text{with})), \notag \\
 np\,(\text{det}(\text{a}),\text{noun}(\text{borrowing-slip})))\).
\]

grammar rules:
\[
\text{START} \rightarrow \text{‘syntaxtree’}, \text{‘(’, SENTENCE, ’)’}, \{ \text{complete-model} \}.
\]
The result of the first step is the interpretation of the natural language sentence. The interpretation shows a transformation of the natural language sentence onto the elements of the EER-model. It describes a one-sentence skeleton design.

4.2.2 Integrating a natural language interpretation into an existing model description

The integration of a natural language interpretation into an existing model description is the second step in the process of getting a skeleton design. There are different cases for integrating the new interpretation into the model description. E.g. a model description can have an entity which also is part of the new interpretation. The new interpretation and the model description has to connect using this entity. All cases of connection has to be checked if they are consistent. Questions will be initiated if there are inconsistencies.

4.2.3 Pragmatical completeness

The pragmatical completeness describes if there any logical possibilities to complete or to extend the given design. A model description is pragmatical complete if there are no such possibilities. Cases of incompleteness are e.g. an relationship does not have two entities or an entity does not have attributes. If a model description is incomplete pragmatical questions will be initiated. The search for incompleteness, in accordance with the design strategy (inside-out), first looks for important elements (missing entity/relationship) and then for special elements. The order of the elements...
in the table corresponds to the order search runs. The whole design is tested, i.e. searched.

The result of the structural transformation is a model description which contains a skeleton design. Within the transformation process the designer will be asked to give detailed information about the application by answering the questions.

4.3 Acquiring semantic information

In a natural language description the designer uses semantic nuances intuitively. Certain parts of the technical language are characterised by pragmatic properties which can be found in all applications. These pragmatic properties are discovered and confirmed by statistic observations. So e.g. the word 'several' implies a set greater than two. A designer using the word 'several' rarely wants it to refer to a set of two elements.

The acquisition of database semantics, e.g. the acquisition of keys or cardinalities from natural language sentences is part of the process of the pragmatic interpretation of the natural language input. Special functions within the attribute grammar are used for the transformation of the natural language semantics into the database semantics.

The acquisition of the semantic information needed for the design of a database is based on a set of heuristic assumptions which are linguistically motivated. These assumptions some of which are illustrated in the following sections are integrated into the knowledge base. The semantic results are assumptions which have to be evaluated in other RADD components (cf. [AlbBDT95]).

Key candidates

An assumption for the acquisition of key candidates is that e.g. attributes which have the character of numbers are defined as key candidates.

The assumption goes along with the fact that the named attribute reflects a number. The German nouns 'Hausnummer' (house number), 'Kartennummer' (card number), 'Datum' (date), 'Zeit’ (time) have the character of numbers. The same applies to nouns which have these nouns as substrings. Nouns which have a character of numerals only within a certain domain can be explained explicitly in the knowledge base (e.g. 'ISBN').

Example: 'Eine Bestandseinheit ist gekennzeichnet durch einen Typ, eine ISBN und Stichworte.'

(A bibliographical unit is characterised by a type, the ISBN and key words.)

Key: \text{keycand(bibliographical unit,type)}

Synonym and frequency dictionaries are used for the identification of words which reflect key assumptions.
Cardinalities

We have especially studied the role of special determiners, e.g. 'ein' (a), 'der' (the), 'jeder' (every), or 'mehrere' (several) are used for the acquisition of cardinalities in natural language utterances (cf. [Cyr95]). A designer using these determiners consciously or subconsciously defines certain cardinalities of an application and has his personal interpretation of the determiners. We try to extract the most plausible interpretations. The order of interpretations is described by the numbering contained in the facts. The following examples will illustrate the view at two determiners. The German word 'ein' (a) has the following meanings:

- mindestens ein (at least one) - 1:n - or
- genau ein (exactly one) - 1:1-.

Any other variants of the interpretation of 'ein'(a) are not relevant. If a designer uses the word 'ein'(a) explicitly we assume that it is most likely that he wants to describe a 1:1 cardinality.

Example: 'Ein Benutzer hat einen Benutzerausweis.'
(A user has a user card.)
Kardinalities: cardcand(1,has,user,1,1)
cardcand(2,has,user,1,n)
cardcand(1,has,user card,1,1)
cardcand(2,has,user card,1,n)

E.g. the zero article (non-occurrence of an article) mainly appears in connection with plural words. These words suggest the repeated occurrence of objects or executers. We assume that the designer when using zero articles does not want to describe exact and concrete objects or executers but prefers a 1:n cardinality.

Example: 'Benutzer entleihen Bücher'
(User borrows books.)
Kardinalities: cardcand(1,borrow,user,1,n)
cardcand(2,borrow,user,1,1)
cardcand(1,borrow,book,1,1)
cardcand(2,borrow,book,1,1)

Determiners have to be analysed for potential cardinalities. Then they are labelled before integration into the knowledge base. In many languages including German determiners are a manageable number of words. Labels on the cardinality potential describe internal characteristics of determiners and are, therefore, domain independent.

Inclusion and Exclusion Dependencies

Inclusion or exclusion dependencies are assumed in natural language sentences when entities are enumerated for which a hypernym exists. Enumeration exists e.g. when nouns are connected by connectors such as 'und' (and), 'oder' (or), 'sowohl als auch' (as well as) or by a comma.
4.4 Using the knowledge base for information on behaviour

In most cases a database will be used for complex processes. In order to be able to maintain the database we have to define transactions. (For the reasons of using transactions see [Tha94:114].) The behaviour of the database can help to make the system more efficient and faster and thus to save time and money. Behaviour can best be gained from the knowledge base. One form of presenting the domain is by classification of the processes involved as a conceptual graph. The knowledge base will be used for gathering relevant processes of the application. Each application can be classified, lending processes are e.g. the library processes or the 'rent a car' processes.

The lending process as a complex process can be further classified into a number of pre and post processes such as:

obtaining-registration-lending-returning.

If a user input contains one of these processes a possible classification will be defined. The pre and post processes can be further subdivided into processes which are summarized in the above classification. Lending thus requires the processes of obtaining a user card, updating the user card if need be, checking whether the book is held and available, filling in a borrowing-slip and signing it.

4.5 Representing the results of the transformation

The results of the transformation processes are transferred into a DataDictionary which we have developed in The RADD database design system [?]. This DataDictionary is available for all tools of the system, e.g. the designer can get a visualized form of his natural language description.

5 Conclusions

We have presented a Dialogue Tool consisting of a syntax analyser, a semantic role definer and a pragmatics interpreter. The Dialogue Tool gathers information on structure and semantics as well as information on behaviour of the prospective database. By means of transformation rules this information is mapped onto an EER model. The advantage of the Dialogue Tool is that the designer can describe the requirements of the database system in a natural language (in our case German) and thus
can specify the knowledge of a domain in a natural way. This knowledge is then employed for gathering database constructs such as entities, attributes, cardinalities, constraints, etc.

An open problem is the 'integrity' of the designer description of an application. We assume that a designer does not contradict himself. Future work will concentrate on take down a design session and give the possibility to change the opinion.

The efficiency of the database greatly depends on the exact interpretation and transformation of the natural language input analysis. The accuracy, on the other hand, depends on the size and complexity of the grammar used and the scope of the lexicon. Work in future has to concentrate on extending the grammar to comprise all types of sentences and other hitherto excluded parts of grammar and on ways of steadily increasing the lexicon.

References


Designing EER-Skeleton Schemes based on Natural Language

Bernhard Thalheim, Edith Buchholz, Haiko Cyriaks,
Antje Düsterhöft, Holger Mehlan

Technical University of Cottbus, Department of Computer Science
University of Rostock, Department of Computer Science

Abstract
The purpose of the paper is to illustrate how natural language can be used for obtaining an EER-skeleton scheme as a first step in designing correct databases. The skeleton scheme includes structural and semantic information as well as behaviour. We show how the knowledge of the designer can be gathered and which concepts of the natural language analysis are used for extracting information. An attribute grammar is applied for the transformation of designer descriptions onto EER-model structures.

1 Introduction
The performance of a database (especially efficiency and consistency) heavily depends on design decisions.
A database designer has to use a high level of abstraction for mapping his real-world application onto an entity relationship model. He/she has to learn the model and the constraints to use it. The bottleneck of the design process is the high degree of abstraction and the abstract language that has traditionally been used for this purpose.

1.1 Why and How use natural language?
Natural language allows the user to present his/her knowledge in an informal way. Describing the application in natural language the designer gives intuitively structural, semantic and behavioural constructs. The aim of a Natural Language Design Interface is to capture these complex information. The main topic is that a general strategy is needed which interpretes the natural language utterances in order to build the structure of database design schemes. During the process of interpretation the acquisition of semantical and behavioural constructs has to be initialized. But in reasons of ambiguity, fuzziness and redundancy of natural language ways have to be found to validate this first design.

We have implemented a knowledge-based Dialogue Design Tool for getting a skeleton design within the Database Design System RADD ([AlbABDT95]). The Dialogue Tool interacts with other components of the RADD-system to form a complex tool. The designer describes the structure of an application in German as a first step in the design process.
The results of a natural language design dialogue will be a structural skeleton, a candidate set of semantic constraints and a set of prospective processes. The designer can validate the results of the natural language analysis by means of other tools (e.g. a graphical editor, a semantics checker). The Data Dictionary which contains the actually design is usable in all tools.
The Dialogue Tool is currently implemented in PROLOG using SUN workstations.
The natural language approach does not fit all users. But the RADD-system allows the user to decide which design tool he/she likes best.

The natural language Dialogue Tool is especially designed for the following user groups:

- users which have no experience with database design and are not familiar with formal constructions
- users which have to confirm the correctness of modelling the application
- users which will work with specific parts of the database; they have view-oriented requests

Tests with these user groups show the usefulness of the tool.

On the other hand the user can work with different tools in order to become sensitive to modelling of the application. The user can e.g. formulate the design in natural language first and then applying the graphical representation in order to visualize the natural language design.

1.2 Restrictions concerning the natural language

It has become evident that a natural language interface as a 'general-purpose-system' for complex applications is not feasible. For this reason we decided to use a specific subject area of the German language with typical sentence structures and words.

Secondly we decided to select application areas for analysing processes. Each application area (e.g. lending processes) has several applications (e.g. library system, rent-a-car-system). The linguistic analysis is based on the application areas in which words are subject related and have no longer a wide scope of meaning.

1.3 Overview of this paper

The Introduction shows which methodologies and tools are existing that deal with natural language supporting database design process and database queries. Starting from this platform our approach and its aims are outlined.

In Section 2 we illustrate by example the potential of natural language which can be exploited for database design.

Section 3 shows the prototype implementation of the Dialogue Tool. The Dialogue together with the knowledge base will be used for drawing to the designer's attention special facts resulting from the syntactic, the semantic and the pragmatic analyses. The system makes suggestions for completing the design applying the knowledge base.

Section 4 deals with the process of transforming knowledge gained from natural language sentences into EER diagrams. Results of syntactic and semantic analyses of a natural language sentence and common heuristic rules, together with the knowledge base, are used for designing the skeleton and extracting information on the semantics as well as information on behaviour of the prospective database.

Section 5 presents conclusions and considers aspects of future work.

1.4 Background

Knowledge base. [StoG88], [GolS91] give an expert systems approach to creating user views. Choosing semi-natural language the so called VCS (view creating system) engages the user in a dialogue about the information requirements using a knowledge base. [BouG83], [FloPR85] show expert systems for database design. The systems interact with the user to ask for missing information.
[Tau90] presents expert systems for conceptual and logical database design which were developed as integral parts of a next generation CASE tool. In [MetML93] natural language techniques of semantic cases and conceptual graphs are used by several functions of schema manipulations.

**Natural language in database design.** [Che83] shows the close relationship between English sentence structure and ER diagrams. Within the system INCOD [AtzBLV83] the user can describe the application using a design language. The structure of the prospective database is developed step by step. The paper illustrates how the insertion of new EER models can be controlled to be consistent. [ColGS83] present a dialogue system for the analysis of natural language to extract knowledge from requirements description of data, transactions and events. Natural language as input is used in the project ANNAPURA [Eic84], [EicL85] for requirement specification. The system also uses questions to clarify the designer input. [RolP92] presents a Requirement Engineering support environment that generates the conceptual specification from a description of the problem area provided through natural language statements. [TjoB93] have described the development of database design and database queries using natural language (German). The main idea of the paper is that syntactic structures of the natural language can be semi-automatically transformed into EER structures using rules and heuristics. A so-called DMG (Data Model Generator) was introduced to process sentences in order to get a requirement specification.

[Ort93] and [Ort94] studies the technical language (German) which is used in working situations in order to develop a standardised enterprise language for software development. Within these studies the relationship between natural language and elements of an object-oriented model will be analysed.

**Semantics.** In [TseCY92] a methodology is presented which maps natural language of example constructs into relational algebras through ER representations. The natural language conjunctives ‘and’ and ‘or’ are mathematically analysed. The close relationship between attribute grammars and the semantics of the natural language is shown in [PitC90]. Recently computational linguistics has produced a series of outstanding contributions to automatic processing of natural language (e.g. in the field of semantics [Pin93]).

**Problem solving methods.** Structured applications can be developed based on methods of skeleton construction [Pol]. This problem solving method is based on abstraction, modularisation and local heuristic selection criterias. These methods have been used for expert systems [Fried79, FrieI85, BarOC]. The knowledge on the application can be hierarchically represented. Expansion is driven by heuristic selection functions. Phase models are based on one skeleton. Refinement is the main construction method.

**Our approach.** We maintain that natural language descriptions of the requirements of a prospective database are so complex that by skilful interaction with the user, a tool can extract not only the structure but the semantics of the database as well as behaviour. We have chosen a Dialogue Tool because we are convinced that for most applications descriptions in formal complex texts are not available. Furthermore, using texts and asking the designer questions only when the system does not understand them will not stimulate him to extract knowledge of the domain.

### 2 The potential of natural language for database design

Let us consider a person who has used a library at least once in his/her life. To this person the sentence

'Der Benutzer entleiht ein Buch mit einem Leihschein.'

(The user borrows a book with a borrowing-slip.)
automatically conveys a large amount of information. Because of language as well as world knowledge the person knows that

- user and book are two partners in an act
- a certain relationship exists between these two, expressed by the verb
- the relationship is the change of possession
- the change of 'ownership' is only temporarily (in contrast to buying or presenting)
- the user is not a specific one, in fact it can be any Person that fulfils the conditions going along with this business
- a book does not mean one book, it can be two or three and more books
- a book does not necessarily have to be a book, it can be a picturebook, a journal, a magazine, an atlas
- in 'book with a borrowing-slip' with does not express possession or part of like in 'a book with pictures', it expresses instrumentality (by means of)
- the 'borrowing-slip' is not just a slip of paper but has the function of a contract
- borrowing means that the book will have to be returned
- a book is borrowed at some place (library)
- this sentence does not include details about user or book, it is a general lending process.

This complex information which a person associates when reading the sentence will be elicited by the Dialogue Tool in various steps. These steps have to comprise

1. a linguistic analysis of the natural language for syntax and semantics
2. a set of transformation rules for extracting the database structure
3. rules and heuristics for handling world knowledge
4. classification rules for modelling the behaviour

In the following chapters we will illustrate the basic transformations to achieve this aim. As the prototype tool is based on syntactic, semantic and world knowledge we had to choose a domain as the background of the prospective database and as the object of the domain knowledge base. We decided to take the field 'library - its tasks and processes'. As a method of obtaining the linguistic corpus we decided to carry out a number of interviews with librarians as well as library users (cf. [BucD95]).

For readability reasons we show the following examples in German (the implementation language) and in English. The results of the analysis are given in English only.

3 A Dialogue Tool for acquiring knowledge

3.1 Dialogue model

For the acquisition of designer knowledge we chose a moderated dialogue. A moderated dialogue can be seen as a question-answer system. The Dialogue Tool reacts appropriately to every input sentence. It asks for additional input if it finds gaps in the presentation of database design information. These
Figure 1: Two-stage Dialogue interpretation
questions are frames which will be updated in the dialogue process. The designer can form the answers in natural language sentences. Figure 3.1 shows the structure of the analysis of a natural language input.

Within the Dialogue the results of the syntactic, semantic and pragmatic analyses will be used for controlling the Dialogue. This means, if an incomplete designer input is received a question will be initiated. Inputs are incomplete if either the results of the linguistic analysis are not complete or the design model generated so far is incomplete. Figure 2 illustrates the general flow chart of the dialogue.

We distinguish three types of questions:

1. CQ: content question (e.g. 'Are there any more details about the application?')
2. LQ: linguistic clarification questions (e.g. 'How is the act -borrow- done?')
3. PQ: pragmatic clarification questions (e.g. 'How is -book- characterized?).

We have chosen the inside-out approach [Tha93] for getting a design. That means we concentrate firstly on the main elements of the design and then we try to find details of the elements. The additional pragmatic questions for clarifications are put to the user in the same order.

In the following sections we will show in which concrete case the questions will be initiated.

### 3.2 Syntactic analysis

The designer input into the dialogue tool is first submitted to a syntax analyser.

**Grammar and lexicon.** We have developed an special phrase structure grammar which uses the ID/LP format (Immediate Dependence/ Linear Precedence [Gaz85]) and includes meta rules. The grammar formalism describes a part of the German language based on analyses of user-input (c.f. also [Meh95]). The grammar analyses main and subsidiary clauses, relative clauses, prepositional clauses as well as all basic verb phrases (except extended infinitive clauses phrases).

The underlying lexicon contains lexeme and morphological rules. As a method of obtaining the linguistic corpus we carried out a number of interviews with librarians as well as library users. These talks in most cases had the nature of dialogues with lengthy monologue parts in them. The talks were tape-recorded and afterwards transcribed. The corpus obtained consists of more than 12,000 lexical units. The information gained from it was sustained by referring to the two-volume 'Lexikon des Bibliothekswesens'.

**Parser.** We have implemented a special parser which uses the grammar as well as the lexicon and transforms natural language utterances into syntax trees. A set of syntax trees is produced as a result of the syntactic analysis. All categories can have syntactic features e.g. genus or case. Analysing the sentence 'Die Leihzeit betraegt 4 Wochen' (The lending period is 4 weeks.) it yields the following syntax trees (Figure 3).

### 3.3 Semantic analysis

Interpreting the semantics of the designer input we use the two-stage linguistic model which inserts a semantic level between the syntax level and the conceptual level (HERM data model). Within the semantic check the meaning of the sentences will be analysed and presented independent of the domain.

**Principle of compositionality.** We assume the principle of computationally. [GaM89, 280] describe Frege's principle: 'The meaning of the whole sentence is a function of the meaning of the parts.' In other words, the meaning of a sentence is composed of the meaning of its parts. The meaning of the parts is then described as the meaning of the subphrases and so on. In the end we have to
Figure 2: The general flow chart

question of contents

designer answer

answer = end? yes no

analyses (syntax, sem. roles)

ling. incomplete? yes no

storing the answer

linguistically motivated question

pragm. interpretation of the answer

pragm. incomplete? yes no

pragmatically motivated question
Die Leihzeit beträgt 4 Wochen.

Figure 3: Syntax tree of the sentence 'Die Leihzeit betraegt 4 Wochen'
examine the meaning of the words. For this reason the basis of the semantics of a sentence is the semantics of words and phrases defined in the lexicon.

**Meaning of a sentence.** To identify the meaning of sentences we have used the model of semantic roles. The units in a sentence or an utterance are seen to fulfill certain roles. Our role concept is mainly based on Jackendoff's hypothesis [Jac83] and consists of the following roles which refer to the objects partaking in the action: Cause, Theme, Result/Goal, Source, Locative, Temporal, Mode, Voice/Aspect.

The roles of a sentence are used to clarify linguistically completeness and to support the extraction of the design.

**Verbs.** Verbs form a central part in defining the meaning of sentences and the relationships between parts of sentences. Basically they describe actions, processes and states. We have tried to find a classification of verb semantics [BucD95] that can be applied to all verbs in the German language (and probably other languages as well). Our aim was to keep the number of classes small and fairly general but large enough to identify their function in a sentence correctly. This classification is, at this stage, independent of the domain to be analysed (cf. Figure 4).

The following example shows the semantic roles of the sentence 'Der Benutzer entleiht ein Buch mit einem Leihchein.' (The user borrows a book with a borrowing-slip.)

**Example:** The user borrows a book with a borrowing-slip.

**Semantics:**
- verb type: verb of movement
- Cause (subject): the user
- Theme (object): a book
- Locative: ? (an additional question will be initiated -LQ-)
- Temporal: ? (an additional question will be initiated -LQ-)
- Mode: with a borrowing-slip

**Nouns and determiners.** Nouns describe objects of the real world. For database design purposes we are not interested in the nature of these objects. But we are interested in the relations which the objects have and in quantifying these relations. For analysing the determiners of the nouns we need a noun classification. We adapted the classification of [Bis91] as a first step and distinguish three types of objects: abstract, local and temporal objects. Each type can also express individuality, continuity and uniqueness (see [Cyr95]). For analysing the determiners we use the model theoretic description ([BarC81]).

Nouns and determiners are used to define cardinalities as well as inclusions and exclusions dependencies shown in Section 4.

### 3.4 Pragmatic interpretation

The pragmatic interpretation is part of the Dialogue Tool. The aim of the pragmatic interpretation is the mapping of the natural language input onto EER model structures using the results of the syntactic and semantic analyses. A basic feature of the pragmatics is the domain model which defines a frame for every specific application and supports the acquisition of semantics and behaviour of the prospective database. Using the results of the linguistic analysis a common domain model is gradually developed. During the design process this model can be updated. **Section 4** will show how common rules are used for making general assumptions about how information gained from natural sentences is related to entities, relationships, sets, keys and other EER model structures.

9
Figure 4: Verb classification

verbs

static
dynamic
modal

verbs of position (locative)
copulative verbs stative verbs

modal verbs epistemic verbs catalysts

transport verbs of production event verbs

change of ownership verbs of movement perception verbs utterance verbs (communication)
4 Transforming knowledge gained from natural language sentences into EER diagrams

4.1 Assumptions concerning the EER model

Due to readability reasons a section of the EER model is used in this paper to illustrate the process of transformation. (The other constructs of the underlying EER model can be modelled in the same fashion.)

The following structural constructs of EER models are the basis of our natural language transformation.

- **entity(ENAME)**: describes an entity with the name ENAME
- **relationships(RName,ENAME1,[ENAME2])**: describes a relationship RName between entity ENAME1 and the list of entities (ENAME2 - describes an set of according entities); the is-a-classification will be described as relationships(is-a,ENAME1,[ENAME2]) where a ENAME2 is a ENAME1
- **attre(ENAME,AName)**: the entity ENAME has an attribute AName
- **attrr(RName,AName)**: the relationship RName has an attribute AName

Semantic constructs are key candidates, cardinalities as well as inclusion and exclusion dependencies.

- **keycand(ENAME/RName,AName)**: the attribute AName is a key candidate of the entity ENAME or the relationship RName
- **cardcand(NR,RName,ENAME,MinCard,MaxCard)**: the relationship RName has a cardinalities MinCard:MaxCard corresponding to the entity ENAME
- **inlcand(ENAME1,ENAME2)**: describes an inclusions dependency of two entities (ENAME1 and ENAME2) where a ENAME1 includes a ENAME2
- **exlcand([ENAME])**: describes a list of excluded entities ENAME
- **neg-exlcand([ENAME1,...,ENAMEN])**: describes a list of entities ENAME1, ...,ENAMEN which have not an empty average

We assume that names of entities and relationships are unambiguous. We define the model description as the representation of the presented design of the prospective data base. The model description is an ordered set of facts.

4.2 Obtaining a skeleton design of the prospective database

The aim of the pragmatic interpretation of a natural language input is firstly obtaining a skeleton structure and secondly getting information on semantics of the prospective database. In this section we will illustrate how we can extract a skeleton design by using world knowledge and heuristics. This general interpretation is domain independent.

The structural pragmatic interpretation consists of three steps. First the syntactic and semantic results of a natural language input are analysed and interpreted. Then this interpretation will be inserted into the actual model description, after which the new model description will be checked in order to find pragmatic incompleteness. If the model description is not complete additional questions (PQ) will be initiated. The following sections illustrate these steps.
4.2.1 Interpreting a natural language sentence

**Heuristics.** The transformation of the structure of natural language sentences into HERM model elements is a process which is based on heuristic assumptions. ([TjoB93] illustrate a large number of such heuristics in an informal way.) These heuristics illustrate the close relationship between words/phrases and model elements. We have defined and formalised such heuristics using contextfree and contextsensitive rules. Figure 5 shows some of these rules. (The '$$' indicates an element which is needed. An special question will be initiated in order to obtain this element.)

**Attribute Grammar.** The result of the syntactic analysis is a set of syntax trees of the natural language input. We see a syntax tree as a tuple structure and the set of possible syntax trees as a language. Now we handle the transformation as a compiler process. A tuple structure can be analysed by an attribute grammar. So, the terminals are the linguistic categories e.g. N, DET, VP and the various words. The words match with variables. The grammar rules including the nonterminals are constructed in order to analyse the tuple structure. The features of the attribute grammar can match with linguistic features e.g. genus or semantic roles. The heuristics are mapped onto the grammar rules and integrated into semantic rules of the attribute grammar. The semantic rules build the interpretation of the natural language input.

The advantage of using an attribute grammar for transformation is the given modularity. The attribute grammar can be developed without knowing the concrete parser and the concrete grammar. Only the syntactic structure of the sentence has to be known. Otherwise the integration of new heuristics or rules is simple because the heuristics are modular integrated only in semantic rules of the attribute grammar. Another fact is that algorithms known in the field of compiler constructions can be applied.

The disadvantage is that we need two times to analyse a natural language sentence. The first time is needed to identify the syntactical structure and in the second time we define the pragmatical interpretation.

The following example illustrates the transformation. First the general tuple structure of a syntax tree is shown. For readability reasons we do not illustrate the syntax tree with features, e.g. genus (c.f. figure 3). Features are lists of entries. The are analysed in the same way as words. Secondly the grammar rules are described. The upper letters refer to nonterminals. The terminals are in brackets. The '$$' is used for marking variables. These variables match with words in the form of lexicon-entries. Semantic rules are included into the grammar. 'assert(X)' asserts a fact X to the interpretation. 'complete-model' completes the interpretation by establishing a relation between the entity and the relationship. At last the interpretation of the sentence is given.

Example: **tuple structure:**

```
syntaxtree(s( np(det(the),noun(user)),
vp(   v(borrow),
    np(det(a),noun(book))),
pp(    prep(with),
    np(det(a),noun(borrowing-slip))))).
```

**grammar rules:**

```
START -> 'syntaxtree', '(' , SENTENCE , ',' , ')' , { complete-model } .
SENTENCE -> '(' , NP-PHRASE , ',' , VP-PHRASE , PP-PHRASE , ',' , ')'.
NP-PHRASE -> '(' , DET , ',' , NOUN , ')'.
DET -> 'det' , '(' , $D , ')' .
NOUN -> 'noun' , '(' , $N , ',') { assert(entity($N)) } .
    / heuristic: nouns transferred to entities */
VP-PHRASE -> '(' , VERB , ',' , NP-PHRASE , ')'.
VERB -> 'v' , '(' , $V , ',') { assert(relship($V,...)) } .
    / heuristic: verbs transferred to relationships */
PP-PHRASE -> 'pp' , '(' , PREP , NP-PHRASE , ')'.
PREP -> 'prep' , '(' , $P , ',') .
complete-model :- relship(X,...), !,
    entity(E).```

12
<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Formalised Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>All nouns are transferred to entities.</td>
<td>noun(X) → entity(X)</td>
</tr>
<tr>
<td>Sentences with the verb 'have' are attribute descriptions.</td>
<td>noun(X), subject(X), noun(Y), object(Y), verb(have) → entity(X), attre(Y,X)</td>
</tr>
<tr>
<td>Sentences with the verb 'be' and an object are is-a-classifications.</td>
<td>noun(X), subject(X), noun(Y), object(Y), verb(be) → entity(X), entity(Y), relship(is_a, X,[Y])</td>
</tr>
<tr>
<td>Personal names indicate nouns.</td>
<td>pername(X) → entity($Y).</td>
</tr>
<tr>
<td>Adjectives indicate attributes.</td>
<td>adj(X) → (entity($Y), attre($Z,$Y)) ; (relship($Y), attrr($Z,$Y))</td>
</tr>
<tr>
<td>Numbers used in a sentence indicate nouns.</td>
<td>number(X) → entity($Y)</td>
</tr>
<tr>
<td>Relative pronouns indicate nouns.</td>
<td>relpro(X) → entity($Y)</td>
</tr>
<tr>
<td>Sentences with two nominal phrases (NP1,NP2) where the NP1 is in genitive</td>
<td>np_phrase(X), np_phrase(Y) → entity(Y), attre(X,Y).</td>
</tr>
<tr>
<td>case to NP2 indicate an attribute relation between NP1 and NP2.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Heuristics expressing the relationship between words/phrases and model elements
interpretation of the sentence:
entity(user).
entity(book).
entity(borrowing-slip).
relship(borrow,user,[book,borrowing-slip]).

The result of the first step is the interpretation of the natural language sentence. The interpretation shows a transformation of the natural language sentence onto the elements of the HERM-model. It describes a one-sentence skeleton design.

4.2.2 Integrating a natural language interpretation into an existing model description

The integration of a natural language interpretation into an existing model description is the second step in the process of getting a skeleton design.

For the integrating the new interpretation into the model description we distinguish the following cases:

1. The model description is empty. The new model description will be the new interpretation.

2. The model description and the interpretation are not empty. There exists names of model elements which have both designs.
   - An entity or a relationship are described in detail with attributes. The new model description contains the entity/relationship with these details.
   - There exists an entity in the model description and the interpretation. The entity has different relationships. The new model description contains the entity and two relationships connected with this entity.
   - An attribute (of an entity -E1-) exists in the model description. Within the interpretation this attribute is an entity (E2) and is described in detail. The new model description contains this attribute as an entity (E2). A question (PQ) is used to find the relationship between the entity (E1) and the entity (E2).
   - The model description contains an entity (E1) and a connected relationship (R). The interpretation has a is-a relationship between the entity (E1) and another entity (E2) where E1 is a E2 (the abstraction). In the new model description the entity (E2) is connected with the relationship (R). The entity (E1) is connected with the entity (E2) via an is-a relationship. We assume that the abstraction (E2) also has the relation (R).
   - The model description has a relationship (R1) transformed from a verb. The interpretation contains an entity (E1) which is the according noun. The entity is described in detail by attributes (case 1) or by a relation (R2) to another entity -E1- (case 2).

     In case 1 the attributes are connected to the relationship (R1) in the new model description. In case 2 the relationship (R1) is connected with the relationship (R2) in the new model description.

3. The model description and the interpretation are not empty. There does not exist names of elements which are in both designs. Questions (PQ) will be initiated in order to find connections between the entities/relationships of the model description and the interpretation.

The following example illustrates an integration.
Example:

Sentence: Books are bibliographical units.

Sentence interpretation:
- entity(book).
- entity(bibliographical unit).
- relship(is-a,bibliographical unit,[book]).

Model description: (From the sentence: The user borrows a book with a borrowing-slip.)
- entity(user).
- entity(book).
- entity(borrowing-slip).
- relship(borrow,book,[user,borrowing-slip]).

Result of integration:
- entity(user).
- entity(book).
- entity(borrowing-slip).
- entity(bibliographical unit).
- relship(borrow,bibliographical unit,[user,borrowing-slip]).
- relship(is-a,bibliographical unit,[book]).

The result, the new model description, has to be checked whether it is pragmatically complete or not. The following section illustrates these checks.

4.2.3 Pragmatical completeness

The pragmatical completeness describes if there any logical possibilities to complete or to extend the given design. A model description is pragmatical complete if there are no such possibilities. If a model description is incomplete pragmatical questions will be initiated. The following Figure 6 shows our assumptions in order to find incompletenesses 1.

The search for incompleteness, in accordance with the design strategy (inside-out), first looks for important elements (missing entity/relationship) and then for special elements. The order of the elements in the table corresponds to the order search runs. The whole design is tested, i.e. searched.

The result of the structural transformation is a model description which contains a skeleton design. Within transformation process the designer will be asked to give detailed information about the application by answering the questions.

4.3 Acquiring semantic information

In a natural language description the designer uses semantic nuances intuitively. Certain parts of the technical language are characterised by pragmatic properties which can be found in all applications. These pragmatic properties are discovered and confirmed by statistic observations. So e.g. the word ‘several’ implies a set greater than two. A designer using the word ‘several’ rarely wants it to refer to a set of two elements.

The acquisition of database semantics, e.g. the acquisition of keys or cardinalities from natural language sentences is part of the process of the pragmatic interpretation of the natural language input. Special functions within the attribute grammar are used for the transformation of the natural language semantics into the database semantics (see Figure 7).

The acquisition of the semantic information needed for the design of a database is based on a set of heuristic assumptions which are linguistically motivated. These assumptions some of which are illustrated in the following sections are integrated into the knowledge base. The semantic results are assumptions which have to be evaluated in other RADD components (cf. [AlbBDT95]).

1The natural language input used within the dialogue can be complete sentences as well as incomplete utterances like noun phrases or verbal phrases
<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Pragmatical Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>An entity does not have a relationship.</td>
<td>Does exist a relation between the entity E and an entity X? (search all entities X)</td>
</tr>
<tr>
<td>R</td>
<td>An relationship does not have entities.</td>
<td>Does R have a relation to the entity X? (search all entities X)</td>
</tr>
<tr>
<td>E</td>
<td>A relationship does not have 2 entities.</td>
<td>The entity E has the relation R described by a verb. Does exist another connection to an entity X?</td>
</tr>
<tr>
<td>E</td>
<td>An entity does not have attributes.</td>
<td>Please give more details about the entity E!</td>
</tr>
<tr>
<td>E</td>
<td>An entity has attributes, but no key attribute.</td>
<td>You gave the following attributes. ... Which is the most important attribute?</td>
</tr>
<tr>
<td>E1</td>
<td>An entity E1 is the abstraction of the entity E2. There does not exist another entity which has also these abstraction.</td>
<td>The entity E1 is a abstraction of the entity E2. Does exists an entity E3 (E3&lt;&gt;E2) which has also this abstraction?</td>
</tr>
</tbody>
</table>

Figure 6: Completeness of the design.
<table>
<thead>
<tr>
<th>quantifier</th>
<th>Q</th>
<th>¬ Q</th>
<th>Q ¬</th>
<th>Q ~</th>
</tr>
</thead>
<tbody>
<tr>
<td>´alle´ (all)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>´nicht alle´ (not all)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>´kein´ (no)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e.g. ´einige´ (some)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>inclusion</td>
<td>negated inclusion</td>
<td>exclusion</td>
<td>negated exclusion</td>
</tr>
</tbody>
</table>
4.3.1 Key candidates

Assumptions for the acquisition of key candidates are:

1. the first named attribute of an entity is defined as a key candidate
2. attributes which have the character of numbers are defined as key candidates
3. identifying nouns which are marked in the lexicon are key candidates

The first assumption is based on the observation that a designer names the most important attribute first.

Example: 'Eine Bestandseinheit hat eine Registriernummer und einen Standort.'
(A bibliographical unit has a registration number and a location.)
Key: keycand(bibliographical unit, registration number)

Assumption 2 goes along with the fact that the named attribute reflects a number. The German nouns 'Hausnummer' (house number), 'Kartennummer' (card number), 'Datum' (date), 'Zeit' (time) have the character of numbers. The same applies to nouns which have these nouns as substrings. Nouns which have a character of numerals only within a certain domain can be explained explicitly in the knowledge base (e.g. 'ISBN').

Example: 'Eine Bestandseinheit ist gekennzeichnet durch einen Typ, eine ISBN und Stichworte.'
(A bibliographical unit is characterised by a type, the ISBN and key words.)
Key: keycand(bibliographical unit, type)
keycand(bibliographical unit, ISBN)

The third assumption is based on the character of certain nouns which makes them identifiers. This group comprises words such as 'Kennzeichnung' (label), 'Bezeichnung' (term), 'Benennung' (naming) or 'Identifikator' (identifier).

Example: 'Ein Buch hat eine Signatur und ein Format.'
(A book has a signature and a format.)
key: keycand(book, signature)

Synonym and frequency dictionaries are used for the identification of words which reflect key assumptions.

4.3.2 Cardinalities

We have especially studied the role of special determiners (e.g. 'ein' (a), 'der' (the), 'jeder' (every), or 'mehere' (several) are used for the acquisition of cardinalities in natural language utterances (cf. [Cyr95]). A designer using these determiners consciously or subconsciously defines certain cardinalities of an application and has his personal interpretation of the determiners. We try to extract the most plausible interpretations. The order of interpretations is described by the numbering contained in the facts. The following examples will illustrate the view at some determiners. The German word 'ein' (a) has the following meanings:

- mindestens ein (at least one) - 1:n - or
- genau ein (exactly one) - 1:1-

Any other variants of the interpretation of 'ein'(a) are not relevant. If a designer uses the word 'ein'(a) explicitly we assume that it is most likely that he wants to describe a 1:1 cardinality.

Example: 'Ein Benutzer hat einen Benutzerausweis.'
(A user has a user card.)
Kardinalities: cardcand(1, has, user, 1,1)
cardcand(2, has, user, 1,n)
cardcand(1, has, user card, 1,1)
cardcand(2, has, user card, 1,n)
The zero article (non-occurrence of an article) mainly appears in connection with plural words. These words suggest the repeated occurrence of objects or executers. We assume that the designer when using zero articles does not want to describe exact and concrete objects or executers but prefers a 1:n cardinality.

Example: 'Benutzer entleihen Bücher'
(User borrows books.)
Kardinalities: cardcand(1,borrow,user,1,n)
cardcand(2,borrow,user,1,1)
cardcand(1,borrow,book,1,n)
cardcand(2,borrow,book,1,1)

The German word 'mehrere' (several) reflects a 1:n relation. An interpretation of 'exactly one' is improbable.

Example: 'Ein Benutzer kann mehrere Bücher entleihen.'
(A user can borrow several books.)
Kardinalities: cardcand(1,borrow,user,1,1)
cardcand(2,borrow,user,1,n)
cardcand(1,borrow,book,1,n)

We assume that the determiners used together with enumerations have a wide scope. That means a determiner are interpreted for each enumerated noun.

Example: 'Ein Benutzer kann 5 Bücher und CDs entleihen.'
(A user can borrow 5 books and cd’s.)
Kardinalities: cardcand(1,borrow,user,1,n)
cardcand(2,borrow,user,1,n)
cardcand(1,borrow,book,1,5)
cardcand(1,borrow,cd,1,5)

Determiners have to be analysed for potential cardinalities. Then they are labelled before integration into the knowledge base. In many languages including German determiners are a manageable number of words. Labels on the cardinality potential describe internal characteristics of determiners and are, therefore, domain independent.

4.3.3 Inclusion and Exclusion Dependencies

Inclusion or exclusion dependencies are assumed in natural language sentences e.g. when entities are enumerated for which a hypernym exists. Enumeration exists when nouns are connected by connectors such as 'und' (and), 'oder' (or), 'sowohl als auch' (as well as) or by a comma.

Example: 'Studenten, Mitarbeiter und Dritte sind Benutzer.'
(Students, staff and third parties are borrowers.)
Exclusion dependency: exclcand([student,staff,third partie])
Inclusion dependency: inclcand(borrower,student)
 inclcand(borrower,staff)
 inclcand(borrower,third partie)

Inclusion and exclusion dependencies have direct pendants in the natural language quantifiers: 'alle' (all), the negation 'nicht alle' (not all) and 'kein' (no).

Example: 'Kein Student ist ein Mitarbeiter.'
(No student is a staff member.)
Exclusion dependency: exclcand([student,staff member])
Negated exclusion dependencies are assumed if the designer describes exceptions. So, we consider that e.g. the quantifier 'nicht alle' (not all) is equivalent to the quantifier 'fast alle' (nearly all) and means that the number of exceptions is small in comparison with the number of the elements of the whole set.

Example: 'Nicht alle Studenten sind Bibliotheksnutzer.'
(Not all students are borrowers.)
Negated exclusions dependencies: neg-excland([student,borrower])

Figure 8 shows the quantifiers and their interpretation in the design process.

4.4 Using the knowledge base for information on behaviour

In most cases a database will be used for complex processes. In order to be able to maintain the database we have to define transactions. (For the reasons of using transactions see [Tha94:114].) The behaviour of the database can help to make the system more efficient and faster and thus to save time and money. Behaviour can best be gained from the knowledge base. One form of presenting the domain is by classification of the processes involved as a conceptual graph. The knowledge base will be used for gathering relevant processes of the application. Each application can be classified, lending processes are e.g. the library processes or the 'rent a car' processes.

The lending process as a complex process can be further classified into a number of pre and post processes (cf. Figure 9). If a user input contains one of these processes a possible classification will be defined.

The pre and post processes in Fig. 6 can be further subdivided into processes which are summarized in the above classification. Lending thus requires the processes of obtaining a user card, updating the user card if need be, checking whether the book is held and available, filling in a borrowing-slip and signing it.

Example: The sentence 'The user borrows a book with borrowing-slip.' implies the following questions:

preprocesses:
1) Does the process 'obtaining' occur before 'lending'?
2) Does the process 'registration' occur before 'lending'?

mainprocesses:
3) Does the process 'document exists' occur before 'lending'?
4) Does the process 'document valid' occur before 'lending'?
...

postprocesses:
5) Does the process 'returning' occur after 'lending'?

The designer has to give the correct answers.

4.5 Representing the results of the transformation

The results of the transformation processes are transferred into a DataDictionary which we have developed in the RADD database design system. This DataDictionary is available for all tools of the system, e.g. the designer can get a visualized form of his natural language description.

5 Conclusions / Future work

We have presented a natural language approach for designing skeleton schemes. The syntactical structure of a natural language input will be transformed on to EER model structures using an attribute grammar and heuristics. The semantics of special words will be used to obtain semantic constraints like keys or cardinalities. A specific process model which depends on the application area
Figure 9: Pre, main and post processes of the act of borrowing/lending
gives the opportunity to capture prospective processes of the database. The knowledge of the designer will be gathered by means of a moderated dialogue.

An unsolved problem is the 'integrity' of the designer description of an application. We assume that a designer does not contradict himself. Future work will concentrate on taking down a design session and giving the designer the chance to change his/her mind. Another contemporary issue is to develop an interesting dialogue with the user. We have to decide which user is able to answer the specific additional questions and in which manner we have to ask the user.

The efficiency of the database greatly depends on the exact interpretation and transformation of the natural language input analysis. The accuracy, on the other hand, depends on the size and complexity of the grammar used and the scope of the lexicon. Work in future also has to concentrate on extending the grammar to comprise all types of sentences and other hitherto excluded parts of grammar and on ways of steadily increasing the lexicon.

References


23

[Fried79] Friedland, P.E.: Knowledge-based experiment design in molecular genetics. PhD, Stanford University, 1979


[Pol] Polya


[TjoB93] Tjoa, A.M., Berger, L.: Transformation of Requirements Specifications Expressed in Natural Lan-
guage into an EER Model. Proceedving of the 12thInternational Conference on ER- Approach, Airlington,
Texas USA,Dec. 15-17th, 1993

[TseCY92] Tseng, F.; Chen, A.; Yang, W.-P: On mapping natural language constructs into relational algebra
Abstract

In the Internet age sophisticated dynamic support for users becomes even more crucial. Users of web systems did not get a CS education. Neither they know query languages. They want to ask their requests directly to the engine. For instance, the development of city information systems such as www.cottbus.de pushed us towards an NL support for querying. Users wanted to ask dynamic queries not knowing the schema, the terminology of the database system supporting the information services. The same situation we met while developing edutainment sites [Cau00]. Meanwhile the Cottbusnet Internet site development team has participated in more than 30 projects aiming in developing large Internet sites. This experience has been summarized in [ScT00]. At the same time, the theoretical basis has been developed [Lew00, FVFT01, GST00, Sri01].

Sophisticated support for users of Internet services is a must nowadays. However, the development of such support is a really challenging task due to several obstacles:

1. Variety of accessed databases: Internet user do not want to stick to one database. Instead of that they want to access several databases. Thus, the different database systems need to be well-integrated. We observe the development of portals which currently allow to survey the
information content of several databases. An integrated access to different databases is still a very task.

**Psychological barriers:** Users have their difficulties to understand the systems.

- Users pay to systems limited attention and use limited memory resources. For this reason they do not want to be overwhelmed with information. They do not want to learn the database schemata which are used for the site support. They want to ask simple queries.
- User lack in their abilities to formally represent a query. They are limited in translating their request to the language of the system.
- Users semantics heavily differs. They use various semantics for logical quantifiers and connectives such as $\forall, \exists, \land, \lor$ and $\neg$. Most search engines try to become more flexible. We have used intensional logics in order to overcome this problem.

**Presentation barriers:** In the first years of site development main concentration has been oriented towards fancyness. Sites should have a great graphical interface. At the same time, users do not want to read long ascii texts representing the answer to their request. They want to briefly capture the sense of the answer and they do not have the time to dig out the information they wanted to see in hills of lines.

**Dynamic access:** Users want to access a service dynamically. They may ask a large variety of questions. The typical approach in order to satisfy this requirement is currently to generate a large variety of components [Rie01] which can be dynamically composed in order to answer complex questions.

Analyzing this situation we found that it might be useful to find a solution to the following problem:

(*) Is it possible to dynamically generate the support which enables an internet service to answer questions to the system which are ask in natural language?

If such support would be possible then internet sites may be used by a large variety of users. Of course, a universal solution is infeasible. However we might use information which is in our hands:

- The database content is known. Thus, we might extract main keywords from the database. Further, we can integrate this knowledge into more general ontology and use sophisticated lexicons.
- Natural language sentences might be complex, ambiguous and use ellipses. Therefore, we need sophisticated analysis support for NL utterances.
- The database structure is known. Thus we might use the structure and compare this structure with the syntax tree of the utterance.

The research and development presented in this paper has been concentrated on the second and the third task. Since solutions to the first problem have been developed (see for instance the NLDB conferences) we wanted to show that under certain circumstances the solution to these problems is feasible. In order to solve the second task we reused results [BDT96, Düs97] which have been obtained while developing a database design workbench [AAB95] which allows natural language specification of applications and generates based on this the corresponding database schema. The solution to the third task presented below is based on an extended entity-relationship model [Tha00]. The tools have been implemented [Kob00] and is used in internet applications.
2 State of the Art

Internet services are far away from flexible and dynamic support for users requests. A large number of research issues is still open need to be solved. Current internet sites are characterized by a number of severe restrictions:

Static query interfaces: Most query interfaces are based on menu technology. There are very few examples of sites which react flexible. This inflexibility comes from a number of research problems which are still open.

- Component technology is not yet matured in order to allow runtime adaptation of result presentation. But XML technology will enable this flexibility in next years.
- Site development is often based on mapping existing systems to internet systems. Instead of that, flexible story specification as proposed in [Sri01] enables in story space and interaction space specification.

No or very restricted NL access: Users use natural language in their daily live. They have learned to use the language very sophisticated. Utterances used for communication are often simple. Deep sentence structures are not very common. The complexity comes however from sophisticated usage of associations in the application. These associations might be represented by paths in the schema of the application. Communication partners usually first agree on a common communication language which enables in communication and which allows transformation of utterances to the language used for either side. Thus, we might use these two ideas:

- The associations are represented by sub-graphs or in the simple case by paths in the conceptual schema. The conceptual schema is better for this purpose since it is used to represent concepts of the application domain and it is not restricted by the implementation platform.
- Similar to human services where service personal is able to translate user requests to specific service specification we might develop such transformers based on a thorough understanding of the application.

Natural language support has to cope with ambiguities and ellipses.

Simple query interfaces: The query interface needs to be as simple as possible. A user should have a way to comprehend the content of the databases which can be used by the web system. And further to ask a question in his/her way.

- Database systems use their vocabulary. This specific language has been introduced under the pressure of the application area and reflects its specific knowledge. This vocabulary is not common to casual users. Thus we need a ‘simplifier’.
- Web systems often provide only access to one specific database. Web portals do not integrate different databases and provide only selection facilities for access to singleton separated applications.

Problematic IR solutions: Information retrieval systems are widely applied in the internet context. Their application has a larger number of restrictions.

- The selection of language and functions is not appropriate. Semantics of Boolean operations differs from system to system. Search is often based on proximity relations and truncation. Users are not used to free and controlled vocabulary available in tools. The commands have often specific semantics. The systems lack in efficient keyword and especially phrase search. Input and management of several search keyword is usually not incorporated.
• User interface of IR systems are often not matured. The systems use unclear and misinterpretable shortcut words. The help facility is not matured and context-sensitive. Systems are not error-prone, e.g., for typos. Systems often require specific predefined terminals. Standardized data are not formalized. Keys have changing meanings. Output formats are too large for the user. The interpretation of provided output is difficult.

• It is difficult for users to judge on relevancy. Short titles are difficult to understand. The hit rates are not understandable. Evaluation of relevancy is often too difficult.

• The selection of the most appropriate service is often impossible. The topics which are covered are not or partially given. The media types which can be obtained cannot be captured. The actuality of the provided data is not given. Some search engines use special collections without notifying the user on that.

• The vocabulary used is not unified. The user needs to know the thesaurus. The selection of an appropriate thesaurus, keyword and classification is often impossible. Formal collection rules are not provided. Request extension gives unexpected results. Services are using foreign vocabulary. Vocabulary is provider- and culture-determined.

ER is better than relational but not used: Database systems are mostly based on the relational paradigm. This choice provides high performance. However, the selection of join paths for queries which are more complex is very difficult. ER representation is much better to comprehend for several reasons.

• Complex associations can be represented by sub-schemata. Their integration into the schema is simpler. Further, since ER views are sub-schemata or generated schemata we do not need towers of views for their representation.

• There is a well-developed theory [Tha00] for ER models which omits the pitfalls of general object-oriented models and which is as powerful as the relational theory.

The application of the ER model, however, is often not appropriate. There are several reasons for that:

• ER design tools have their specific translation of ER types to relational types. This choice is not made explicit. Sometimes, the translation depends on the order of type introduction.

• ER design tools do not allow the selection of appropriate integrity enforcement strategies.

Furthermore, the logical and physical schemata initially obtained from the conceptual schema is often changed by database operators and administrators due to performance issues. This situation can be avoided if database design workbenches such as RADD are used throughout database lifespan.

Another difficulty with database design systems is caused by their specific schema representation languages which are often not public. Systems such as RADD or DBMain are have open formats.

Summarizing, ER schemata can be used in a simpler manner. This solution is - up to our knowledge - not used in the literature.

Problematic DB solutions: Database schemata are developed and redeveloped over a larger period of time. Instead of revision of existing types new types are added. For instance, SAP R/3 has more than 70 different relations which are representing the same or similar addresses instead
of using a few as proposed in [Tha00] and which consistency is maintained in different fashion. Web sites also have to use legacy databases. In this case rather specific vocabulary is used. SAP R/3 is using almost 20,000 base relations - a number which is not comprehensible especially since it has twice as much views. Such large systems need very sophisticated interfaces in order to become easily to use.

3 Our Solution

The aim of the project was the development of an Intelligent NL Request Transformer. This transformer should

- analyze natural language utterances and transform these into syntax trees,
- unify variables in the syntax tree with database content, and
- generate a set of SQL queries which can be applied to the databases used for support of the information service.

The tool has been integrated into websites by an industrial cooperation partner. They developed the input interface which is text-based. They, further, developed a tool called 'web presenter'. This tool enables the internet site to satisfy user questions by connection support to DBMS engines, sending queries to the associated database, by integrating answers obtained this way into the presentation interface. Thus, users accessing a kiosk may ask questions on the city facilities, last events in the city, on decisions of town management etc. They obtain an answer whenever the NL utterance can be resolved and matched against the database.

The architecture of the tool developed so far is displayed in Figure 1. The tool consists of the following components:

**Query liquefaction:** This component is the main component of the tool. It allows to melt the request into parts, to integrate the parts with the database and the meta-data contained in the database schema manager. It consists of three main components.

**Syntactical analysis:** The syntactical analysis sub-component is based on the NL approaches developed by [Düsl97]. It accepts NL utterances and generates a (small) number of syntax trees.

**Parser:** The parser transfers the utterance to a syntactical ground structure. We use for this the principal form of words. We use for the parser a variant of the Earley algorithm which has been proved its usefulness in the RADD project [Düsl97].

**Grammar:** ID/LP grammars have shown to be applicable to the flexible structure of German language. Therefore, we use this kind of generalized phrase structure language for this sub-component.

**Lexicon:** The lexicon contains a number of lexical units such as quantifiers, adjectives, substantives, verbs, prepositions etc. Units have their characterization by additional structural attributes (casus, numerus, genus, finiteness, valency, ...) and semantical attributes in dependence on different semantical categories.

Syntactical analysis aims transferring the utterance into a syntax tree which represents the structure of the utterance. We extended the syntax tree by nodes which represent database information. These nodes are adorned by variables which values are found in the content db. Further, the adornment of nodes is extended by ontology or lexicon information. This information is used for schema matching. Thus, the architecture of the syntactical analysis component can be roughly represented as displayed in Figure 2. The number of trees generated might be large. We use heuristics for ordering this set. Only trees with high preference are considered in the next steps.
Figure 1: The Cottbus Intelligent NL Request Transformer

Figure 2: The Syntactical Analysis Sub-Component
**Intelligent path extraction:** This sub-component uses the syntax tree and the information maintained in the database schema manager and the information provided by the thesaurus for generation of paths in the ER schema. The main functionality of this component is displayed in Figure 1.

**Graph search:** The nodes of the syntax tree which are unrelated are matched against the names of types in the ER schema. If there is no match for a certain node then thesaurus information is involved. It is also planned to integrate other lexicon information.

**Path discoverer:** After the syntax tree has been adorned by schema names and variables the ER schema is used for the discovery of paths which might be used for the corresponding query. We use classical graph algorithms [Cau00, WaN87] for path generation. The paths are ordered by a number of heuristics.

**Narrowing:** Users can be characterized by their typical behavior patterns. We can use evidence theory [Alt00] for association of specific interpretation patterns with user behavior patterns. This enables a higher flexibility of path treatment. Furthermore, we developed a number of preference patterns which can be used for generation of paths based on narrowing.

**Disambiguation:** We might still find a number of ambiguities and paths which are considered to be of equal rights. In order to add a reasoning facility in a later stage of development we added another sub-component for disambiguation. We might plug in preference profiles of user groups or of site developers. Furthermore, users might be asked based on graphical tools which of the found paths is the most appropriate for their request.

**Relational query melting-pot:** After we have generated a number of paths in the ER schema we generate now SQL queries. This generation is based on the translation style. Then the ER path can be translated to a relational path with hinge attributes. The hinge attributes are used in the `where` part of queries in order to relate different relations according to the path structure. A typical hinge structure in the example discussed below is the following enriched schema:

```sql
SPL_Person(P_ID, P_Name, P_Vorname, P_Akad_Titel, P_Telefon, P_Email, P_Aktiv)
SPL_Struktureinheit(SE_ID, SE_Kostenstelle, SE_Fakoder_Dezernat, SE_Lehrstuhloder_Sachgebiet, SE_Art, SE_Telefon, SE_Fax, SEAdresse)
SPL_gehoert_zu(P_ID -> SPL_Person, SE_ID -> SPL_Struktureinheit)
```

Finally, since computation of one-to-many paths is less complex cardinality constraints are used for detection of directions in the relational schema. If a path is partially cyclic then we use relation type variables similar to the introduction of variables in the `from` part of SQL clauses.

**DB schema manager:** The request is compiled in the liquefaction component by analyzing the utterance and matching the utterance against the thesaurus and against the database schema. Thus, the schema manager keeps the schema information in a form which enables in matching.

**ER schema storage engine:** The ER schema is used for matching schema components against the request keywords. After match we try to find appropriate paths in the schema which might be used in order to answer the request. Thus, the intelligent path extractor is based on this schema information.
ER2R translation suite: It is not very well-known that there is a large variety of translations of ER schemata to relational schemata. This variety is caused by the expressive power of the ER model. We might vary the translation of attribute types, of relationship types. For instance, relationship types can be integrated into existing relational types, can be used for the creation of a new relational type or can caused the integration of several existing types. Each of the design workbenches is using its own strategies. In order to cope with this variety we made the translation style explicit. This allows to associate the paths found in ER schemata with the paths derivable for the corresponding relational schema.

Integration of DB design workbenches: In practical application database design workbenches such as Silverrun or ErWin are used. These tools are closed and use their own database structuring which is only partially known. They use, further, specific translation techniques. Due to this partiality of knowledge we decided to use freeware tools such as DBMain or RADD. For instance, DBMain has several schema representation languages which are integrated. We decided to use ISL which is a text-based form and which is easy to parse.

DB thesaurus manager: The request usually only partially matches against the database. Users might use different keywords. These keywords might have associations with schema items, can be more general or more specific. Users might use data which are in the database. Users might use general vocabulary which has to be detected. Thus, the query liquefaction component is supported by a thesaurus component which contribute to query generation by general and database information. This thesaurus is based on the following components:

Database content extraction: The content of the database might be partially known to users. In this case, the user might use such data in the request vocabulary. In order to detect this, we pin such attributes in the database schema which data values might be used for querying. Then we can use queries of the form

\[
\text{select "relation", "attribute", attribute into content_db from relation}
\]

in order to generate triples containing the relation name, the attribute name and the corresponding value. Then the content database is matched against nouns and other specific components of utterances.

Ontology: Since users might use their vocabulary we need to be more flexible in the vocabulary we are using. Especially, users may ask for information using very general terms. For this reason we integrated a simple ontology based on the knowledge of the application domain. This ontology enables the query liquefaction component to use more specific or more general words instead of those in the schema or the database. Furthermore, users or the database developers might use abbreviations. For this reason, an abbreviation extraction facility is useful.

Lexicon integration: We have built in a facility in order to integrate a lexicon such as WordNet. This facility might be extended also to utilization of dictionaries for foreign languages. Currently this facility is rudimentary. However, it can be easily extended.

Further, the thesaurus relates natural language expressions with short-cut and specific expressions used by the database developer.

4 An Example

The system is currently used for the query interface of information sites of several towns in the region. For these systems convenient web input interfaces and web presenters have been devel-
The construction of these interfaces is out of the scope of this paper. Further, since the database schemata used for these systems are not public we use another application example in order to demonstrate the power of the tool developed. In Figure 6 the corresponding schema is displayed. It is used for university lecture scheduling. The corresponding database contains information on lectures, the people which are responsible, acting in different roles and giving lectures. Lectures might be given either on a weekly basis or grouped within a certain time interval. Lectures are scheduled for in certain rooms. For scheduling a number of restrictions must be considered, e.g. parallel lectures, room equipment, prerequisites. Structural units of the university such as departments, institutes and chairs may play different roles during scheduling and for maintenance. Some actions of users might be recorded for later proof of action. The corresponding person must have the rights to use these actions.

Let us consider some sample requests for illustration of the approach. A user might ask the system the following questions:

- *Which events are read by Thalheim and Vierhaus? (Welche Veranstaltungen lesen Thalheim und Vierhaus?)*

This request uses the logical connective $\land$ instead of the correct $\lor$. There are no lectures given together by the two professors. Using narrowing and the user profile the system finds that the preferred interpretation is $\lor$.

The analysis of the sentence is first based on a comparison with database data. The words ‘Vierhaus’ and ‘Thalheim’ are found in the database. Thus using variables tktktktk, the sentence is translated to

```
Welche Veranstaltungen lesen tktktktk und tktktktk.
```

Now we generate syntax trees. Their number reaches in the case of the example 23. The final syntax tree is the following:

```
s [praes]
 np [akk,plural,3,noun]
  quant [akk,plural,fem,finit]
  welch [akk,plural,fem,finit]
  n [akk,plural,fem,finit,3,noun]
  noun [akk,plural,fem,finit,3]
  Veranstaltung [akk,plural,fem,finit,3]
 vp [{np,[n,gen,sing,3,noun]},{plural,3,praes}]
  v [vf,[nouns,haben],[finit,nopsp},{plural,3,praes,nopraes}]
  les [vf,[nouns,haben],[finit,nopsp},{plural,3,praes,nopraes}]
 conn []
 np [gen,sing,3,noun]
  n [gen,sing,mas,infin,3,noun]
  noun [gen,sing,mas,infin,3]
  tktktktk [gen,sing,mas,infin,3]
 conn []
 und []
 np [akk,plural,3,noun]
  n [akk,plural,mas,infin,3,noun]
  noun [akk,plural,mas,infin,3]
  tktktktk [akk,plural,mas,infin,3]
```

Now, correctness of the translation is tested. We use retranslation techniques. The sample sentence has the following form:

```
"quant,welch noun,veranstaltung verb,les noun,tktktktk conn,und noun,tktktktk"
```

Now the request terms are unified with types of the ER schema.
This unification allows to identify two paths:

1. Veranstaltung les Thalheim
2. Veranstaltung les Vierhaus

For each path the corresponding paths in the ER schema are detected. We prefer shortest paths. Thus, for the sub-path Veranstaltung les the following shortest path is obtained:

\[\text{SPL}_{\text{Veranstaltung}} \rightarrow \text{SPL}_{\text{LV Info}} \rightarrow \text{SPL}_{\text{Lehrveranstaltung}} \rightarrow \text{SPL}_{\text{wird durchgefuert von}}.\]

For the path Veranstaltung les Thalheim the following path is obtained:

\[\text{SPL}_{\text{Veranstaltung}} \rightarrow \text{SPL}_{\text{LV Info}} \rightarrow \text{SPL}_{\text{Lehrveranstaltung}} \rightarrow \text{SPL}_{\text{wird durchgefuert von}} \rightarrow \text{SPL}_{\text{Lehrender}} \rightarrow \text{SPL}_{\text{Person}}.\]

The next step uses the translation style which has been applied for translating the ER schema to the relational schema. The types belonging to the begin and to the end of the path are used for the select attributes of the query. All other attributes are cut out. The obtained SQL query for the first path is:

```sql
SELECT
    SPL_{Veranstaltung}._LV_ID,
    SPL_{Veranstaltung}._LV_Titel,
    SPL_{Veranstaltung}._LV_Art,
    SPL_{Veranstaltung}._LV_SWS_Asz,
    SPL_{Veranstaltung}._LV_Stand,
    SPL_{Veranstaltung}._LV_Kurzinhalt,
    SPL_{Veranstaltung}._LV_Langtext_URL,
    SPL_{Veranstaltung}._LV_Abschluesselzertifikat,
    SPL_{Person}.P_ID, SPL_{Person}.P_Name, SPL_{Person}.P_Vorname,
    SPL_{Person}.P_Akadem_Titel, SPL_{Person}.P_Telefon,
    SPL_{Person}.P_EMail, SPL_{Person}.P_Aktiv
FROM SPL_{Person}, SPL_{wird durchgefuert von}, SPL_{Lehrender},
SPL_{Veranstaltung}, SPL_{Lehrveranstaltung}
WHERE
    SPL_{Lehrender}.P_ID = SPL_{Person}.P_ID
AND SPL_{wird durchgefuert von}.P_ID = SPL_{Lehrender}.P_ID
AND SPL_{Lehrveranstaltung}.LV_Titel_ID =
    SPL_{wird durchgefuert von}.LV_Titel_ID
AND SPL_{Lehrveranstaltung}.LV_ID = SPL_{Veranstaltung}.LV_ID
AND SPL_{Person}.P_Name = 'Thalheim'
```

Correspondingly, the second path generates the following SQL query:

```sql
SELECT
    SPL_{Veranstaltung}._LV_ID,
    SPL_{Veranstaltung}._LV_Titel,
    SPL_{Veranstaltung}._LV_Art,
    SPL_{Veranstaltung}._LV_SWS_Asz,
    SPL_{Veranstaltung}._LV_Stand,
    SPL_{Veranstaltung}._LV_Kurzinhalt,
    SPL_{Veranstaltung}._LV_Langtext_URL,
    SPL_{Veranstaltung}._LV_Abschluesselzertifikat,
    SPL_{Person}.P_ID, SPL_{Person}.P_Name,
    SPL_{Person}.P_Vorname, SPL_{Person}.P_Akadem_Titel,
    SPL_{Person}.P_Telefon, SPL_{Person}.P_EMail, SPL_{Person}.P_Aktiv
FROM SPL_{Person}, SPL_{wird durchgefuert von}, SPL_{Lehrender},
SPL_{Veranstaltung}, SPL_{Lehrveranstaltung}
WHERE
    SPL_{Lehrender}.P_ID = SPL_{Person}.P_ID
AND SPL_{wird durchgefuert von}.P_ID = SPL_{Lehrender}.P_ID
AND SPL_{Lehrveranstaltung}.LV_Titel_ID =
    SPL_{wird durchgefuert von}.LV_Titel_ID
AND SPL_{Lehrveranstaltung}.LV_ID = SPL_{Veranstaltung}.LV_ID
AND SPL_{Person}.P_Name = 'Vierhaus'
```
The screen shot in Figure 3 shows the administrators information.

Figure 3: Which events are read by Thalheim and Vierhaus?

- Which events are organized for the studies in architecture by the head of the chair ‘statics’? (Welche Veranstaltungen für den Studiengang Architektur hält der Lehrstuhlinhaber des Lehrstuhls Statik?)

The sentence

Welche Veranstaltungen für den Studiengang Architektur hält der Lehrstuhlinhaber des Lehrstuhls Statik?

is analyzed by the system on the basis of the database schema and the thesaurus. In the schema there are two paths and thus two queries which correspond to this request:

1. The first path extracted from the schema and thesaurus information is the path
   SPL_Veranstaltungstyp, SPL_LV_Info, SPL_LV_Info, SPL_LV_Info, SPL_Professor, SPL_LV_Info, SPL_LV_Info, SPL_LV_Info, SPL_Professor.

2. The second path extracted from the schema and thesaurus information is the path
   SPL_Lehrveranstaltung, SPL_Professor, SPL_LV_Info, SPL_LV_Info, SPL_Professor, SPL_LV_Info, SPL_LV_Info, SPL_Professor.

These paths express that a certain lecture belongs to architecture. It is directly translated to the query

select SPL_Lehrveranstaltung.LV_ID, SPL_Lehrveranstaltung.LV_Titel, SPL_Lehrveranstaltung.LV_Art, SPL_Lehrveranstaltung.LV_SZ.

11
The Cottbus Intelligent NL Request Transformer allows to analyze NL utterances, to compare the syntax tree with the vocabulary contained in the thesaurus and with the database schema. The
Figure 4: Which people belong to the databases chair?

au tk 7 (...)/diplom/SQL-generator) : echo "Welche Person gehört zu Iatenbanken,"
1 ./src/syntax-analyse/sql-gen
Anfrage: Welche Person gehört zu Iatenbanken.
1. Pfad: Person gehör Iatenbanken
   1. Weg: SPL_Person -> SPL_gehoert_zu -> SPL_Struktureinheit

au tk 8 (...)/diplom/SQL-Generator) :

Figure 5: On which days opens the department?

au tk 21 (...)/diplom/SQL-generator) : echo "Welche Tag öffnet das Dezernat."
1 ./src/syntax-analyse/sql-gen
Anfrage: Welche Tag öffnet das Dezernat.
1. Pfad: Tag öffn Dezernat
   1. Weg: tbl_tage -> tbl_vwltg_oeffnung -> tbl_vwltg_dezernat
      SQL-Query: select tbl_tage.tag_id, tbl_tage.tagname, tbl_tage.farbe, tbl_vwlt g_dezernat.dez_id, tbl_vwltg_dezernat.dezname, tbl_vwltg_dezernat.dez_leiter fro m tbl_vwltg_oeffnung, tbl_vwltg_dezernat, tbl_tage where tbl_vwltg_oeffnung.dez_id = tbl_vwltg_dezernat.dez_id and tbl_tage.tag_id = tbl_vwltg_oeffnung.tag_id

au tk 22 (...)/diplom/SQL-generator) :
The thesaurus uses data extracted from the database supporting the internet service and additional lexicons. Furthermore, the thesaurus uses an ontology which allows to look for more general categories. Based on this analysis we can derive a number of candidate paths through the schema with some unification conditions. This information is used for generating SQL queries which are applicable to the database system. The Transformer has been integrated into city information sites where users can ask questions in natural language.

Our development has shown that this approach allows sophisticated support for casual users of internet sites. The approach needs further refinement in different directions: broader ontology, selection of appropriate parts of the database for detection of useful vocabulary, support for decomposition of query answers according user profiles. The tool has been applied to information sites. We are currently working on applications to other sites.

References


Figure 6: The DBMain Schema for Scheduling of University Lectures
Semantics and Acquisition of Constraints
Semantikakquisition im Datenbankentwurf und Reverse-Engineering *

Meike Albrecht, Bernhard Thalheim

Technische Universität Cottbus
Fachbereich Informatik

e-mail: meike@informatik.uni-rostock.de/ thalheim@informatik.tu-cottbus.de

Abstract

Die vollständige und richtige Angabe semantischer Constraints ist besonders für große Anwendungen eine der schwierigsten Aufgaben im Datenbankentwurf. Das gleiche Problem tritt im Reverse-Engineering auf, hat man bestehende Datenbanken und möchte sie in andere Modelle überführen, so muß an erster Stelle eine Analyse der Daten (Ermittlung der geltenden semantischen Constraints) stehen.
In diesem Artikel wird ein Vorgehen gezeigt, das vollständig informal ist und aus Beispielen und über Heuristiken Constraints finden und anhand eines Beispieldialogs validiert.
Diese Methode ist sowohl für den Datenbankentwurf als auch für das Reverse-Engineering einsetzbar.

Keywords: Semantik-Reverse-Engineering für Datenbanken, Schlüssel, funktionale Abhängigkeiten, Inklusions- und Exklusionsabhängigkeiten, maschinelles Lernen

1 Einleitung


Die formale Angabe semantischer Constraints ist eine der schwierigsten Aufgaben im Datenbankentwurf. In dem hier vorgestellten Zugang beschränken wir uns auf die im Datenbankentwurf am häufigsten verwendeten Constraints (Schlüssel, funktionale Abhängigkeiten, Inklusions- und Exklusionsabhängigkeiten, Kardinalitäten). Die Schwierigkeit der Erfragung dieser Constraints ist auf folgende Probleme zurückzuführen:

- Fast alle Entwurfsmethoden erwarten vom Entwerfer eine vollständige Angabe der Semantik. Normalisierungsalgorithmen liefern z.B. nur dann eine korrekte Normalform, wenn die Menge der funktionalen (und mehrwertigen) Abhängigkeiten vollständig angegeben wurde und alle nicht angegebenen Abhängigkeiten sich entweder aus der angegebenen Menge ableiten lassen oder nicht gelten.
- Oft werden aber einige semantische Constraints vergessen, obwohl oder auch weil sie offensichtlich sind, oder semantische Constraints sind zu kompliziert, besonders, wenn sie über mehreren Attributen definiert sind, und werden deshalb entweder gar nicht gesehen oder falsch angegeben.

Wünschenswert wäre also eine Unterstützung bei der Semantikakquisition, bei der dem Entwerfer semantische Constraints vorgeschlagen werden. Dabei treten weitere Probleme auf:

- Insbesondere bei komplizierten semantischen Constraints, die über mehreren Attributen definiert sind, ist es schwer zu entscheiden, ob diese gelten oder nicht.
- Nicht alle Constraints können in einfacher und unmissverständlicher Form graphisch dargestellt werden.
- Es müssen potentiell exponentiell viele mögliche semantische Constraints untersucht werden, um die vollständige Menge semantischer Constraints zu finden.

*Diese Arbeit wird durch das DFG-Projekt Th465/2 unterstützt.
2 THEORETISCHE GRUNDLAGEN


2.1 Grundlagen der relationalen Datenbanken


Eine *relationale Datenbasis* (oder *Relationsschema*) \( RS = (U, D, \text{dom}, I) \) wird durch eine endliche Menge von Attributen \( U \), eine Menge von Wertebereichen \( D \) und eine Funktion \( \text{dom} : U \rightarrow D \), die jedes Attribut mit seinem Wertebereich verbindet, und eine Menge von Integritätsbedingungen \( I \) definiert.

Ein Tupel in einem Relationsschema \( RS = (U, D, \text{dom}, I) \) ist eine Funktion \( t : U \rightarrow D \), mit \( t(A) \in \text{dom}(A), \forall A \in U \). Jedes Tupel repräsentiert ein Entity in der realen Welt.

Eine Relation \( r \) des Relationsschemas \( RS \) ist eine Teilmenge der Menge aller Tupel über \( RS \), die die Integritätsbedingung \( I \) erfüllt.

Die Projektion eines Tupels auf eine Menge von Attributen \( X \) (geschrieben \( t[X] \)) ist eine einstellige Operation auf einem Tupel. Dabei werden die Attribute, die nicht in der Attributmenge \( X \) enthalten sind, aus dem Tupel entfernt.

2.2 Semantische Constraints


Die dazu am häufigsten verwendeten Integritätsbedingungen werden im folgenden definiert.

2.2.1 Definitionen semantischer Constraints

Eine der wichtigsten Integritätsbedingungen in relationalen Datenbanken stellen *funktionale Abhängigkeiten* dar.

**Definition 1** Sei \( V \) die Attributmenge eines Relationsschemas \( RS \) und \( X, Y \subseteq V \). Eine *funktionale Abhängigkeit* \( X \rightarrow Y \) gilt, wenn für alle konsistenten Relationen \( r \) von \( RS \) folgende Bedingung gilt:

\[
\forall m, n \in r: m[X] = n[X] \rightarrow m[Y] = n[Y].
\]

Ein *Schlüssel* ist die minimale Attributkombination, die ein Entity in jeder Relation eines Relationenschemas identifizierbar macht. Schlüssel in Relationen werden wie folgt definiert:

**Definition 2** Sei \( U \) die Attributmenge eines Relationsschemas \( RS \) mit \( X \subseteq U \). \( X \) heißt *Schlüssel*, wenn für alle konsistenten Relationen von \( RS \) folgende Bedingung gilt:

\[
\forall t, t' \in r, t[X] \neq t'[X] \text{ und } \exists X' \subseteq X \text{ mit } t[X'] \neq t'[X'].
\]


**Definition 3** \( R \) und \( S \) sind Relationenschemata eines Datenbankschemas \( D \), \( X \) ist eine Attributsequenz von \( R \), \( X = (X_1, \ldots, X_n) \), \( Y \) ist eine Attributsequenz von \( S \), \( Y = (Y_1, \ldots, Y_m) \). Eine *Inklusionsabhängigkeit* \( R.X \subseteq S.Y \) ist erfüllt, wenn für alle konsistenten Relationen \( r \) von \( R \) und \( s \) von \( S \) eines Datenbankschemas \( D \) gilt:

\[
\forall t \in r \text{ gibt es ein Tupel } t' \in s, \text{ für das } t[X_i] = t'[Y_i], \forall 1 \leq i \leq n.
\]

Deshalb ermittelt der hier vorgestellten Ansatz neben Schlüsseln, funktionalen Abhängigkeiten und Inklusionsabhängigkeiten auch Exklusionsabhängigkeiten.

Definition 4 \( R \) und \( S \) sind Relationenschemas eines Datenbankschemas \( D \), \( X \) ist eine Attributmenge von \( R \), \( X = \{X_1, \ldots, X_n\} \), \( Y \) eine Attributmenge von \( S \), \( Y = \{Y_1, \ldots, Y_n\} \). Eine Exklusionsabhängigkeit \( R.X \not\rightarrow S.Y \) gilt, wenn es keine konsistenten Relationen \( r \) von \( R \) und \( s \) von \( S \) mit \( t \in r \) und \( t' \in s \) gibt, für die \( t[X_i] = t'[Y_i] \) gilt für \( 1 \leq i \leq n \).

### 2.2.2 Behandlung semantischer Constraints

In der Semantikakquisition kennt man sowohl einige geltende Constraints als auch Constraints, die nicht gelten. Es ist sinnvoll, diese nicht geltenden Constraints getrennt zu speichern, um sie von den noch unbekannten Constraints unterscheiden zu können. Damit wird keine Closed-World-Assumption vorausgesetzt.

Ein Constraint ist noch unbekannt, wenn es sich weder aus der Menge der geltenden, noch aus der Menge der nicht geltenden Constraints ableiten läßt.


Für die Konsistenzprüfung und Normalisierung interessieren nur die geltenden Constraints. Für das Finden dieser müssen jedoch auch die nicht geltenden semantischen Constraints betrachtet werden, da sich der Suchraum zur Ermittlung der Constraints einschränkt.

### 2.2.3 Axiomatisierung

Funktionale Abhängigkeiten sind axiomatisierbar [Ull 88]. Es gibt eine Menge von vollständigen und richtigen Ableitungsregeln, den sogenannten Armstrongableitungsregeln. Nicht geltende funktionale Abhängigkeiten sind ebenfalls axiomatisierbar [Jan 89].

Durch Schlüssel werden in einem Relationenschema funktionale Abhängigkeiten bedingt. Ein Schlüssel macht die Entities, die in einer Relation dargestellt sind, identifizierbar. Da die einzelnen Tupel Entities darstellen, sind auch die Attribut einer jeden Tupels durch den Schlüssel bestimmt.

Aus der eindeutigen Identifikation ergibt sich, daß nach folgender Vorschrift funktionale Abhängigkeiten aus dem Schlüssel abgeleitet werden können. \( V \) sei die Attributmenge einer Relation. \( X \) sei die Menge der Schlüsselattribute der Relation, \( X \subseteq V \). Dann gilt \( \forall A \in V : X \rightarrow A \).

Das heißt, alle Attribute einer Relation sind von den Schlüsselattributen funktional abhängig.

Inklusionsabhängigkeiten sind axiomatisierbar [MaR 93]. Funktionale Abhängigkeiten und unäre Inklusionsabhängigkeiten sind ebenfalls axiomatisierbar, funktionale Abhängigkeiten und höhere Inklusionsabhängigkeiten sind nur bei nicht-cirkulären Abhängigkeiten axiomatisierbar [MaR 93], [Mit 83].

Inklusions- und Exklusionsabhängigkeiten sind ebenfalls axiomatisierbar [CaV 83].

### 3 Allgemeines Vorgehen bei der Semantikakquisition


Aus bekannten Instanzen können einige semantische Informationen abgeleitet werden. Welche semantischen Constraints aus Instanzen ableitbar sind und unter welchen Bedingungen sie zu erkennen sind, wird im nächsten Kapitel erläutert.

Aus Instanzen lassen sich jedoch keine vollständigen Informationen über die Semantik der Relationen ableiten. Es lassen sich jedoch Kandidaten für Constraints mit Hilfe von Heuristiken finden. In Kapitel 5 wird gezeigt, über welche Heuristiken Kandidaten für Constraints gefunden werden und wie die einzelnen Heuristiken kombiniert werden.

In der Menge der Constraintkandidaten können falsche oder redundante Informationen auftreten. Diese müssen ermittelt und aus der Menge der zu untersuchenden Kandidaten gelöscht werden. Welche Untersuchungen dabei für jeden ermittelten Kandidaten erforderlich sind, ist in Kapitel 6 angegeben.

Das nächste Kapitel erläutert, wie Kandidaten für Constraints effektiv und informal validiert werden. Dazu werden bekannte Ansätze vorgestellt und ein eigener Ansatz vorgeschlagen.
4 Ableitung semantischer Constraints aus Instanzen

Aus Instanzen einer Datenbank lassen sich einige Aussagen über die Semantik der Datenbank ableiten. Die folgende Übersicht zeigt, über welchen Instanzen die einzelnen in Kapitel 2 beschriebenen Constraints definiert wurden.

<table>
<thead>
<tr>
<th>zwei Tupel einer Instanz</th>
<th>eine Instanz</th>
<th>alle konsistenten Instanzen eines Datenbankschemas</th>
</tr>
</thead>
<tbody>
<tr>
<td>negierte Schlüssel</td>
<td>Schlüssel</td>
<td>Schlüssel</td>
</tr>
<tr>
<td>afunktionale Abhängigkeiten</td>
<td>funktionale Abhängigkeiten</td>
<td>Funktionale Abhängigkeiten</td>
</tr>
<tr>
<td>negierte Inklusionsabhängigkeiten</td>
<td>Inklusionsabhängigkeiten</td>
<td>Inklusionsabhängigkeit</td>
</tr>
<tr>
<td>negierte Exklusionsabhängigkeiten</td>
<td>Exklusionsabhängigkeit</td>
<td>Exklusionsabhängigkeit</td>
</tr>
</tbody>
</table>
Die Ableitung von Constraints aus Instanzen kann so erfolgen:

- **negierte Schlüssel und afunktionale Abhängigkeiten**
  Wenn in einer Relation in zwei Tupeln der gleiche Wert bei einem oder mehreren Attributen vorkommt, so kann man ableiten, daß diese Attribute kein Schlüssel der Relation sein können, da sie bereits in dieser einen Relation nicht alle Tupel identifizieren können.

<table>
<thead>
<tr>
<th>Personennummer</th>
<th>Nachname</th>
<th>Vorname</th>
<th>Wohnort</th>
<th>PLZ</th>
<th>Straße</th>
<th>Nummer</th>
</tr>
</thead>
<tbody>
<tr>
<td>218263</td>
<td>Meier</td>
<td>Ilona</td>
<td>Berlin</td>
<td>10249</td>
<td>Mollstr.</td>
<td>149</td>
</tr>
<tr>
<td>948547</td>
<td>Mueller</td>
<td>Karl</td>
<td>Berlin</td>
<td>12621</td>
<td>Mittelweg</td>
<td>281</td>
</tr>
<tr>
<td>323983</td>
<td>Schmidt</td>
<td>Klaus</td>
<td>Rostock</td>
<td>18055</td>
<td>Gerberbruch</td>
<td>30</td>
</tr>
<tr>
<td>239283</td>
<td>Weber</td>
<td>Peter</td>
<td>Rostock</td>
<td>18055</td>
<td>Gerberbruch</td>
<td>15</td>
</tr>
</tbody>
</table>

ableitbare nicht geltende Schlüssel
- Nachname
- Vorname Wohnort
- Wohnort PLZ Straße

ableitbare afunktionale Abhängigkeiten
- Nachname ↔ Personennummer
- Nachname ↔ Vorname
- Nachname ↔ Wohnort
- Nachname ↔ PLZ
- Nachname ↔ Straße
- Nachname ↔ Nummer
- Wohnort PLZ Straße ↔ Personennummer
- Wohnort PLZ Straße ↔ Nachname
- Wohnort PLZ Straße ↔ Nummer
- Wohnort PLZ ↔ Straße

Aus dieser Relation läßt sich ableiten, daß folgende Attribute nicht Schlüssel sein können:
Weiterhin können Teilmengen dieser angegebenen Mengen keine Schlüssel sein, diese negierten Schlüssel lassen sich aus den oberen ableiten.

Für die obige Relation läßt sich folgende nichtredundante Menge von afunktionalen Abhängigkeiten ableiten:

Der Algorithmus dazu sieht so aus:

```
FOR t1:=1 TO Tupelanz-1 DO
  FOR t2:=i TO Tupelanz DO
    (* In der Menge m werden die Attribute gesammelt,
    die gleiche Einträge in t1 und t2 haben *)
    m:= { };
    FOR a := 1 TO Attributanz DO
      IF (t1[a]= t2[a] THEN
        (* Aufnahme von a in m *)
        m:= m ∪ a;
      END;
    END;
    FOR a := 1 TO Attributanz DO
      IF (a ∉ m) THEN
        ⟨ Bilden von rechtsminimalen afunktionalen Abhängigkeiten
          mit m auf der linken Seite und a auf der rechten Seite ⟩
        ⟨ Abspeicherung der gebildeten afunktionalen Abhängigkeiten,
          soweit noch nicht vorhanden ⟩
      END;
    END;
  END;
END;
```

In dem Algorithmus müssen $m^2-m*n$ Vergleichsoperationen durchgeführt werden, wobei $n$ die Anzahl der Attribute ist und $m$ die Anzahl der Tupel der Relation ist.
negierte Inklusionsabhängigkeiten

Über allen Attributpaaren der Datenbank, die gleiche Typen und ähnliche Längen haben, wird versucht, negierte Inklusionsabhängigkeiten abzuleiten. Negierte Inklusionsabhängigkeiten gelten nach Definition ?? zwischen zwei Attributen, wenn Werte eines Attributes nicht in der Menge der Werte des anderen Attributes auftreten. Negierte Inklusionsabhängigkeiten lassen sich jedoch nur ableiten, wenn man vollständige Instanzen für die rechte Seite der Abhängigkeit auswerten kann.

negierte Exklusionsabhängigkeiten

Treten in der Datenbank Attribute oder Attributsequenzen auf, in denen gleiche Einträge stehen, so hat man nach Definition ?? negierte Exklusionsabhängigkeiten zwischen diesen Attributen gefunden. Auch hier ist ein Wertvergleich nur sinnvoll, wenn die Attribute gleiche Typen und gleiche oder ähnliche Längen haben.


5 Ermittlung von Kandidaten für Constraints

Man kann nicht alle noch unbekannten Constraints untersuchen, da diese Anzahl sehr groß sein kann. Die Anzahl der voneinander unabhängigen funktionellen Abhängigkeiten einer Relation liegt in der Ordnung $O(2^n)$, wobei $n$ die Anzahl der Attribute der Relation ist. Die Anzahl möglicher Schlüssel ist $2^n$. In einer Datenbank müssen $n \times (n - 1)$ unäre Inklusions- und Exklusionsabhängigkeiten untersucht werden, wobei $n$ hier die Anzahl der Attribute der Datenbank ist.


5.1 Heuristiken zur Schlüsselsuche

In Datenbanken kann es verschiedene vage Hinweise auf Schlüssel geben, die durch ein Tool ausgewertet werden können:

1. **Auswertung der Attributtypen und -längen**
   Oft werden in Datenbanken künstliche Schlüssel verwendet. Wenn ein Attribut mit dem Typ Integer und einer sehr großen Länge vereinbart wurde, so könnte dieses ein künstlicher Schlüssel sein.

2. **Auswertung der Attributnamen**

3. **Auswertung der Instanzen**
   Sind in den Daten der Instanzen bei einem Attribut laufende Nummern eingetragen, so deutet das stark darauf hin, daß dieses Attribut ein künstlicher Schlüssel ist. Diese Heuristik wurde in [ChV 92] auf Formulare angewendet, sie kann auf Relationen übertragen werden.

4. **Auswertung der semantischen Constraints**
   Sind bereits funktionale und afunktionsale Abhängigkeiten bekannt, so kann man daraus Informationen über Schlüssel ableiten.
   
   Folgende Abhängigkeiten deuten darauf, daß ein Attribut $A$ Schlüssel oder Teil eines Schlüssels ist:
   
   - nichttriviale funktionale Abhängigkeiten $X \rightarrow Y, A \subseteq X$
   - afunktionale Abhängigkeiten $Y \not\rightarrow A$.
   
   Das Auftreten folgender Abhängigkeiten spricht dagegen, daß $A$ Schlüssel oder Teil eines Schlüssels ist:
   
   - nichttriviale funktionale Abhängigkeiten $Y \rightarrow A$. 
   
   Folgende Abhängigkeiten deuten darauf, daß ein Attribut $A$ Schlüssel oder Teil eines Schlüssels ist:
   
   - nichttriviale funktionale Abhängigkeiten $X \rightarrow Y, A \subseteq X$
   - afunktionale Abhängigkeiten $Y \not\rightarrow A$.
5.2 Heuristiken zur Suche nach funktionalen Abhängigkeiten

• afunktionale Abhängigkeiten \( X \not\rightarrow Y, A \subseteq X \).

Durch Untersuchung dieser Abhängigkeiten wird ein Heuristikfaktor für jedes Attribut berechnet. Der so berechnete Faktor ist besonders aussagekräftig, weil er sich im Verlaufe der Semantikakquisition ändert und die Aussagen dadurch besser werden. Auf diese Weise ist eine lernende Komponente in den Heuristikregeln enthalten.

5. Auswertung der Transaktionen

Auch aus bekannten Transaktionen lassen sich Vermutungen über Schlüssel der Relationen ableiten. Die Attribute, die bei Updates nie oder nur sehr selten verändert werden, sind mit größerer Wahrscheinlichkeit Schlüssel- oder Teil des Schlüssels als Attribute, die oft geändert werden. Attribute, die bei Update- und Deleteoperationen zur Identifikation von Tupeln verwendet werden, können Schlüssel der Relation sein.

Diese Heuristikregeln werden für alle Attribute der Relation untersucht und ausgewertet. Die einzelnen Heuristiken sollen dabei gegeneinander gewichtet werden, aus diesen Heuristikregeln soll eine Abschätzung für Schlüsselkandidaten getroffen werden. Attribute, für die mehrere Heuristiken erfüllt waren, sind eher Schlüssel als Attribute, auf die weniger Heuristiken hinweisen.

Folgende einfache Näherungsformel wird zur Schlüsselkandidatensuche eingespürt, die diese angegebenen Forderungen erfüllt:

\[
P (A \text{ ist Schlüsselattribut}) := w_1 r_1 (A) + w_2 r_2 (A) + w_3 r_3 (A) + w_4 r_4 (A) + w_5 r_5 (A)
\]

\[
w_i \text{ sind die Gewichte zwischen } 0..100 \text{ und } w_1 + w_2 + w_3 + w_4 + w_5 = 100.
\]

Diese Formel kann nicht exakt sein, da die einzelnen Heuristikregeln nicht voneinander unabhängig sind. Die Berechnung genügt aber den Anforderungen an die Abschätzung.


Für die Abschätzung, ob eine Attributmenge Schlüssel ist, wird folgende Näherung verwendet:

\[
P(X \text{ ist Schlüssel}) := \left( \frac{1}{n} \sum_{i=1}^{n} P(X_i \text{ ist Schlüsselattribut}) \right)
\]

Die Verwendung des Mittelwertes für die Abschätzung der Schlüsselfähigkeiten ist ebenfalls eine Vereinfachung, weil die Ergebnisse von Bedingung 4 und 5 für eine Attributmenge anders ausfallen als der Mittelwert der für alle Attribute dieser Menge. Der Aufwand bei der Berechnung der Heuristikfaktoren wird jedoch auf diese Weise verringert, da so die einzelnen Heuristikregeln nur einmal für jedes Attribut ausgewertet werden müssen.

Es gibt eine weitere Möglichkeit, um weitere Schlüsselkandidaten aus der bereits bekannten Semantik abzuleiten.

• Auswertung der Schlüsselinformationen anderer Knoten
Tritt in der Relation eine Attributmenge auf, die in einer anderen Relation bereits als Schlüssel bestimmt wurde, so kann die Schlüsselfähigkeit auch in dieser Relation gelten und wird deshalb untersucht.

Durch diese Heuristik werden ebenfalls Kandidaten für Schlüssel abgeleitet, die validiert werden müssen.

5.2 Heuristiken zur Suche nach funktionalen Abhängigkeiten


• Ermittlung berechenbarer Attribute
Gibt es Relationen, in denen mehrere Attribute als Integer definiert sind, so wird überprüft, ob sich die Werte eines Attributes (C) aus zwei anderen Attribute (AB) durch Addition oder Multiplikation errechnen lassen. Ist das der Fall, so wird die funktionale Abhängigkeit \( AB \rightarrow C \) vermutet.

Das gleiche gilt, wenn die Werte eines Attributes B für alle Tupel einen festen Prozentsatz der Werte des Attributes A darstellen. In diesem Fall wird \( A \rightarrow B \) Kandidat für eine funktionale Abhängigkeit.
Auswertung gleicher Werte in den Instanzen
Man kann aus Instanzen afunktionale Abhängigkeiten ableiten, dieses wurde im Kapitel 4 gezeigt. Über alle anderen möglichen Constraints können noch keine Aussagen getroffen werden. Sie können entweder funktionale Abhängigkeiten sein, die in allen richtigen Relationen gelten müssen, oder es können afunktionale Abhängigkeiten sein, die noch nicht erkannt wurden, weil in den Instanzen kein Beispiel aufgetreten ist, dass die entsprechende funktionale Abhängigkeit widerlegt.

Diese beiden Fälle kann man durch folgende Heuristik unterscheiden.
Es wird untersucht, welche Attributmengen gleiche Einträge in mehreren Tupeln aufweisen. Dieses Vorgehen soll an einem Beispiel erläutert werden:

<table>
<thead>
<tr>
<th>Personennummer</th>
<th>Nachname</th>
<th>Vorname</th>
<th>Wohnort</th>
<th>PLZ</th>
<th>Straße</th>
<th>Nummer</th>
</tr>
</thead>
<tbody>
<tr>
<td>218263</td>
<td>Meier</td>
<td>Ilona</td>
<td>Berlin</td>
<td>10249</td>
<td>Mollstr.</td>
<td>149</td>
</tr>
<tr>
<td>948547</td>
<td>Mueller</td>
<td>Karl</td>
<td>Berlin</td>
<td>12621</td>
<td>Mittelweg</td>
<td>281</td>
</tr>
<tr>
<td>323983</td>
<td>Schmidt</td>
<td>Klaus</td>
<td>Rostock</td>
<td>18055</td>
<td>Gerberbruch</td>
<td>30</td>
</tr>
<tr>
<td>239283</td>
<td>Weber</td>
<td>Peter</td>
<td>Rostock</td>
<td>18055</td>
<td>Gerberbruch</td>
<td>15</td>
</tr>
</tbody>
</table>

In diesem Beispiel tritt bei (Wohnort PLZ Straße) 1 gleicher Eintrag und bei (Wohnort PLZ) 2 gleiche Einträge in der Instanz auf. Aus diesen gleichen Einträgen lassen sich folgende Kandidaten für funktionale Abhängigkeiten ableiten, die im Verlaufe der Validierung untersucht werden:

(Wohnort $\rightarrow$ PLZ)
PLZ $\rightarrow$ Wohnort
(Wohnort $\rightarrow$ Straße)
(PLZ $\rightarrow$ Straße)
Straße $\rightarrow$ Wohnort
Straße $\rightarrow$ PLZ
(Wohnort PLZ $\rightarrow$ Straße)
Wohnort Straße $\rightarrow$ PLZ
PLZ Straße $\rightarrow$ Wohnort

Die eingeklammerten Kandidaten sind bereits durch die gleiche Relation widerlegt.

Auswertung von Transaktionen
Aus Updateoperationen einer Datenbank lassen sich ebenfalls Kandidaten für funktionale Abhängigkeiten erkennen. Die Attribute einer Updateoperation, die zur Identifikation der zu ändernden Tupel verwendet wurden, bilden dabei die linke Seite der Abhängigkeit, die Attribute, die in der Operation geändert wurden, die rechte Seite. Diese abgeleitete Abhängigkeit muß in der Instanz gegolten haben, auf die die Transaktion angewendet worden ist. Man kann deshalb vermuten, daß diese funktionale Abhängigkeit auch allgemeingültig ist, sie muß jedoch ebenfalls validiert werden.

Auswertung der funktionalen Abhängigkeiten
Die letzte Heuristik zur Suche nach funktionalen Abhängigkeiten geht von folgender Überlegung aus:

Wurde bereits eine funktionale Abhängigkeit $(X \rightarrow Y)$ gefunden, so wird zwischen der linken Seite $(X)$ und allen Attributen von $(Z = U - X - Y)$ versucht, weitere rechtsminimale funktionale Abhängigkeiten abzuleiten. Deshalb werden alle so gebildeten Kandidaten untersucht.

Auswertung der interrelationalen Semantik
Analoga und FD $\Rightarrow$ weitere FD

Jede einzelne dieser Heuristiken liefert wichtige Hinweise auf funktionale Abhängigkeiten. Auch hierbei sollen am stärksten die Kandidaten für funktionale Abhängigkeiten gewichtet werden, bei denen mehrere dieser Heuristiken erfüllt waren. Dazu wird wieder eine gewichtete Summe verwendet:

$$ P(X \rightarrow Y) := w_1 r_1(X, Y) + w_2 r_2(X, Y) + w_3 r_3(X, Y) + w_4 r_4(X, Y) $$

$r_i$ - Ergebnis der Heuristikregel $i \in [0..1]$  
$w_i$ sind die Gewichte zwischen 0..100 und $w_1 + w_2 + w_3 + w_4 = 100$
Afunktionale Abhängigkeiten sind als Negation der funktionalen Abhängigkeiten definiert, deshalb können die Heuristiken auch zur Abschätzung einer afunktionalen Abhängigkeit verwendet werden.

\[ P(X \not \rightarrow Y) := 100 - P(X \rightarrow Y) \]

Auf diese Weise erhält man gewichtete Kandidaten für funktionale und afunktionale Abhängigkeiten.

5.3 Heuristikregeln zur Analogasuche


Inklusions- und Exklusionsabhängigkeiten sind nur über Attributen mit gleicher Bedeutung sinnvoll. Deshalb muß zuerst nach solchen Attributen gesucht werden.

Dabei müssen sowohl Synonyme als auch Homonyme berücksichtigt werden. *Synonyme* sind Attribute, die die gleiche Bedeutung haben, aber unterschiedliche Attributnamen besitzen. *Homonyme* sind Attribute mit gleichem Attributnamen aber unterschiedlicher Bedeutung.


Bei der Suche nach Analogien werden alle Attribute einer Datenbank, die den gleichen Typ und verschiedene Längen haben, überprüft. Da diese Anzahl in großen Relationen sehr groß sein kann, müssen weitere Hinweise zur Einschränkung der Analogienmenge ausgewertet werden.

1. **Auswertung der Attributname**
   Gleiche Attributnamen oder gleiche Teilstrings in Attributnamen weisen auf Analogien hin.

2. **Auswertung der Entity/Relationshipnamen**
   Gleiche Teilstrings in Entity- oder Relationshipsnamen können ebenfalls ein Hinweis auf eine ähnliche Bedeutung von Knoten im Entwurf sein, sodaß zwischen diesen Knoten Analogien auftreten können.

3. **Auswertung der Instanzen**

4. **Auswertung der ermittelten Analogien**
   Wurden zwischen zwei Knoten bereits Analogien gefunden, so können zwischen diesen auch weitere Analogien auftreten, da es sein kann, daß größere gleiche Attributsequenzen in diesen beiden Knoten definiert sind.

5. **Auswertung der Semantik**
   Treten innerhalb von zwei Attributmengen an verschiedenen Knoten funktionale Abhängigkeiten auf, so sind diese ein zusätzlicher Hinweis auf Synonyme.


   Diese Heuristik wird jedoch nur betrachtet, wenn bereits weitere Hinweise auf diese Analogien vorliegen, sie ist nur in Ergänzung zu den anderen Heuristiken sinnvoll.

6. **Auswertung der Pfad im DB-Entwurf**

7. **WW mit interrel. Semantik**

   Auf diese Weise können Analogien gefunden werden. Die einzelnen Heuristikregeln werden gegeneinander gewichtet. Dabei wird wieder eine einfache Abschätzung über eine gewichtete Summe verwendet.

   Mit dieser Abschätzung werden nicht nur Synonyme als Analoga ermittelt, es werden auch Homonyme ausgeschlossen bzw. mit schwächerer Wichtung versehen, da sich die Identifizierung gleicher Attribute nicht nur auf die Attributnamen stützt.

   Inklusions- und Exklusionsabhängigkeiten sind nur über Attributen, die als Analoga ermittelt wurden, sinnvoll. Der Suchraum zur Ermittlung von Inklusions- und Exklusionsabhängigkeiten wird so eingeschränkt.
5.4 Heuristiken zur Suche nach Inklusions- und Exklusionsabhängigkeiten

Mit folgenden Ansätzen bekommt man durch Auswertung der Instanzen Kandidaten für Inklusions- und Exklusionsabhängigkeiten über den bereits ermittelten Analog $R.X \approx S.Y$.

- Aus den Instanzen einer Datenbank kann man Kandidaten für Inklusionsabhängigkeiten ableiten. Sind die Werte des einen Attributes $R.X$ vollständig in den Werten eines anderen Attributes $S.Y$ enthalten sind, so kann man hier die Inklusionsabhängigkeit ($R.X \subseteq S.Y$) vermuten.
- Sind die Werte von $S.Y$ Teilmenge der Werte von $R.X$, so kann man hier die Inklusionsabhängigkeit ($S.Y \subseteq R.X$) vermuten.
- Wenn in den Attributen $R.X$ und $S.Y$ gleiche Werte auftreten aber keine Teilmengenbeziehungen existieren, so kann man hier eine der Inklusionsabhängigkeiten ($R.X \subseteq S.Y$ oder $S.Y \subseteq R.X$) bestehen.
- Ein Kandidat für eine Exklusionsabhängigkeit ($R.X \not\subseteq S.Y$) wird abgeleitet, wenn in den Instanzen keine gemeinsamen Einträge in $R.X$ und $S.Y$ auftreten.

Wurde durch die Validierung eine Inklusionsabhängigkeit oder Exklusionsabhängigkeit gefunden, so sind zwischen den betreffenden Attributen die semantischen Beziehungen geklärt. Bei der Ableitung von negierten Inklusionsabhängigkeiten bzw. negierten Exklusionsabhängigkeiten muß folgende Nachfrage zur Präzisierung der semantischen Constraints erfolgen.

- Hat man bereits eine negierte Inklusionsabhängigkeit ($R.X \not\subseteq S.Y$) abgeleitet, so muß auch untersucht werden, ob zwischen den Knoten sogar eine Exklusionsabhängigkeit besteht ($R.X \not\subseteq S.Y$).
- Ist eine negierte Exklusionsabhängigkeit ($R.X \not\subseteq S.Y$) bekannt, so muß auch eine Inklusionsabhängigkeit zwischen den Knoten untersucht werden ($R.X \subseteq S.Y$ oder $S.Y \subseteq R.X$).

Mit den Methoden zur Suche nach Inklusionsabhängigkeiten werden nicht nur schlüsselbasierte Inklusionsabhängigkeiten gefunden, sondern es können auch semantische Inklusionsabhängigkeiten ermittelt werden. Deshalb ist in den Heuristiken keine Auswertung der Schlüsseleigenschaften vorhanden, diese würde die Menge der ermittelten Inklusionsabhängigkeiten einschränken.

Es kann trotz des Einsatzes verschiedener Heuristiken passieren, daß Relationen auftreten, für die keine Verbindung zu anderen Knoten gefunden werden konnte, da hier evtl. keine Analogien mit den Heuristiken nicht gefunden werden konnten, weil die Verbindung über Anwendungsprogramme hergestellt wurde oder der Entwurf noch unvollständig ist. In diesem Fall muß eine direkte Befragung des Nutzers im Anschluß an die interaktive Semantikvalidierung erfolgen.

6 Ableitbarkeitsprüfung für die Kandidaten

Ermittelte Kandidaten für Constraints können falsch sein. Teilweise ist das vor der Validierung zu erkennen, wenn:

1. ermittelte Kandidaten bereits im Bsp nicht erfüllt sind oder
2. vermutete Constraints in Konflikt zur Menge der semantischen Constraints stehen oder
3. Kandidaten bereits aus der Menge der semantischen Constraints ableitbar und damit redundant sind.

Ist einer dieser Punkte erfüllt, so werden die betreffenden Kandidaten aus der Menge der zu validierenden Kandidaten gelöscht.

7 Validierung semantischer Constraints anhand von Beispielen

Durch die gezeigten Verfahren wurde eine Menge von unbekannten semantischen Constraints ermittelt, die validiert werden muß, da sie über Heuristiken abgeleitet wurde.

7.1 Probleme bei der Validierung von Constraints

Die Validierung semantischer Constraints kann nur im Dialog mit dem Nutzer erfolgen. Dabei treten zwei Probleme auf:

- Durch die große Anzahl der möglichen semantischen Constraints können sehr viele Kandidaten für Constraints existieren, die validiert werden müssen.
- Manche semantische Constraints, insbesondere Constraints, die über vielen Attributen definiert sind, sind für viele Nutzer schwer zu verstehen und dadurch schwer zu bestätigen oder abzulehnen.

Man benötigt also eine informale und effiziente Methode zur Validierung der Menge der ermittelten Constraintkandidaten.


7.2 Effiziente und informale Validierung anhand von Beispielen


7.2.1 Bestimmung der Reihenfolge der Validierung

Semantische Constraints sind aus anderen Constraints ableitbar, deshalb hat die Reihenfolge der Erfragung von Constraints Auswirkungen auf die Anzahl der notwendigen Dialogschritte. Es sollen zuerst die Constraints erfragt werden, aus denen die meisten anderen, noch unbekannten Constraints ableitbar sind. Die Anzahl der ableitbaren, noch unbekannten Constraints soll als der Informationsgehalt der Constraints bezeichnet werden. Dieser läßt sich folgendermaßen berechnen:

\[ \text{Informationsgehalt}(X) := \frac{\text{Anz}(X) + \text{Anz}(\neg X)}{2} \]

wobei \( \text{Anz}(X) \) die Anzahl der aus dem Constraint \( X \) ableitbaren unbekannten Constraints ist.

Bsp: Sind in der Relation \( R = (A, B, C) \) noch keine Constraints bekannt, so lassen sich aus dem Schlüssel \( A \) drei unbekannte Constraints ableiten (\( A \) - Schlüssel, \( AB \) - Schlüssel, \( AC \) - Schlüssel). Aus dem negierten Schlüssel \( A \) läßt sich nur ein unbekanntes Constraint ableiten (\( A \) - negierter Schlüssel). Der Informationsgehalt des Knotens \( A \) ist also 2.

Hat man Informationen darüber, wie ein Constraint wahrscheinlich validiert wird, so kann man den Informationsgehalt genauer abschätzen.

\[ \text{Informationsgehalt}(X) := (\frac{P(X)}{100} \times \text{Anz}(X)) + (\frac{P(\neg X)}{100} \times \text{Anz}(\neg X)) \]

wobei \( \text{Anz}(X) \) die Anzahl der aus dem Constraint \( X \) ableitbaren unbekannten Constraints ist und \( P \) die in Kapitel 5 angegebenen Heuristikabschätzungen für Constraints.

Beispiel: Nehmen wir für die obige Relation an, daß die Wahrscheinlichkeit, dass \( A \) Schlüssel ist mit 80 A kein Schlüssel ist, beträgt dann 20 Informationsgehalt des Knotens \( A \) dann 2.6.


7.2.2 Informale Validierung


Erfragung funktionaler Abhängigkeiten anhand von Beispielen

Funktionale Abhängigkeiten können in einfacher Weise anhand von Beispielen erfragt werden. Dabei möchte man wissen, welche funktionalen Abhängigkeiten in jeder konsistenter Relation gelten müssen. Die Darstellung dieser funktionalen Abhängigkeiten in einem Beispiel ist nicht möglich, man kann jedoch in einfacher Weise die Negation einer funktionalen Abhängigkeit (in Definition ?? als afunktionale Abhängigkeit eingeführt) in einer Beispielrelation darstellen und erfragen. Für eine Relation Personen sieht die Erfragung der funktionalen Abhängigkeit (Name Vorname → Wohnort) am Beispiel so aus:
Können in der Relation — Personen — zwei Einträge auftreten, die in — Nachname, Vorname — den gleichen Wert und in — Wohnort — verschiedene Werte haben (j/n) ?

Bei diesem Beispiel lassen sich folgende Informationen ableiten:

bei Annahme: Nachname Vorname −→ Wohnort
bei Ablehnung: Nachname Vorname −→ Wohnort

Man kann also Informationen über funktionale und nicht geltende funktionale Abhängigkeiten in einfacher Weise erfragen. Der Nutzer muß nur über die Richtigkeit von Beispielen entscheiden, formale Constraints können aus den Antworten abgeleitet werden.

Erfragung von Inklusionsabhängigkeiten

Die Methode zur Validierung von Inklusionsabhängigkeiten ist analog zur Erfragung funktionaler Abhängigkeiten. Eine negierte Inklusionsabhängigkeit wird in einem Beispiel dargestellt und die Richtigkeit dieses Beispiels erfragt.


Der Nutzer wird nach der Richtigkeit dieses Beispiels gefragt. Da negierte Inklusionsabhängigkeiten nur über vollständigen Relationen dargestellt werden können, müßte in der Relation für $S$ eine vollständige Instanz aufgeführt sein. Um das zu umgehen, wird die folgende Frage formuliert, die eine negierte Inklusionsabhängigkeit verbal erfragt:

Ist das generierte Beispiel richtig, dann gilt eine negierte Inklusionsabhängigkeit (für das Beispiel Studenten.Name $\subseteq$ Personen.Nachname), im anderen Fall eine Inklusionsabhängigkeit (für das Beispiel Studenten.Name $\subseteq$ Personen.Nachname).

Erfragung von Exklusionsabhängigkeiten

Auch die Validierung von Exklusionsabhängigkeiten ist anhand von Beispielen möglich.


Das Vorgehen soll wieder an einem Beispiel erläutert werden.
Auf diese Weise wird wieder eine negierte Exklusionsabhängigkeit dargestellt, deren Richtigkeit folgendermaßen erfragt wird:
Wenn das Beispiel auftreten kann, so gilt eine negierte Exklusionsabhängigkeit (Studenten.Name || Personen.Nachname), im anderen Fall kann eine Exklusionsabhängigkeit (Studenten.Name || Personen.Nachname) abgeleitet werden.
Auch Exklusionsabhängigkeit lassen sich also in einfacher Weise erfragen, ohne daß der Nutzer die formale Definition dieser Constraints kennen muß.

References

[CaV 83] M. A. Casanova, V.M.P. Vidal, Towards a Sound View Integration Methodology, 2nd ACM SIGACT-SIGMOD, PODS, 1983, Atlanta
[KiM 92] Jyrki Kivinen, Heikki Mannila, Approximate Dependency Inference from Relations, in J. Biskup, R. Hull (Eds), Database Theory, ICDT ’92, Lecture Notes in Computer Science 646, Springer Verlag Berlin 92


[TAA 94] Bernhard Thalheim, Meike Albrecht, Margita Altus, Edith Buchholz, Antje Düsterhöft, Klaus-Dieter Schewe, Die Intelligente Tool Box zum Datenbankentwurf RAD, GI-Tagung, Kassel, Datenbankrundbrief, Mai 1994

[Tha 91a] Bernhard Thalheim, Dependencies on Relational Databases, Teubner Texte zur Mathematik, Bd. 126, Stuttgart, Leipzig, 1991

[Tha 91b] Bernhard Thalheim, Using Semantics in Extended Entity-Relationship Models, Preprint CS-02-92, Universität Rostock

[Tha 91c] Bernhard Thalheim, Constraints in Extended Entity-Relationship Models, Preprint CS-03-92, Universität Rostock

[Tha 94] Bernhard Thalheim, Fundamentals of Entity-Relationship Modelling, in Vorbereitung


An Informal and Efficient Approach for Obtaining Semantic Constraints using Sample Data and Natural Language Processing *

Meike Albrecht, Edith Buchholz, Antje Düsterhöft, Bernhard Thalheim

University of Rostock, Department of Computer Science
e-mail: {meike/buch/duest}@informatik.uni-rostock.de

Cottbus Technical University, Department of Computer Science
e-mail: thalheim@informatik.tu-cottbus.de

Abstract

The main objective of database modelling is the design of a database that is correct and can be processed efficiently by a database management system. The efficiency and correctness of a database depends among other things on knowledge about database semantics because semantic constraints are the prerequisite for normalisation and restructuring operations. Acquisition of semantic constraints remains one of the bottlenecks in database design because for most database designers formal definition of semantic constraints is a very difficult task. Within the framework of the project RADD (Rapid Application and Database Development) experience was gathered with the informal modelling of database structures.

We show in this paper an approach for acquisition of semantic constraints which is informal, easy understandable and efficient. This method uses natural language input, sample data and a discussion of sample relations to find out semantic constraints of a database.

1 Introduction

The performance of a database (especially efficiency and consistency) heavily depends on design decisions. In order to achieve an effective behaviour of the database, database designers are requested to find the best structure and the simplest basic database operations.

The efficiency and correctness of a database depends among other things on knowledge about database semantics because semantic constraints are the prerequisite for normalisation and restructuring operations.

1.1 Problems in the Acquisition of Semantic Constraints

Formal determining of semantic constraints is one of the most difficult tasks in database design. We concentrate in our tool on the most commonly used constraints (functional dependencies, keys, inclusion dependencies, exclusion dependencies, cardinality constraints).

The problems in the acquisition of these constraints are caused by the following reasons:

*This work is supported by DFG Th465/2.
• For the correct utilisation of the semantic information a deep understanding of logic’s is required. This task not only demands high abstraction abilities but it is also very complex in general, especially for relations with many attributes and for large databases.
• Nearly all design methods need complete semantic information to get the right results (for instance in the normalisation of relational databases). Incomplete information causes wrong or inefficient databases. But even if a designer is able to determine formal semantic constraints he may forget some constraints because they are either too obvious or too difficult.
• Often designers misunderstand semantic constraints and therefore, they interpret them wrong. Especially, if different designers work together it could happen that they interpret the same semantic constraints in different ways. Therefore, the meaning of the semantic constraints must be determined.

Therefore, a support for the acquisition of semantic constraints is desirable which suggests semantic constraints to the database designer. Thereby further problems occur:

• Some constraints are complicate to decide for a database designer, especially if they are defined over many attributes.
• Not all semantic constraints can be visualised in graphic notation.
• The number of semantic constraints which has to be checked is exponential in the number of attributes.

In this paper we want to show an approach which overcomes these problems.

1.2 Overview of our Approach

Because of these known problems an informal approach must be chosen that enables designers to determine semantic constraints correctly. Further, the search for semantic constraints must be efficient to minimise the number of possible constraints that has to be checked. Therefore, heuristic rules can be taken. Also, it is useful to combine the structural design and the determination of semantic constraints. An approach using these characteristics is shown in this paper.

Main parts of the approach that we want to represent in this paper are also implemented in a database design tool. We use an extended entity-relationship model for this task (detailed represented in [BOT90]). In the tool we concentrate on the acquisition of the following semantic constraints: keys, functional dependencies, inclusion dependencies as they are defined in [MaR92] and exclusion dependencies. We also want to acquire cardinality constraints, these are defined in the following way: If $R_1$ is a relationship type, $R_2$ an entity or relationship type then $\text{card}(R_1,R_2) = (\text{min},\text{max})$ specifies that in each consistent state of the database an item from $R_2$ appears in $R_1$ at least $\text{min}$ and at most $\text{max}$ times.

1.3 Objectives and Related Research

Starting from [Che83] several work (e.g. [FPR85] or [TjB93]) exist for mapping the natural language utterances onto entity relationship models. The project ANNAPURA ([Eic84]) implements a dialogue tool for designing a binary relational model and for supporting the translation into a logical schema. Natural language sentences are the input to the system. [CGS83] shows an approach for the acquisition of semantics and behaviour of the database using natural language.
[Ort93] studies the technical language (German) which is used in working situations in order to develop a standardised enterprise language for the software development. The idea of our work is the development of a complex dialogue tool for getting information about structure, semantic and behaviour of the prospective database from natural language utterances.

Informal acquisition of semantics is not often treated in the relevant literature. A method for the derivation of keys and functional dependencies from data is described by [CaS93] and [MaR92]. In this method a closed-world assumption is taken for granted, or the derived constraints are simply confirmed (pseudo natural language acquisition in [BGM85] and Armstrong relationships in [MaR92]). Simple heuristics for the search for keys [StG88] and foreign key [CBS94] are offered. The approach presented here is an expansion of them and offers heuristics for the search for further constraints. In addition, methods for the efficient acquisition of constraints are used and not valid constraints are also evaluated. For the validation of constraints a query by means of examples representing semantic constraints is used.

2 General Approach

In our approach we want to acquire the most common used semantic constraints in databases: keys, functional dependencies, inclusion dependencies, exclusion dependencies and cardinality constraints.

2.1 Treatment of Semantic Constraints

In database design we want to know which semantic constraints must be valid in every consistent state of the database. We are not interested in the semantic constraints which are valid in one relation or database. Therefore, in this paper we do not handle only the set of valid semantic constraints but also the set of not valid constraints. Information about not valid constraints often can be found easier than information about valid semantic constraints. They restrict search space, therefore it is useful to collect and exploit them, too. Both information are derivable from the input information of the approach. Therefore, these two sets are collected:

- valid semantic constraints
- not valid semantic constraints.

The collection and storing of not valid constraints as well as valid constraints enables the derivation of unknown semantic constraints. All unknown semantic constraints must be checked. We will show methods which determine which of the still unknown semantic constraints seem probable be fulfilled and which of them seem to be not valid. Therefore, heuristic rules are used which exploit vague information about semantic constraints in the database. This information enters in the candidate sets for semantic constraints:

- candidates for valid semantic constraints
- candidates for not valid semantic constraints.

In the tool candidates are validated in a dialogue with the designer. In that way from the set of candidates valid or not valid semantic constraints are derived. We can determine if all semantic constraints are known, so we can ensure that no semantic constraints are forgotten.
2.2 Overview on the Approach

Databases can be designed from a natural-language description of the application. Information about structure of the database and candidates for valid and not valid semantic constraints are derivable from natural sentences (section 3). But assertions in natural language are often vague and ambiguous. Therefore, also derived databases and semantic constraints are vague. In section 4 we declare how derived structural information can be validated.

To validate structure and semantics we collect further information about the database. Therefore, the designer is asked to enter sample data. From these data some not valid semantic constraints are derivable (section 5).

In the tool also an explicit determining of semantic constraints is supported (section 6). From descriptions in natural language, sample data and explicit entered semantic constraints not all semantic constraints are derivable therefore it is necessary to search for further candidates for semantic constraints. In section 7 different heuristics are shown for this task.

All candidates for semantic constraints which are derived from the natural-language input and the heuristic rules are vague. That is the reason why they must be validated. An informal approach based on an example discussion is used for this task (section 8). Further, we show in this section how inquiring of candidates can be made efficient.

Section 9 gives a conclusion.
This approach is even useful for unskilled database designers because the design of databases in natural language is a very informal method. Validation of structure is supported by an comfortable graphical editor. Also validation of constraints with examples is easy-understandable.

3 The Linguistic Approach

3.1 Aims of the Linguistic Approach

The aim of the natural language component is to offer the designer the possibility to present his informal knowledge in natural language. Basic prerequisite for the adequate modelling of a problem concerning the database design is the presentation of it. Using a natural language is for most users the best way of presenting the contents and structure of a database - in contrast to using formal models such as diagrams. We believe that a problem expressed in natural language will offer more information on its structure, its semantics as well as its behaviour than any other mode of expression.

In database design a rough structure of the application is obtained first. In doing so, a special
language is used in which words have no longer a wide scope of meaning. In a special context
ambiguities are mostly soluble. In this respect the German language because of its variable
sentences structure offers even more means of expression than the English language.
Recently computational linguistics has produced a series of outstanding contributions to auto-
matic processing of natural language (e.g. in the field of semantics [Pin93]). In order to use
these new findings in design tools further linguistic analysis and restrictions to special domains
of language are needed.
It has become evident that a ‘general-purpose-system’ for complex applications is not feasible.
Complex applications are marked by a variety of semantic constructs as well as by a frequent use
of formal parts. Database design is an example of such a complex application. On the one hand
it requires a formalised description, on the other hand it requires a language which has certain
standardised forms. Thus database design is well suited to use approaches of computational
linguistics and to demonstrate how a natural language interface can help to make the design
more effective.

3.2 General Analysis of Natural Language Input

The natural language interface will be developed as a dialogue component working interactively
together with the other components of the RADD system. It consists of a dialogue tool, a syntax
tool, a semantics tool and a pragmatics tool. For a more detailed description of these tools see
[BuD94a] and [BuD94b].

For the acquisition of designer knowledge we decided to develop a question-answer system (dia-
logue tool). This tool reacts appropriately to every input, and it asks for additional input if this
is needed for the acquisition of database design information. The questions are frames which are
constantly being updated as the dialogue proceeds. The designer inputs his answers in natural
language sentences or parts of sentences.

The designer input into the dialogue tool is first submitted to a syntax analyser. In order to
check the syntax an ID/LP parser was implemented. The parser works on the basis of a lexicon
of German words and phrases and was built for a restricted area of German sentences (descriptions
and definitions).

The semantic tool is used to obtain an interpretation of the semantics of the designer input. The
interpretation consists of two stages: firstly the semantics of the individual words and secondly
the semantics of complete utterances and sentences will be defined. For the identification of the
meaning of sentences we have used the model of semantic roles: the units of an utterance or a
sentences are seen to fulfil certain functional roles.

The aim of the pragmatic interpretation is the mapping of the natural input, i.e. the results
of the syntactic and semantic analyses, onto the entity-relationship model. This transformation
is handled like a compiler process using an attribute grammar. The results of this process are
a skeleton design, a list of possible semantic constraints and information on the behaviour of
the prospective database. Interface procedures are used to transform these results into the Data
Dictionary structure.

When an NLI has been designed for a specific domain, it can easily be adopted for closely related
domains - in our case all lending institutions and domains. Some new entities (nouns) and
relationships (verbs mainly) will have to be entered into a specific domain lexicon. Even loosely
related processes, e.g. all selling and vending processes, or the processes of storing objects, can
make use of the same interface after establishing at the outset that the entities which will form
the stock of the database will normally not be returned. A specific domain library will have to be updated and specified prior to the design dialogue.

3.3 The Linguistic and Pragmatic Basis for the Extraction of Semantic Information

3.3.1 The Transformation Process

The acquisition of database semantics, e.g. the acquisition of keys or cardinalities from natural language sentences is part of the process of the pragmatic interpretation of the natural language input. Special functions within the attribute grammar are used for the transformation of the natural language semantics into a database semantics. In a natural language description the designer uses semantic nuances intuitively. New findings in the field of computational linguistics are integrated in order to capture these nuances (e.g. [Pin93]).

3.3.2 Linguistic principles

For the acquisition of the database semantics we use linguistic methods which define rules for the assignment of meaning.

Principle of compositionality. [GaM89, 280] describe Frege’s principle: 'The meaning of the whole sentence is a function of the meaning of the parts.' In other words, the meaning of a sentence is composed of the meanings of its parts. The meanings of the parts are then described as the meaning of the subphrases and so on. In the end we have to examine the meaning of the words. For this reason the basis of the semantics of a sentence is the semantics of words and phrases defined in the lexicon.

Let us consider this principle for our database design application. First we have to define which parts of a sentence will play an important part during the acquisition of database semantics and which words form the basis of these parts (cf. also [Pin86]).

Meaning as a reference. If we accept the principle of compositionality then we have to answer the question which relation exists between the parts or words of a sentence. In order to handle this problem we have developed a classification of words. This classification is integrated into the knowledge base (e.g. the verb 'eat' belongs to the verb class of activity verbs and correlates with the noun class 'food'). The classifications and relations are only partially domain independent. Special knowledge or information about the domain is necessary for all applications. In this way the description of meaning can also be seen as a 'feature instantiation'.

3.3.3 Pragmatic approach

Certain parts of the technical language are characterised by pragmatic properties which can be found in all applications. These pragmatic properties are discovered and confirmed by statistic observations. So the word 'several' implies a set greater than two. A designer using the word 'several' rarely wants it to refer to a set of two elements.

The acquisition of the semantic information needed for the design of a database is based on a set of heuristic assumptions which are linguistically motivated. These assumptions some of which are illustrated in the following section are integrated into the knowledge base.
3.4 Selected Examples of the Acquisition of Semantics

**Key candidates.** Assumptions for the acquisition of key candidates are:

1. the first named attribute of an entity is defined as a key candidate
2. attributes which have the character of numbers are defined as key candidates
3. identifying nouns which are marked in the lexicon

The key candidates thus extracted are described as `keycand(entity- or relationship-name, key attribute)`. For readability reasons we show the examples in German (the implementation language) and in English. The results of the analysis are given in English only.

The first assumption is based on the observation that a designer names the most important attribute first.

Example: ‘Ein Gewässer hat einen Namen und einen Standort.’
(Waters have a name and a site.)
Key: `keycand(water,name)`

Assumption 2 goes along with the fact that the named attribute reflects a number. The German nouns ‘Hausnummer’ (house number), ‘Kartennummer’ (card number), ‘Datum’ (date), ‘Zeit’ (time) have the character of numbers. The same applies to nouns which have these nouns as substrings. Nouns which have a character of numerals only within a certain domain can be explained explicitly in the knowledge base (e.g. ‘ISBN’).

Example: ‘Eine Messung ist gekennzeichnet durch ein Datum, eine Uhrzeit und eine Konzentration.’
(A measurement is characterised by a date, a time and a concentration.)
Key: `keycand(measurement,date)`
       `keycand(measurement,time)`

The third assumption is based on the character of certain nouns which makes them identifiers. This group comprises words such as ‘Kennzeichnung’ (label), ‘Bezeichnung’ (term), ‘Benennung’ (naming) or ‘Identifikator’ (identifier).

Example: ‘Eine Ionenkonzentration hat eine Bezeichnung und einen Typ.’
(A concentration of ions has a term and a type.)
Key: `keycand(concentration of ions, term)`

For the identification of words which reflect key assumptions synonym and frequency dictionaries are used.

**Cardinalities.** Linguistic studies of special determiners (e.g. ‘ein’ (a), ‘der’ (the), ‘jeder’ (every), or ‘mehrere’ (several) for the acquisition of cardinalities in natural language utterances. A designer using these determiners consciously or subconsciously defines certain cardinalities of an application and has his personal interpretations of the determiners. We try to extract the most plausible interpretations. The order of interpretations is described by the numbering contained in the facts. The following examples will illustrate the view at some determiners. The resulting cardinalities are defined as `cardcand(number, relationship-name, entity-name, mincard, maxcard)`. The German word ‘ein’ (a) has the following meanings:
• mindestens ein (at least one) - 1:n - or
• genau ein (exactly one) - 1:1.

Any other variants of the interpretation of ‘ein’(a) are not relevant.
If a designer uses the word ‘ein’(a) explicitly we assume that is more likely that he wants to
describe a 1:1 cardinality.

Example: ‘Ein Gewässer weist eine Ionenkonzentration auf.’
(A water has a concentration of ions.)
Cardinality constraint:
- cardcand(1, has, water, 1, 1)
- cardcand(2, has, water, 1, n)
- cardcand(1, has, water, 1, 1)
- cardcand(2, has, water, 1, n)

The zero article (non-occupance of an article) mainly appears in connection with plural words.
These words suggest the repeated occupance of objects or executers. We assume that the devel-
oper when using zero articles does not want to describe exact and concrete objects or executers
but prefers a 1:n cardinality.

Example: ‘Gewässer werden durch Faunen besiedelt.’
(Waters are populated by faunae.)
Cardinality constraint:
- cardcand(1, populate, waters, 1, n)
- cardcand(2, populate, waters, 1, 1)
- cardcand(1, populate, faunae, 1, n)
- cardcand(2, populate, faunae, 1, 1)

The German word ‘mehrere’ (several) reflects a 1:n relation. An interpretation of ’exactly one’
is improbable.

Example: ‘Für ein Gewässer werden mehrere Eintrittspunkte deﬁniert.’
(Several points of entry are deﬁned for waters.)
Cardinality constraint:
- cardcand(1, deﬁne, water, 1, 1)
- cardcand(2, deﬁne, waters, 1, n)
- cardcand(1, deﬁne, points of entry, 1, n)

Determiners have to be analysed for potential cardinalities. Then they are labelled before integra-
tion into the knowledge base. In many languages including German determiners are a manageable
number of words. Labels on the cardinality potential describe internal characteristics of deter-
miners and are, therefore, domain independent.

Inclusion and Exclusion Dependencies. Inclusion or exclusion dependencies are assumed
in natural language sentences when entities are enumerated for which a hypernym exists. Enum-
eration exists when nouns are connected by connectors such as ‘und’ (and), ‘oder’ (or), ‘sowohl
als auch’ (as well as) or by a comma.
Detected inclusion and exclusion dependencies are presented in the form inclcand(entity/ rela-
tionship, entity included/ relationship) and exclcand(list of entities/ relationship).
Example: ‘Seen, Flüsse und Teiche sind Gewässer.’
(Lakes, rivers and ponds are waters.)
exclusion dependency: exclcand([lake, river, pond])
inclusion dependency: inclcand(water, lake)
inclcand(water, river)
inclcand(water, pond)

**Synonyms**  For a further definition of inclusion and exclusion dependencies by examples the tool searches the lexicon for synonyms of all identifiers appearing in the natural language input (entity-names, relationship-names, attribute-names). If these synonyms appear in the skeleton design as well, they are made available in the form of a fact synonym(identifier, list-of-synonyms).

### 3.5 Example of a dialogue

The following example of a dialogue shows the basic course of the dialogue. In this dialogue the developer makes so-called content utterances, e.g. ‘Several points of entry are defined for a body of water.’ when given so-called content prompts or questions. Such prompts are hidden in questions such as ‘Are there any more details?’. The system can put additional questions for the clarification of linguistic or pragmatic ambiguities or defects. A linguistic clarification is aimed at by the following question: How is the act of ’defining’ done? A pragmatic clarification question by the system could be: ‘How is point of entry characterised?’. This questions aim at a completion of the design.

When looking at the general acquisition of knowledge through natural language input, this seems to be best achieved by putting such pragmatic questions to the developer that will yield an exact definition of the cardinalities. This will, however, make the dialogue very long and tedious. Furthermore, developers are mostly not able to answers these questions correctly. Therefore we will limit the elucidation process to the acquisition of cardinality candidates only. They are verified by means of examples at a later point (see chapter 4).
Ein Gewässer weist Ionen auf. 
(A water has a concentration of ions.)

Wodurch ist Gewässer gekennzeichnet?
(How is water characterised?)
Ein Gewässer hat einen Namen und einen Standort. 
(A water has a name and a site.)

Wodurch ist Ion gekennzeichnet?
(How is concentration of ions characterised?)
Durch den Typ, die Art und eine Bezeichnung. 
(The type, the kind and the term.)

Gibt es weitere Details?
(Are there any more details?)

- Seen, Flüsse und Teiche sind Gewässer. 
(Lakes, rivers and ponds are waters.)

Muß man See außer mit -Name- und -Standort- noch weiter kennzeichnen? Wenn ja, wie?
(Does lake have to characterised any more? If yes, how?)
Ein See hat eine Größe und eine Tiefe. 
(A lake has a size and a depth.)

Muß man Fluß außer mit -Name- und -Standort- noch weiter kennzeichnen? Wenn ja, wie?
(Does river have to characterised any more? If yes, how?)
Ein Fluß hat eine Länge. 
(A river has a length.)

Gibt es weitere Details?
(Are there any more details?)

Für ein Gewässer werden mehrere Eintrittspunkte definiert. 
(Several points of entry are defined for waters.)

Auf welche Art und Weise wird die Aktion - definieren - durchgeführt?
(In what way is the action -define- carried out?)
Mit einer Datenblattnummer. 
(With a number of a data paper.)

Wodurch ist Eintrittspunkt gekennzeichnet?
(How is point of entry characterised?)
Ort, Tiefe, Fließvектор. 
(Place, depth, vector of flowing.)

Gibt es weitere Details?
(Are there any more details?)

An jedem Eintrittspunkt werden Ionen gemessen. 
(Ions are measured on every point of entry.)

Auf welche Art und Weise wird die Aktion - messen - durchgeführt?
(In what way is the action -measure- carried out?)
Es wird eine Konzentration an einem bestimmten Tag und in einer bestimmten Tiefe gemessen. 
(A concentration is measured on a particular day and a particular depth.)

Gibt es weitere Details?
(Are there any more details?)

Gewässer werden durch Faunen besiedelt. 
(Waters are populated by faunae.)

The following suggestion of a skeleton design is derived from the example of a dialogue:

The following key candidates are derived from the example:

keycand(water,name).
keycand(ion,term).
keycand(lake,name).
keycand(river,name).
keycand(pond,name).
keycand(lake,size).
keycand(river,length).
keycand(define,number of data paper).
keycand(point of entry,depth).
keycand(measure,day).
keycand(measure,depth).
The following candidates of cardinalities are acquired from the example:

```
cardcand(1, has, gewässer, 1, 1).
cardcand(2, has, gewässer, 1, n).
cardcand(1, has, ion, 1, 1).
cardcand(2, has, ion, 1, 1).
cardcand(1, define, water, 1, 1).
cardcand(2, define, water, 1, n).
cardcand(1, define, point of entry, 1, n).
cardcand(1, measure, point of entry, 1, 1).
cardcand(2, measure, point of entry, 1, n).
cardcand(1, measure, ion, 1, n).
cardcand(2, measure, ion, 1, 1).
cardcand(1, populate, water, 1, n).
cardcand(2, populate, water, 1, 1).
cardcand(1, populate, faunae, 1, n).
cardcand(2, populate, faunae, 1, 1).
```

The following inclusion and exclusion dependencies are extracted from the example as well as the following synonyms using the lexicon:

```
exklcand([lake, river, pond]).
inklcand(water, lake).
inklcand(water, river).
inklcand(water, pond).
synonym(name, [term]).
synonym(size, [place]).
```

All derived information are vague therefore, they must be checked in a dialogue with the designer.

### 4 Validation of Structure

From the natural language description a skeleton of the database was derived. This skeleton can only be a first proposal for the database structure. Derived database structure must be confirmed or corrected and extended by the designer, therefore a graphical representation in entity-relationship diagrams which was already shown in the last section for the example is useful. For this task a graphical editor was developed [TAA94, Alt94]. The editor is not only a tool that supports development of entity-relationship models but it also declares design decisions and suggests further design steps. In that way it comfortably supports changes and extensions in entity-relationship diagrams. The derived database structure also must be completed by attribute types and lengths. These
are technical details therefore asking for them already in the natural language design dialogue is not useful.

Incomplete or wrong designed parts of the database can be found if the database designer enters real world data. That is why we support input of sample data in the tool.

5 Sample Relation Exploitation

Sample data entered by the designer are very useful to search for semantic constraints because background knowledge of the designer is implicitly contained in the data.

It can be derived from the data that the following semantic constraints cannot be fulfilled because they are violated by the sample data:

- **not valid keys**
  If in the sample relations there are two tuples which have the same values in some attributes then these attributes cannot be keys because they do not identify all tuples of the sample relation.

- **not valid functional dependencies**
  A not valid functional dependency \( X \rightarrow Y \) exists if there are two tuples \( t, t' \) in the sample relation and \( t[X] = t'[X] \) and \( t[Y] \neq t'[Y] \). If we find those tuples then we can conclude that no functional dependency between \( X \) and \( Y \) exists.

- **not valid exclusion dependencies**
  If there are two attributes or attribute sequences which contain the same values in the sample databases then an exclusion dependency between these attribute sequences is not valid.

- **not valid cardinality constraints**
  From sample data it is derivable that cardinalities \( \text{cand}(R_1,R_2)=(\_1,\_1) \) are not valid if in \( R_1 \) the same value in the foreign key from \( R_2 \) occurs twice or more times.

Sample data entered by the designer normally contain only a few tuples. Therefore, only some not valid semantic constraints are derivable.

We try to get more information about the database from the designer. Therefore, he/she is asked whether he/she can determine how many tuples will occur in every relation of the database and how many distinct values every attribute will have. From this information further not valid semantic constraints are derivable:

- **not valid keys**
  If there exist attributes or attribute sets which have less different values then the number of tuples in the relation then these attributes cannot identify all tuples and therefore they are no keys.

  Example: In a relation \( \text{Students}=(\text{Stud}_\text{nr},\text{Course},\text{Department}) \) shall be 5000 tuples. In such a relation there are 5000 different values in \( \text{Stud}_\text{nr} \), 20 different values in \( \text{Department} \) because the university has only 20 different departments and 12 different values in \( \text{Course} \). Then we know that \( \text{Course} \) and \( \text{Department} \) and \( \text{Course},\text{Department} \) cannot be key.

- **not valid functional dependencies**
  A not valid functional dependency exists between two attributes \( (X \text{ and } Y) \) if there is an attribute or attribute set \( (X) \) which has less different values than another attribute \( (Y) \).

  For the example: \( \text{Course},\text{Department} \rightarrow\rightarrow \text{Stud}_\text{nr} \) and \( \text{Course} \rightarrow\rightarrow \text{Department} \) are derivable from the distinct values.
• **not valid inclusion dependencies**
  If there is an attribute which has more different values than another attribute then an inclusion dependency between these attributes cannot be valid. This information is only sensible if the attributes coincide in attribute types and lengths.

• **not valid cardinality constraints**
  Also from the number of tuples of the relations not valid cardinalities can be derived. If $R_1$ contains more tuples than $R_2$ then we can derive that $\text{cand}(R_1,R_2)=(-1,1)$ is not valid.

The information about the number of tuples and the number of distinct values is not necessary in the tool but it is very useful for derivation of not valid constraints and as we will show also in the heuristic rules.

6 **Explicit Determining of Semantic Constraints**

Our system supports informal acquisition of semantic constraints. Sometimes, designers can determine some constraints, for instance a key of a relation. In that case a long search for these semantic constraints is not sensible, therefore in the tool it is also possible to enter formal semantic constraints. It is checked whether the semantic constraints are not in contradiction to the already known semantic constraints and the sample relation.

By that, it is possible to support users with different abilities. A designer can determine all formal semantic constraints which he/she knows and in addition further constraints that he/she forgot are discussed in the tool.

7 **Heuristics to Derive Candidates for Constraints**

Now, some valid and not valid semantic constraints are known from the exploitation of sample data, distinct values and the explicit determining of semantic constraints. Further, we have some candidates for semantic constraints derived from the natural language input. In general, these candidates are not complete. Therefore, in the tool a search for further candidates for semantic constraints takes place.

Heuristic rules are used to find further candidates. Structural information, already known semantic information, sample relations and sample transactions (if known) are utilised in the heuristic rules. These heuristics are shown in this section.

Some of the heuristic rules were used in the natural language interface, too. That is why both tools are part of a complete design system and it also shall be possible to use the tool to acquire semantic constraints without the NLI tool.

7.1 **Heuristics to Find Keys**

Candidates for keys can be derived from structural information of the database, sample data, already known semantic constraints and transactions. The following heuristic rules can be used to find keys:

**Exploitation of Structural Characteristics**

1. Often artificial keys are defined in relational databases. Most of them are numbers. If there is an attribute which is defined as an integer with a long length then this attribute can be such an artificial key.
2. Sometimes an attribute name indicate keys. If the substring -name-, -key-, -identifier-, -number-, -#- is part of the attribute name, it points to an artificial key. These two methods are also suggested in [StG 88] and in [ChV 92] for forms.

Indications in the Sample Relations

3. A running number in an example of the relation is a very strong indication of the existence of an artificial key. This method is suggested in [ChV 92] for forms, it can also be used for relations.

4. Also the distinct values (if known) can be used to derive candidates for keys. Attributes which have more distinct values than other attributes in a relation are more probably key or part of a key.

Semantic Constraints

5. Conclusions about keys are derivable from the set of already known valid and not valid functional dependencies.

   For each attribute $A$ the following dependencies point to the possibility that $A$ is a key attribute:
   
   - nontrivial valid functional dependency $X \rightarrow Y, A \subseteq X$
   - not valid functional dependency $Y \not\rightarrow A$

   These dependencies point to the possibility that the attribute $A$ is no key:

   - nontrivial valid functional dependency $Y \rightarrow A,$
   - not valid functional dependency $X \not\rightarrow Y, A \subseteq X$

   In this way, further candidates for keys are derivable from the already known semantic constraints.

6. If there are attributes in a relation which are already determined as keys of another relation then these attributes are checked for being key of this relation, too.

   The two heuristic rules which exploit semantic constraints are especially useful because the results of these heuristics improves during the acquisition of semantic constraints.

Transactions

7. Attributes which are rarely or seldom updated are more probably keys than attributes which are often changed.

8. Attributes which are used in update or delete operations to identify tuples can be keys of the relation.

These eight heuristic rules are utilised and weighted. Therefore a simple estimation is used:

$$\text{Plausibility}(A \text{ is part of a key}) := \sum_{i=1}^{8} (w_i r_i(A))$$

$r_i$ is the result of heuristic rule $i$ between 0..1 and $w_i$ are weights between 0..100,

$$w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7 + w_8 = 100.$$
This estimation cannot be correct because the heuristic rules are not independent. But for finding candidates for keys this estimation will do, because the established heuristic values get higher if more heuristic rules indicate a key.

The weights can be adapted to the database designer and the field of application in a simple way: in the beginning they are determined, rules which have been successful several times are weighted higher in the next search for key candidates.

Every user develops his special style in creating databases. For instance, if a designer uses artificial keys then we often can derive key information from the attribute types and names. With the adaptation of the heuristic rules onto the designer and the database we can find valid semantic constraints of the database more efficient.

With this estimation we determine the plausibility that every attribute is part of a key. The plausibility of being key for an attribute set with more than one attribute is estimated as the mean value of the probabilities of all attributes of the set:

\[
\text{Plausibility}(X = \langle X_1..X_n \rangle \text{ is key}) := \left( \frac{1}{n} \sum_{i=1}^{n} \text{Plausibility}(X_i \text{ is part of a key}) \right)
\]

Not valid keys are the negation of a valid key. Therefore, the same heuristic rules can be used for the estimation how probable a key is not valid:

\[
\text{Plausibility}(X \text{ is no key}) := 100 - \left( \frac{1}{n} \sum_{i=1}^{n} \text{Plausibility}(X_i \text{ is part of a key}) \right)
\]

In that way we can find candidates for valid and not valid keys and how probably they seem to be fulfilled.

### 7.2 Heuristic Rules to Find Functional Dependencies

There are some heuristic rules that can derive candidates for functional dependencies from the database structure, already known semantic constraints, sample data, and transactions.

#### Derivation of Candidates for Functional Dependencies from Sample Relations:

1. From sample relations we cannot derive valid functional dependencies. But we can try to find candidates for valid functional dependencies in the sample data. For all attribute sequences with more than one attribute we find out how many tuples of the sample relations coincide in the attributes of the attribute sequence.

Example:

<table>
<thead>
<tr>
<th>Person_number</th>
<th>Surname</th>
<th>First_name</th>
<th>City</th>
<th>ZIP</th>
<th>Street</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>39283</td>
<td>Schmidt</td>
<td>Peter</td>
<td>Berlin</td>
<td>12555</td>
<td>Mollstrasse</td>
<td>20</td>
</tr>
<tr>
<td>48934</td>
<td>Meier</td>
<td>Paul</td>
<td>Berlin</td>
<td>12555</td>
<td>Mozartstr</td>
<td>13</td>
</tr>
<tr>
<td>23299</td>
<td>Schulz</td>
<td>Paul</td>
<td>Rostock</td>
<td>18055</td>
<td>Gerberbruch</td>
<td>12</td>
</tr>
<tr>
<td>12983</td>
<td>Lehmann</td>
<td>Karl</td>
<td>Rostock</td>
<td>18055</td>
<td>Grubenstr</td>
<td>6</td>
</tr>
</tbody>
</table>

In this small sample relation we have twice the same values in City and ZIP. The following candidates for functional dependencies are derived:

\[
\text{City } \xrightarrow{?} \text{ ZIP and ZIP } \xrightarrow{?} \text{ City}
\]
If there are many tuples having the same value in the attribute sequence, then a functional
dependency between the attributes is more probable than between attributes which always
have different values.

Especially, for larger sample relations we can derive detailed information about valid se-
monic constraints in that way.

**Exploitation of Structural Characteristics and Sample Data**

2. If there is an integer attribute \((X)\) in a relation having values which are always the sum or
product of two other integer attributes \((YZ)\), then the functional dependency \((YZ \rightarrow X)\)
can be expected. If the values of an attribute \((X)\) are always a constant percentage of
another integer attribute \((Y)\) then the functional dependency \((Y \rightarrow X)\) is expected.
Relations with such functional dependencies often occur in economic and technical appli-
cations.

**Semantic Constraints**

3. Further, vague conclusions about functional dependencies are derivable from the set of
already known valid and not valid functional dependencies.

For each attribute \(A\) the following dependencies point to the possibility that \(A\) is left side
of further functional dependencies:

- nontrivial valid functional dependency \(X \rightarrow Y, A \subseteq X\)
- not valid functional dependency \(Y \not\rightarrow A\)

These dependencies point to the possibility that the attribute \(A\) is on the right side of a
functional dependency:

- nontrivial valid functional dependency \(Y \rightarrow A,\)
- not valid functional dependency \(X \not\rightarrow Y, A \subseteq X\)

In this way, further candidates for keys are derivable from the already known semantic
constraints.

**Analysis of Transactions**

4. From update transactions we can derive that the attributes \((X)\) which identify the tuples
are the left side of a functional dependencies, the attributes \((Y)\) which are changed in the
operation are the right side. This dependency \((X \rightarrow Y)\) must has been fulfilled in the
instance on which the transaction has been run.

These heuristic rules are also weighted against each other in a simple way. All derived candidates
for functional dependencies has to be validated.

**7.3 Heuristic Rules to Search for Analogue Attributes**

Synonyms are attributes which have the same meaning but different names. Homonyms are
attributes having the same name and different meaning. Prerequisite for detecting inclusion
dependencies, exclusion dependencies and cardinality constraints is finding attributes which have
the same meaning. We call these attributes *analoga*.

Some analoga are derivable from the structure of entity-relationship diagrams. To enter data in
entity-relationship diagrams these diagrams must be translated into relational models. Thereby,
foreign keys are assigned to the relationships. These foreign keys of relationships and the belonging keys of entities are analoga. Cardinality constraints must be determined, here. But it is possible that users forget paths in the design. Therefore, we also search for further attributes in the database seeming to be analoga. If we find those, we can find forgotten paths in the diagrams and also inclusion and exclusion dependencies which are not represented in the structure.

In the analoga synonyms shall be contained, but homonyms not. That is why we concentrate in the determining of analoga not only on the attribute names but also on further characteristics of the database.

All attributes of a database having the same type and similar length are checked for being analoga. The following information are exploited:

**Derivation from Structural Characteristics**

1. Same or similar attribute names point to analoga.
2. Similar relation names are an indication for analoga between these relations.
3. Path information in the entity-relationship diagrams determine which nodes of a database belong closely together. Therefore we can find more probably analoga between nodes which are directly connected by a path than between attributes which do not have a closed connection.

**Exploitation of Sample Relations**

4. If there exist attributes in the database having the same values in the sample data then these attributes can be analoga.
5. If we know that attributes in the database have the same or similar number of distinct values then these attributes can be analoga.

**Transactions**

6. From sample transactions vague information about analoga are derivable. If in a transaction there are update or insert operations in two relations where the same value is updated or inserted in two or more attributes it is possible that these attributes are analoga.

**Semantic Constraints**

7. If we have found analoga (or inclusion/exclusion dependencies or foreign keys) between two relations then even more analoga between these relations can occur because there can exist attribute sequences which are contained in both relations.

The different heuristics are also weighted against each other. We get candidates for analogue attributes which are used to derive candidates for inclusion and exclusion dependencies and cardinality constraints. We show in the next sections how it can be done.

### 7.4 Derivation of Candidates for Inclusion and Exclusion Dependencies

The analoga are prerequisite to derive candidates for inclusion and exclusion dependencies. We exploit the *sample data* entered by the designer to follow candidates for inclusion and exclusion dependencies from analoga.

If there are no dependencies known between two relations \( R, S \) and we found analoga \( R.X \approx S.Y \), then the following dependencies are checked:
• If the values of \( R:X \) are completely contained in the values of \( S:Y \) then we obtain a candidate for an inclusion dependency \( (R:X \subseteq S:Y) \).

If the values of \( S:Y \) are subset of the values of \( R:X \) then we obtain a candidate for inclusion dependency \( (S:Y \subseteq R:X) \).

• If identical values in \( R:X \) and \( S:Y \) occur but the values are no subsets then it is also possible that there exists an inclusion dependency \( (R:X \subseteq S:Y) \) or \( (S:Y \subseteq R:X) \). These constraints have to be validated.

• Candidates for exclusion dependencies can be obtained if we found analoga \( (R:X \approx S:Y) \) which do not have the same values in sample relations \( (R:X \parallel S:Y) \).

In that way we can find candidates for inclusion and exclusion dependencies in the databases. These candidates for inclusion and exclusion dependencies do not only base on structural characteristics.

7.5 Cardinality Constraints

From the structure of entity-relationship models we can derive which cardinality constraints have to be determined. These are specified with path information.

Also, if we find attributes seeming to be analoga \( (R:X, S:Y) \) and \( Y \) is key in \( S \) then the cardinality \( \text{card}(R,S) \) must be determined. Especially, if there are parts of the designed entity-relationship diagram which have no connection to others then we try to find further cardinality constraints in that way.

The values of these candidates for cardinality constraints are derived from the natural language input or the sample data. The validation of these candidates is shown in section 8.

8 Informal and Efficient Validation

In the previous section we showed how candidates for semantic constraints are derivable. All these candidates are vague. Therefore, a validation by the database designer is necessary. We want to chose an approach which is understandable even for unskilled designers and therefore ensures correct results.

We use an informal approach for validation in the tool. We will show that this approach is efficient, too.

8.1 Informal Validation

For validation of candidates for semantic constraints we use an approach based on examples. Only one candidate for a semantic constraint is shown in a sample relation at a time and is inquired. We want to show this method for exclusion dependencies and cardinality constraints.
A candidate for an exclusion dependency or a not valid exclusion dependency can be inquired in the following way:

<table>
<thead>
<tr>
<th>Lake:</th>
<th>Name</th>
<th>Cite</th>
<th>Area</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pel</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>River:</th>
<th>Name</th>
<th>Cite</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is it possible that the same values are in — Lake.Name — and — River.Name — (yes/no/unknown)?

From the decision of the designer the following constraints can be obtained. If this sample relation can occur then we concluded that the exclusion dependency is not valid (Lake.Name || River.Name). Otherwise, if the example cannot occur, then this exclusion dependency (Lake.Name || River.Name) is valid.

We also want to demonstrate validation of candidates for semantic constraints for cardinality constraints:

<table>
<thead>
<tr>
<th>have:</th>
<th>Water.Name</th>
<th>Ion.Destignation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fe³⁺</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fe³⁺</td>
</tr>
</tbody>
</table>

Can same values in — Ion.Destignation — occur twice or more times (yes/no/unknown)?

If the example can occur we know that the cand(have,Ion,_,1) is not correct otherwise card-cand(have,Ion,_,1) is valid. The minimum value of the cardinality can be determined with an example discussion, too.

These two examples show that formal semantic constraints can be inquired with a sample data discussion. For the other semantic constraints (functional dependencies, keys, inclusion dependencies) acquisition can be achieved in the same way [Alb94b].

8.2 Sequence of Validation

It is not possible to check all unknown and inderivable constraints in an ordered way, because the number of independent functional dependencies of a relation is $O(2^n)$ where $n$ is the number of attributes of a relation. The number of unary inclusion and exclusion dependencies is $O(n^2)$ where $n$ is the number of attributes of the database as a whole. Therefore, also the number of derived candidates for semantic constraints can be very large.

From a set of semantic constraints further constraints are derivable. We use this property to make the dialogue for validation more effective.

Therefore, it is determined how many unknown semantic constraints are derivable from every candidate for a semantic constraint or how many information one semantic constraints will bring. The following formula can be used for that task:
\[ Information(X) := \left( \frac{P(X)}{100} \ast N(X) \right) + \left( \frac{P(\neg X)}{100} \ast N(\neg X) \right) \]

\( X \) is a still unknown semantic constraint,
\( N \) the number of unknown semantic constraints derivable from \( X \) and
\( P \) the probability that \( X \) is valid.

In the tool inquiring of semantic constraints with examples starts with those semantic constraints from which most other unknown constraints are derivable. In that way the number of dialogue steps for validation can be decreased.

### 8.3 Incomplete Acquisition of Semantic Constraints

For large relations or databases it is not possible to inquire all semantic constraints because the dialogue for validation of all unknown semantic constraints would be too long. But because of the many different heuristic rules we derive candidates for semantic constraints which seem to be probably valid. We can inquire these candidates in the tool. Often in that way all or the most important valid semantic constraints can be found because of the lot of heuristic rules.

### 9 Conclusions

As we showed our approach is informal. We only need a natural language description of the application and sample data from the designer. We derive structure of a database and belonging semantic constraints. These decisions are discussed with sample relations, only. Therefore, the tool is even for users which do not have formal knowledge about databases and semantic constraints usable.

Main parts of the described methods for acquisition of semantic constraints are implemented. They are part of a complete system for database design (Rapid Application and Database Development) that is developed in our groups.

The derivation of semantic constraints from sample databases, the heuristic rules for determining of candidates for semantic constraints and the informal validation are also usable for reverse engineering. These methods can be taken to find out the valid semantic constraints of databases. These semantic constraints are prerequisite for translating existing databases into other data models.

### References


Ein Werkzeug zur Gewinnung semantischer Constraints aus
natürlichsprachlichen Eingaben und Beispieldaten *

Meike Albrecht, Edith Buchholz, Antje Düsterhöft, Bernhard Thalheim

Technische Universität Cottbus, Fachbereich Informatik
Universität Rostock, Fachbereich Informatik

Zusammenfassung

1 Einleitung

*Die Arbeiten am Projekt werden z.T. über die DFG unter dem Kennzeichen Th 465/2 unterstützt.
1Ehemals RAD abgekürzt, da dieser Name bereits als Softwarewerkzeug im europäischen und amerikanischen Raum lizensiert ist, erfolgte die Änderung.
Entwurfsystem kann einen hochwertigen Entwurf garantieren, aber ein solches System kann es erleichtern, einen guten Entwurf zu finden.

In Zusammenarbeit mit Arbeitgruppen der Universitäten Münster und Dresden wird eine integrierte Toolbox zur Unterstützung des gesamten Entwurfsprozesses entwickelt, die auf der Grundlage eines erweiterten Entity-Relationship Modells [BOT90] nicht nur einem oder mehreren Entwerfern die Darstellung von Struktur, Semantik, Funktionalität und Verhalten einer Datenbankanwendung erlaubt, sondern auch über Transformations-, Modifikations- und Nutzerberatungs- und -anpassungstools verfügt (Vorstellung des Gesamtprojektes in [TAA94]).

Im Rahmen dieser Arbeit werden Methoden zur Semantikakquisition vorgestellt, die

- das linguistische Wissen für den Datenbankentwurf und die Erfassung der semantischen Zusammenhänge verwenden,
- zeigen, inwieweit eine Suche nach semantischen Constraints mit Heuristiken und eine informale Validierung der Constraints möglich ist,
- diese Zugänge in das RADD-Projekt integrieren und entsprechende Tools implementieren.


2 Probleme bei der Semantikakquisition


Die formale Angabe semantischer Constraints ist eine der schwierigsten Aufgaben im Datenbankentwurf. In dem hier vorgestellten Zugang beschränken wir uns auf die im Datenbankentwurf am häufigsten verwendeten Constraints (Schlüssel, funktionale Abhängigkeiten, Inklusions- und Exklusionsabhängigkeiten, Kardinalitäten). Die Schwierigkeit der Erfahrung dieser Constraints ist auf folgende Probleme zurückführbar:

- Der traditionelle Zugang zur Erfassung der Semantik basiert auf einer abstrakten mathematischen Darstellung von Integritätsbedingungen. Dabei wird eine hohe Abstraktionsfähigkeit vorausgesetzt. Diese Forderung ist nicht immer erfüllbar. Sogar bei kleinen Schemata ist der Abstraktionsgrad von semantischen Constraints so hoch,
daß selbst erfahrene Datenbankentwerfer überfordert werden, wie einige Tests mit Nutzergruppen verschiedener Systeme ergaben.

- Fast alle Entwurfsmethoden erwarten vom Entwerfer eine vollständige Angabe der Semantik. Normalisierungsalgorithmen liefern z.B. nur dann eine korrekte Normalform, wenn die Menge der funktionalen (und mehrwertigen) Abhängigkeiten vollständig angegeben wurde und alle nicht angegebenen Abhängigkeiten sich entweder aus der angegebenen Menge ableiten lassen oder nicht gelten. Oft werden aber einige semantische Constraints vergessen, obwohl oder auch weil sie offensichtlich sind, oder semantische Constraints sind zu kompliziert, besonders, wenn sie über mehreren Attributen definiert sind, und werden deshalb entweder gar nicht gesehen oder falsch angegeben.

Wünschenswert wäre also eine Unterstützung bei der Semantikakquisition, bei der dem Entwerfer semantische Constraints vorgeschlagen werden. Dabei treten weitere Probleme auf:

- Insbesondere bei komplizierten semantischen Constraints, die über mehreren Attributen definiert sind, ist es schwer zu entscheiden, ob diese gelten oder nicht.
- Es müssen potentiell exponentiell viele mögliche semantische Constraints untersucht werden, um die vollständige Menge semantischer Constraints zu finden.
- Nicht alle Constraints können in einfacher und unmißverständlicher Form graphisch dargestellt werden.

Als Lösung dieser Probleme bieten sich folgende Ansätze an:
1. Es ist möglich, semantische Constraints informal zu erfragen.
2. Eine starke Einschränkung des Suchraumes kann durch einen intelligenten Zugang, der Heuristiken zur Beschleunigung der Suche verwendet, erreicht werden.

Ein Ansatz, der diese Möglichkeiten nutzt, soll hier gezeigt werden.

### 3 Ein Lösungsansatz


Die Kandidaten für semantische Constraints werden gewichtet, um zuerst die Kandidaten
zu validieren, die am plausibelsten erscheinen. Im Dialog mit dem Entwerfer kann dann anhand von Beispielen eine Validierung der Kandidaten erfolgen. Da alle erkannten Kandidaten, die ja vage sind, validiert werden, liefert dieses Vorgehen eine Menge richtiger Constraints. Das folgende Bild zeigt die Kombination dieser beiden Methoden.

Die einzelnen Komponenten der Methode werden in den folgenden Abschnitten ausführlich beschrieben.

4 Abgrenzung gegenüber anderen Arbeiten


5 Linguistischer Zugang

5.1 Motivation und Ziele des linguistischen Zugangs


Im Datenbankentwurf wird zuerst die Strukturierung einer Anwendung zumindest im Groben festgelegt. Zugleich wird eine eingeschränkte Fachsprache gewählt, in der Worte nicht
mehr über alle möglichen Interpretationsmöglichkeiten verfügen. Mit dieser Fachsprache sind auftretende Ambiguitäten oft eindeutig auflösbar. Gerade die deutsche Sprache verfügt aufgrund ihrer variablen Satzstruktur über noch umfangreichere Darstellungsmittel in dieser Beziehung als die englische Sprache.


Es hat sich aber auch gezeigt, daß ein ‘general-purpose system’ für komplexere Anwendungen weder realisierbar noch wünschenswert erscheint. Komplexere Anwendungen zeichnen sich sowohl durch eine semantische Vielfalt als auch durch eine stärkere Benutzung formalisierbarer Teile aus. Der Datenbankentwurf ist ein Beispiel einer solchen komplexen Anwendung. Er führt zum einen auf eine formalierte Beschreibung, setzt aber zum anderen eine Sprache voraus, die bereits eine gewisse Standardform besitzt. Dadurch ist der Datenbankentwurf sehr gut geeignet, die bereits z. T. vollständig ausgearbeiteten Verfahren der Computerlinguistik zu verwenden und zum anderen zu zeigen, wie durch ein natürlichsprachiges Interface der Entwurf effektiver gestaltet werden kann.

5.2 Prinzipielle Analyse natürlichsprachlicher Eingaben


5.3 Extraktion von Semantik — Linguistische und Pragmatische Basis

Transformationsprozeß

Die Erfassung von Datenbank-Semantik aus natürlichsprachlichen Eingaben erfolgt während des Interpretationsprozesses, also in der Pragmatik. Es werden Funktionen innerhalb der eingesetzten attribuierten Grammatik genutzt, um die Semantik des Satzes bzw. der Wörter auf die Datenbank-Semantik (z.B. Erkennen von Schlüsseln, Kardinalitäten) abzubilden. Dazu werden semantische Feinheiten, die der Entwerfer meist unbewußt nutzt, ausgewertet. Erkenntnisse der Computerlinguistik unterstützen diesen Transformationsprozeß ([Pin93]).

Computerlinguistische Prinzipien

Für die Semantikerfassung können wir eine Reihe von linguistischen/ computerlinguistischen Methoden nutzen, die Regeln für die Bedeutungszuordnungen zu natürlichsprachlichen Äußerungen beschreiben.

Prinzip der Kompositionalityät. [GaM89, 280] beschreiben das Fregesche Prinzip als: 'The meaning of the whole sentence is a function of the meaning of the parts.' Die Bedeutung eines Satzes setzt sich somit zusammen aus der Bedeutung der Satzteile. Die Bedeutung dieser wird dann durch die Bedeutung der Subphrasen beschrieben und so weiter; letztlich gilt es, die Bedeutung der Wörter zu untersuchen. Grundlage für die Satzsemantik bildet also eine im Lexikon definierte Wortsemantik.

Betrachten wir dieses Prinzip für unsere Anwendung, den Datenbankentwurf, so müssen wir klären, welche Satzphrasen eine wichtige Rolle bei der Semantikakquisition spielen und welche Wörter die Basis der Phrasen bilden (vgl. auch [Pin86]).


Pragmatischer Ansatz

Gewisse Teile der Fachsprachen sowie Teile der Umgangssprache allgemein zeichnen sich durch bestimmte über die Anwendungsgebiete hinausgehende pragmatische Eigenschaften aus. Diese auf Beobachtungen basierenden Eigenschaften betreffen insbesondere gewisse Lesarten von Wörtern/Sätzen die dem Nutzer plausibler oder 'auf den ersten Blick logischer' erscheinen. Zum Beispiel wird mit dem Wort 'mehrere' eine Menge von Elementen, die größer als zwei sind beschrieben; selten ist eine Menge mit zwei Elementen gemeint. Die Erfassung von semantischen Informationen hinsichtlich der zu entwerfenden Datenbank stützt sich auf eine Reihe solcher heuristischen, linguistisch motivierten Annahmen,
die in die Wissensbasis integriert sind. Der folgende Abschnitt beschreibt einige dieser Annahmen.

5.4 Ausgewählte Beispiele der Semantikerfassung

Schlüsselkandidaten

Annahmen für das Erkennen von Schlüsselkandidaten sind:

1. das zuerst genannte Attribut eines Entities wird als Schlüsselkandidat aufgestellt
2. Attribute, die Nummerncharakter haben, sind Schlüsselkandidaten
3. künstlich eingeführte, im Lexikon ausgezeichnete, identifizierende Substantive sind Schlüsselkandidaten.

Die erfassten Schlüsselkandidaten werden in der Form \textit{keycand(Entity-Relationship,Schlüsselattribut)} dargestellt.

Die erste Annahme beruht auf der Beobachtung, daß der Nutzer das ihm wichtigste Attribut zuerst nennt.

Beispiel: 'Ein Gewässer hat einen Namen und einen Standort.'
Schlüssel: keycand(gewässer,name)


Beispiel: 'Eine Messung ist gekennzeichnet durch ein Datum, eine Uhrzeit und eine Konzentration.'
Schlüssel: keycand(messung,datum)  
keycand(messung,uhrzeit)

Der dritten Annahme liegt eine Charakterisierung von bestimmten Substantiven zugrunde, die diese als identifizierend einstuft. Zu dieser gehören z.B. Wörter wie Kennzeichnung, Bezeichnung, Benennung, Identifikator.

Beispiel: 'eine Ionenkonzentration hat eine Bezeichnung und einen Typ.'
Schlüssel: keycand(ionenkonzentration,bezeichnung)

Bei der Erfassung von Wörtern, die Schlüsselannahmen reflektieren, werden Synonym- und Häufigkeitswörterbücher genutzt.

Kardinalitäten

Bei der Akquisition von Kardinalitäten aus natürlichsprachlichen Eingaben liefern uns linguistische Untersuchen zum Auftreten von bestimmten Determinantien (z.B. ein, der, jeder, mehrere) die Grundlage für die pragmatische Umsetzung. Der Entwerfer beschreibt (bewußt oder unbewußt) mittels dieser Wörter seine Anwendung insofern scharf, daß er mögliche Kardinalitäten definiert und ihm dabei eine bestimmte Lesart plausibler erscheint als eine andere. Diese Reihenfolge wird durch die in den Fakten enthaltene Numerierung
beschrieben. Die folgenden Beispiele illustrieren die Sichtweise auf bestimmte Determinan-
tien.
Die erfassten Kardinalitäten werden in der Form \texttt{cardcand(Nr,Relationship-Name,Entity-
Name,MinKard,MaxKard)} dargestellt.
Zum Beispiel kann das Wort 'ein' immer nur bedeuten:
- mindestens ein (1:n) oder
- genau ein (1:1).
Alle anderen Möglichkeiten der Verwendung von 'ein' werden ausgeschlossen.
Benutzt der Entwerfer explizit in seiner Beschreibung dieses Wort, so nehmen wir an, daß
er eher eine 1:n Kardinalität darstellen möchte.
\textbf{Beispiel:} ‘Ein Gewässer weist eine Ionenkonzentration auf.’
\textbf{Kardinalitäten:} \texttt{cardcand(1,aufweisen,gewässer,1,1)}
\texttt{cardcand(2,aufweisen,gewässer,1,n)}
\texttt{cardcand(1,aufweisen,ionenkonzentration,1,1)}
\texttt{cardcand(2,aufweisen,ionenkonzentration,1,n)}
Der Nullartikel (Auftretens keines Artikels) wird meistens in Verbindung mit Pluralia
gebraucht. Diese suggerieren das wiederholte Vorkommen der Objekte oder Aktanten. Wir
gehen davon aus, daß der Entwerfer bei Verwendung des Nullartikels keine Konkretisierung
Objekte/Aktanten (Ausführende) beschreiben will, sondern daß ihm eine 1:n Kardinalität
plausibler erscheint.
\textbf{Beispiel:} ‘Gewässer werden durch Faunen besiedelt.’
\textbf{Kardinalitäten:} \texttt{cardcand(1,besiedeln,gewässer,1,n)}
\texttt{cardcand(2,besiedeln,gewässer,1,1)}
\texttt{cardcand(1,besiedeln,fauna,1,n)}
\texttt{cardcand(2,besiedeln,fauna,1,1)}
Das Wort 'mehrere' im Gegensatz zu 'ein' referenziert immer eine 1:n Beziehung. Die
Betrachtung von 'genau ein' ist unwahrscheinlich.
\textbf{Beispiel:} ‘Für ein Gewässer werden mehrere Eintrittspunkte definiert.’
\textbf{Kardinalitäten:} \texttt{cardcand(1,definieren,gewässer,1,1)}
\texttt{cardcand(2,definieren,gewässer,1,n)}
\texttt{cardcand(1,definieren,eintrittspunkt,1,n)}

Bei der Aufnahme von Determinantien in die Wissensbasis müssen diese hinsichtlich ih-
ner Kardinalitäten-Mächtigkeit untersucht und gekennzeichnet werden. (Für das Deutsche
sind die Determinantien eine überschaubare, endliche Menge von Wörtern.) Die Angaben
zur Kardinalitäten-Mächtigkeit beschreiben innere zwingende und mögliche Eigenschaften
der Determinantien und sind somit anwendungsunabhängig.

\textbf{In- und Exklusionsabhängigkeiten}

In- bzw. Exklusionsabhängigkeiten werden durch die natürlichsprachliche Unterstützung
in Aufzählungen von Entities vermutet, zu denen ein Oberbegriff existiert. Aufzählungen
treten auf, wenn Substantive mittels Konnektoren verbunden sind; z.B. und, oder, sowohl
als auch.
Die erkannten In- und Exklusionsabhängigkeiten werden in der Form \texttt{inklcand(Entity/ Re-
lationship, enthaltenes Entity/ Relationship), exklcand(Liste der Entities /Relationship)} dar-
gestellt.
Beispiel: 'Seen, Flüsse und Teiche sind Gewässer.'
Exklusionsabhängigkeit: exklcand(see, fluss, teich)
Inklusionsabhängigkeit: inklcand(gewässer, see)
inklcand(gewässer, fluss)
inklcand(gewässer, teich)

Synonyme
Für die weitere Bestimmung der In- und Exklusionsabhängigkeiten durch Beispiele werden in der linguistischen Unterstützung für alle auftretenden Identifikatoren (Entity-Namen, Relationsship-Namen, Attributnamen) aus dem Lexikon entsprechende Synonyme gesucht. Treten diese Synonyme auch im Skelettentwurf auf, so werden sie in Form eines Faktens synonym(identifikator, liste-der-synonyme) bereitgestellt.

5.5 Ein Beispieldialog

Der folgende Beispieldialog illustriert den prinzipiellen Dialogverlauf. Im Dialog äußert der Entwerfer sogenannte Inhaltsaussagen wie z.B. 'Für ein Gewässer werden mehrere Eintrittspunkte definiert.' auf sogenannte Inhaltsabforderungen, die sich im Beispieldialog hinter der Aufforderung 'Gibt es weitere Details?' verborgen. Das System stellt linguistisch und pragmatisch orientierte Nachfragen. Eine linguistische Nachfrage wäre z.B. 'Auf welche Art und Weise wird die Aktion - definieren - durchgeführt?' Die Nachfrage vom System 'Wodurch ist - Eintrittspunkt - gekennzeichnet?' ist eine pragmatische Nachfrage, die darauf zielt, den Entwurf zu vervollständigen.


```plaintext
> : ...
- Ein Gewässer weist Ionen auf.
> : Wodurch ist - Gewässer - gekennzeichnet?
- Ein Gewässer hat einen Namen und einen Standort.
> : Wodurch ist - Ion - gekennzeichnet?
- Durch den Typ, die Art und eine Bezeichnung.
> : Gibt es weitere Details?
- Seen, Flüsse und Teiche sind Gewässer.
> : Muß man - See - außer mit
- Name -
- Standort -
noch weiter kennzeichnen? Wenn ja, wie?
- Ein See hat eine Größe und eine Tiefe.
```
 Folgender erster Strukturentwurf wurde aus diesem Beispieldialog abgeleitet:
Aus dem Dialog wurden folgende Schlüsselkandidaten abgeleitet:

keycand(gewässer, name).
keycand(ion, bezeichnung).
keycand(see, name).
keycand(see, größe).
keycand(teich, name).
keycand(teich, größe).
keycand(see, tiefe).
keycand(teich, tiefe).
keycand(messen, tag).
keycand(messen, tiefe).
keycand(definieren, datenblattnummer).

Folgende Kandidaten für Kardinalitäten wurden aus dem Dialog erstellt:

cardcand(1, aufweisen, gewässer, 1, 1).
cardcand(2, aufweisen, gewässer, 1, n).
cardcand(1, aufweisen, ion, 1, n).
cardcand(2, aufweisen, ion, 1, 1).
cardcand(1, definieren, gewässer, 1, 1).
cardcand(2, definieren, gewässer, 1, n).
cardcand(1, definieren, eintrittspunkt, 1, n).
cardcand(1, messen, eintrittspunkt, 1, 1).
cardcand(2, messen, eintrittspunkt, 1, n).
cardcand(1, messen, ion, 1, n).
cardcand(2, messen, ion, 1, 1).
cardcand(1, besiedeln, gewässer, 1, 1).
cardcand(1, besiedeln, fauna, 1, n).
cardcand(2, besiedeln, fauna, 1, 1).

Aus dem Dialog wurden folgende In- und Exklusionsabhängigkeiten erstellt sowie folgende Synonyme aus dem Lexikon extrahiert:

exklcand([see, fluß, teich]).
inklcand(gewässer, see).
inklcand(gewässer, fluß).
inklcand(gewässer, teich).
synonym(name, [bezeichnung]).
synonym(standort, [ort]).
Nach der Analyse der natürlichsprachlichen Eingaben hat man auf diese Weise einen ersten Datenbankentwurf sowie Kandidaten für semantische Constraints erhalten.

### 6 Strukturvalidierung und Beispieldateneingabe

Die erhaltenen Datenbankentwürfe müssen bestätigt oder korrigiert werden, dazu ist eine graphische Darstellung der entworfenen Strukturen in Entity-Relationship Diagrammen, wie sie im vorigen Abschnitt gezeigt wurde, günstig. Die graphische Darstellung erfolgt in einem Editor (vorgestellt in [TAA94]), hier kann der abgeleitete Entwurf vom Entwerfer verändert werden. Der graphische Editor ist nicht nur ein Zeichentool, er enthält Erklärungs- und Nutzerführungskomponenten, die die Bestätigung oder Änderung der Entwürfe erleichtern.

Die entstandenen Entwürfe müssen gegebenenfalls noch um Attributtypen und -längen vervollständigt werden. Dieses sind technische Details, es ist deshalb nicht sinnvoll, diese bereits im natürlichsprachlichen Entwurf vollständig zu erfragen.

Weiterhin ist es günstig, frühzeitig Beispieldaten zu der entworfenen Datenbank eingeben zu lassen. Hierbei werden zunächst zu den Entities in der Datenbank Beispieldaten erfragt.


<table>
<thead>
<tr>
<th>Name</th>
<th>Standort</th>
<th>Größe</th>
<th>Tiefe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pel</td>
<td>Grö 10.2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Rie</td>
<td>Grö 5.3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Saa</td>
<td>Grö 15.9</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Wald</td>
<td>Grö 4.8</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ion</th>
<th>Bezeichnung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kation</td>
<td>Fe</td>
</tr>
<tr>
<td>Kation</td>
<td>Mn</td>
</tr>
<tr>
<td>Anion</td>
<td>Cl</td>
</tr>
<tr>
<td>Anion</td>
<td>SO4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eintrittspunkt Ort</th>
<th>Ion Bezeichnung</th>
<th>Tag</th>
<th>Konzentration</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.10’/15’/47.50’</td>
<td>Fe</td>
<td>15.8</td>
<td>0.20</td>
</tr>
<tr>
<td>14.10’/15’/47.50’</td>
<td>Mn</td>
<td>15.8</td>
<td>0.30</td>
</tr>
<tr>
<td>14.10’/35’/47.48’</td>
<td>Fe</td>
<td>20.8</td>
<td>0.06</td>
</tr>
</tbody>
</table>
Durch die frühzeitige Verwendung von Beispieldaten im Entwurf können vom Entwerfer Unvollständigkeiten und Fehler in der Datenbank gefunden werden. Weiterhin können die eingegebenen Beispiele verwendet werden, um zu überprüfen, ob die Kandidaten für semantische Constraints wirklich gelten können und wie plausibel sie sind. Diese Überprüfung wird im nächsten Abschnitt gezeigt.

7 Semantikakquisition und informelle Semantikvalidierung


7.1 Ermittlung weiterer Kandidaten für semantische Constraints


Heuristiken zur Schlüsselsuche

In der Struktur, den Beispieldaten und der bereits bekannten Semantik können Hinweise auf Schlüssel vorhanden sein. Einige davon werden hier aufgezählt.

- Kandidaten für Schlüssel können aus Attributnamen (bestimmten Teilstrings in den Namen wie -name-, -nummer-, -#-, -id-) vermutet werden.

- Oft existieren in Relationen künstliche Schlüssel, die als sehr große Integerwerte vereinbart sind. Ist eines der Attribute so definiert, dann ist es wahrscheinlicher Schlüssel als andere Attribute.

- Laufende Nummern in den Beispieldaten sind ebenfalls ein starker Hinweis, daß dieses Attribut Schlüssel ist.

- Tritt in der Relation eine Attributmenge auf, die in einer anderen Relation bereits als Schlüssel bestimmt wurde, so kann die Schlusseleigenschaft auch in dieser Relation gelten.

Heuristiken zur Suche nach Inklusions- und Exklusionsabhängigkeiten


Nicht immer sind Attributen mit dem gleichen Attributnamen Analoga. Es kann Bezeichnungen geben (z.B. Nummer, Name, Preis), die mehrfach innerhalb einer Datenbank verwendet werden und dabei unterschiedliche Bedeutung haben. Deshalb werden mehrere Merkmale zur Suche nach Analoga herangezogen.

- Die notwendige Bedingung zur Suche nach Analoga ist, daß Attribute in den Typen übereinstimmen müssen und ähnliche oder gleiche Längen haben müssen. Es gibt jedoch darüber hinaus weitere Hinweise auf Analoga.

- Gleiche oder synonyme Attributnamen sind ein Hinweis auf Analoga. (Aus dem Wörterbuch der natürlichsprachlichen Analyse kann ermittelt werden, welche Substantive synonym verwendet wurden.)

- Ebenfalls deuten Knotenamen, die synonym verwendet werden können, darauf hin, daß sich zwischen diesen Knoten Analoga befinden.

- Kommen in den Beispieleinträgen gleiche Werte vor, so ist das ebenfalls ein Hinweis auf Analoga.

- Gibt es zwischen zwei Knoten bereits Attribute mit gleicher Bedeutung, so können hier auch weitere Analoga auftreten.

- Besteht zwischen mehreren Attributen die gleiche intrarelationale Semantik (Schlüssel der Relation oder funktionale Abhängigkeiten zwischen diesen Attributen), so ist das ein zusätzlicher Hinweis auf Analoga.

Hat man Analoga gefunden, so werden in Auswertung der Beispieleinträge Kandidaten für Inklusions- und Exklusionsabhängigkeit über diesen ermittelt.

Treten in den Beispieleinträgen dieser Analoga keine gemeinsamen Werte auf, so wird eine Exklusionsabhängigkeit vermutet. Treten gemeinsame Einträge auf, so wird nachfolgend versucht, eine Inklusionsabhängigkeit zu validieren.

Ist diese Validierung nicht erfolgreich, so muß versucht werden, diese Constraints zu präzisieren. Wurde z.B. herausgefunden, daß eine Exklusionsabhängigkeit nicht gilt, so muß man untersuchen, ob hier sogar eine Inklusionsabhängigkeit auftritt.

Heuristiken zur Suche nach weiteren Constraints

Auch zur Suche nach funktionalen Abhängigkeiten und Kardinalitäten gibt es verschiedene Heuristiken. Diese werten vorwiegend die Beispieldaten, sowie die bereits validierten semantischen Constraints aus.
7.2 Überprüfung, ob Kandidaten möglich sind

Es wurde gezeigt, daß sowohl über die natürlichsprachige Analyse als auch über Heuristiken Kandidaten für semantische Constraints, die in der entworfenen Datenbank gelten können, ermittelt werden. Man verfügt über weitere Informationen zur Datenbank, die Beispieleinträge und die bereits validierte Semantik. Anhand dieser wird geprüft, ob gefundene Kandidaten überhaupt auftreten können.


Folgende Schlüsselkandidaten der natürlichsprachlichen Analyse sind bereit durch die in Abschnitt 6 gezeigten Beispiel data widerlegt:

keycand(see,tiefe)
keycand(messen,tag)

Auch funktionale Abhängigkeiten, Exklusionsabhängigkeiten oder Kardinalitäten können bereits im Beispiel nicht erfüllt sein.

Auch die ermittelten Kardinalitäten

\[
\text{cardcand}(1,\text{messen},\text{eintrittspunkt},1,1) \\
\text{cardcand}(2,\text{messen},\text{ion},1,1)
\]

sind durch das Beispiel in Abschnitt 6 widerlegt.

In diesen Fällen sind die ermittelten Kandidaten falsch und werden nicht weiter untersucht.

Es ist auch möglich, daß Kandidaten für semantische Constraints bereits aus anderen semantischen Constraints ableitbar sind, diese müssen dann ebenfalls nicht weiter untersucht werden, da diese Information redundant ist. Z.B. ist eine vermutete funktionale Abhängigkeit redundant, wenn die linke Seite der Abhängigkeit Schlüssel der Relation ist.

Kandidaten für Constraints können auch im Widerspruch zu bereits ermittelten semantischen Constraints stehen, in diesem Fall weiß man, daß diese Kandidaten nicht gelten können, es würden also falsche Kandidaten abgeleitet.

Es müssen nur solche Kandidaten weiter untersucht werden, die nicht im Widerspruch zum Beispiel oder zu den bereits validierten semantischen Constraints stehen und aus den semantischen Constraints auch noch nicht ableitbar sind.

7.3 Wichtung der Kandidaten für semantische Constraints

Die ermittelten Kandidaten werden deshalb vor der Validierung gewichtet, d.h. es wird abgeschätzt, wie plausibel diese Kandidaten sind.

Zur Wichtung der Kandidaten werden die in Abschnitt 7.1 beschriebenen Heuristiken angewendet und gegeneinander gewichtet. Manche Heuristiken deuten stärker auf geltende Constraints hin als andere, daher wird eine gewichtete Summe zur Abschätzung eingesetzt. Eine einfache Näherungsformel genügt diesen Anforderungen. Für das Beispiel Schlüssel:

\[
P(A \text{ ist Schlüsselattribut}) := w_1 r_1(A) + w_2 r_2(A) + w_3 r_3(A) + w_4 r_4(A)
\]

\[r_i - \text{Ergebnis der Heuristikregel } i \in [0..1]
\]

\[w_i \text{ sind die Gewichte zwischen 0..100 und } w_1 + w_2 + w_3 + w_4 = 100\]


Mit dieser Abschätzung der Plausibilität wird eine Reihenfolge der Validierung festgelegt. Eine Methode zur Validierung wird im nächsten Punkt beschrieben.

### 7.4 Informale Validierung


Für die ermittelte Exklusionsabhängigkeit zwischen See und Fluß müssen zuerst Analogien zwischen den Knoten ermittelt werden, um die Exklusionsabhängigkeit zu konkretisieren. Diese werden hier leicht gefunden, da beide Knoten den gleichen Schlüssel (Name) haben. Die Erfragung dieses Constraints sieht dann so aus:

<table>
<thead>
<tr>
<th>See:</th>
<th>Name</th>
<th>Standort</th>
<th>Größe</th>
<th>Tiefe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fluß:</th>
<th>Name</th>
<th>Standort</th>
<th>Länge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pel</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Könnten in == See.Name == und == Fluß.Name == gleiche Einträge auftreten (j-ja/ n-nein/ u-unbekannt) ? |

Aus der Antwort des Entwerfers lassen sich folgende Schlußfolgerungen ableiten:

- ja  \rightarrow \text{exklcand(see,fluß) gilt nicht}
- nein \rightarrow \text{exklcand(see,fluß) gilt}
Man hat also im Ergebnis der Validierung Informationen über semantische Constraints, die
gelten und semantische Constraints, die nicht erfüllt sind. Constraints, über die noch keine
Aussage bekannt oder ableitbar ist, sind unbekannt. Durch diese “Negativinformationen“
setzt man keine Closed-World-Assumption voraus. Man kann dadurch bestimmen, wann
alle semantischen Constraints einer Relation oder Datenbank bekannt sind.
Bei der vollständigen Akquisition der Semantik müssen alle unbekannten Constraints er-
fragt werden. Begonnen wird dabei mit den plausibelsten Kandidaten.
Die Validierung anhand einer Beispieldiskussion soll auch noch für Kardinalitäten gezeigt
werden.

<table>
<thead>
<tr>
<th>Gewässer.Name</th>
<th>Ion.Bezeichnung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pel</td>
<td>Fe</td>
</tr>
<tr>
<td>Pel</td>
<td>Mn</td>
</tr>
<tr>
<td>Pel</td>
<td>Cl</td>
</tr>
<tr>
<td>Pel</td>
<td>SO4</td>
</tr>
</tbody>
</table>

Können einem Wert in == Gewässer.Name == verschiedene
Werte für == Ion.Bezeichnung == zugeordnet sein
(j-ja/ n-nein/ u-unbekannt) ?

In diesem Fall läßt sich aus der Antwort des Entwerfers ableiten:

ja → cardcand(1,aufweisen,gewässer,1,1) ist falsch
nein → cardcand(1,aufweisen,gewässer,1,1) ist richtig

Die Validierung der Semantik anhand von Beispielen ist für alle hier betrachteten Con-
straints analog möglich. Aus den Antworten des Nutzers können dabei immer formale
semantische Constraints abgeleitet werden.

8 Zusammenfassung/Ausblick

In dieser Arbeit wurde eine Methode vorgestellt, mit der Struktur und Semantik einer
Datenbank über die Analyse naturlichsprachiger Eingaben erkannt und validiert werden
können. Die natürlichsprachliche Entwurfskomponente sowie das Tool zur Akquisition von
Semantik aus Beispielen wurde innerhalb des RADD-Projektes prototypisch implement-
tiert. Für die einzelnen Komponenten wurden günstige Programmiersprachen gewählt,
z.B. wird für das NLI Prolog als Programmiersprache genutzt. Beide Teile, sowie alle an-
deren Tools im RADD-Projekt, laufen als parallele Prozesse und kommunizieren über ein
Data Dictionary. Das Projekt läuft unter SunOS 5.3.

In analoger Weise ist die Erfassung von Operationen und Verhaltensinformationen über
der Datenbank möglich.
Dazu wird für die natürlichsprachliche Analyse ein Verhaltensmodell benötigt, welches
in die Wissensbasis integriert wird. Alle auftretenden Prozesse werden in dieses Modell
eingeordnet. Für einen Prozeß können dann, nach einer Klassifizierung, Aussagen über
notwendige Pre- und Postprozesse getroffen werden. Das Problem bei der Erstellung eines
solchen Modells besteht darin, daß es möglichst allgemein gestaltet wird.
Zwischen den Operationen einer Datenbank und den semantischen Constraints gibt es Wechselwirkungen. Ist bekannt, welche Operationen auf einer Datenbank laufen, so können hieraus auch Rückschlüsse über mögliche semantische Constraints abgeleitet werden, die Heuristiken können also entsprechend erweitert werden.

9 Danksagung

Wir möchten uns an dieser Stelle bei Andreas Heuer für die Hinweise zu früheren Versionen dieses Artikels bedanken.

Literatur


[Eic84] Ch. F. Eick: From Natural Language Requirements to Good Data Base Definitions - A Data Base Design Methodology. In: Proc. of the International Conference on Data Engineering, pp.324-331, Los Angeles, USA, 24.-27.4.1984


Semantische Analyse im RADD-NLI:  
Zur Interpretation von Possessiv-Relationen

Edith Buchholz, Haiko Cyriaks, Antje Düsterhöft, Bernhard Thalheim

Universität Rostock, Fachbereich Informatik  
e-mail: dust@informatik.uni-rostock.de

Brandenburgische Technische Universität Cottbus, Fachbereich Informatik  
e-mail: thalheim@informatik.tu-cottbus.de


1. Einleitung


Das RADD-NLI erzeugt nach der syntaktischen Analyse eingegebener Sätze Syntaxbäume [Mehlan 95], die anschließend einer semantischen Analyse unterzogen werden. Dabei wird der Satztyp anhand einer, auf das Prädikat des Satzes angewendeten Verbklassifikation bestimmt und die Nominalphrasen semantischen Rollen zugewiesen.

Die Analyse der Sätze

(1) Alle Bestandseinheiten haben einen Titel.
(2) Leser entleihen Bücher.
würde beispielsweise folgendes Ergebnis liefern (vereinfacht):

(1) Satztyp: Possessiv
   Thema: Titel
   Causa: Bestandseinheit
(2) Satztyp: Besitzwechsel
   Thema: Bücher
   Causa: Leser

1.1. Possessivität im Entwurfsmodell


2. Repräsentation von „haben“ im natürlichsprachigen Satz – mögliche possessive Relationen

Possessive Relationen—und dieser Begriff wird hier, wie im folgenden noch gezeigt werden wird, im weiten Sinne gebraucht—sind eines der häufigsten sprachlichen Mittel zum Ausdruck von vielfältigen Beziehungen der realen Welt. Ein deutliches Zeichen dafür ist auch die Vielzahl morphosyntaktischer Mittel, mit denen das semantische Phänomen der possessiven Relation sprachlich realisiert werden kann. Es sind dies im Einzelnen:

   (i) possessive Verben (haben, besitzen, umfassen, etc.)
   (ii) Genitivattribute
   (iii) Possessivartikel
   (iv) Genitiv des Demonstrativartikels (dessen / deren)

Wie häufig possessive Relationen in der natürlichen Sprache Verwendung finden, zeigt z.B. die Tatsache, daß das Verb haben das nach sein zweithäufigste Vollverb der (gesprochenen) deutschen Sprache ist. [Fuchs & al. 81] Wir haben es also z.B. bei haben mit einem hochgradig ambigen Verb zu tun, bei dessen Interpretation vielfältige Möglichkeiten gegeben sind.

2.1. Verschiedene Klassen von Possessivität

Der Terminologie von [Bisle-Müller 91] folgend, soll das durch einen Possessivartikel determinierte Objekt (possessives) Referenzobjekt, das Objekt, auf das die
Zugehörigkeitsbeziehung gerichtet ist, **Referenzrahmen** genannt werden.

(3) Die Bibliothek hat einen Bestand.

*Die Bibliothek* ist somit in (1) der Referenzrahmen, *Bestand* der Bibliothek das possessive Referenzobjekt.

Possessivität bezeichnet keineswegs—wie die deutsche Bezeichnung „besitzanzeigenend“, z.B. in „besitzanzeigenendes Fürwort“ für „Possessivpronomen“ fälschlich suggeriert—nur Besitzverhältnisse, wenn dies sicher auch eine prototypische Anwendung possessiver Relationen ist. Bei genauer Betrachtung läßt sich gleichwohl eine Reihe verschiedener Klassen von possessiven Relationen feststellen. (nach [Bisle-Müller 91], [Cyriaks 95])

a) Teil-Ganzes-Relation
   (i) **Objekt** -> **Komponente**

   (4) Die Bibliothek hat einen Lesesaal.

   (ii) **Menge** <-> **Element**

   (5) Die Universität hat viele Studenten.
   (6) Diese Organisationen haben einen Dachverband.

   (iii) **Region** -> **Plätze**

   (7) Indonesien hat viele Inseln.

b) Besitzrelation
   (i) im engen Sinne

   (8) Der Student hat ein Fahrrad.

   (ii) **Verfügung**

   (9) Er hat ein Zimmer im Wohnheim.

   (iii) **Anspruch**

   (10) Ich habe meine Entschädigung nicht bekommen.

c) Erzeugung
   (i) **Produktion**

   (11) Die Firma hat viele Produkte im Angebot.
(12) Dieser Computer hat einen bekannten Hersteller.

**(ii) bei Vorgängen und Ereignissen**

(13) Alles hat ein Ende.

d) Attribut

(14) Sein Übereifer störte die Kollegen sehr.
(15) Dieses Buch hat ein Alter von über 100 Jahren.

e) soziale Beziehung

(16) Jeder Mensch hat Eltern.
(17) Der Angestellte hatte keinen großen Respekt vor seinem Chef.

f) Repräsentation

(18) Jedes Objekt hat einen Identifikator.

Möglicherweise ist diese Aufzählung noch nicht einmal vollständig. Sie verdeutlichen aber bereits, welch hoher Grad an Ambiguität in einem so häufig verwendeten Verb wie *haben* steckt.

Die Schwierigkeit bei der Entwicklung eines natürlichsprachigen Interfaces oder einer anderen Form des automatischen Sprach„verstehens“ besteht nun hier in der richtigen Erkennung der jeweiligen Relation. Mißverständnisse können dabei nicht ausgeschlossen werden und müssen— sofern relevant—durch Nachfragen geklärt werden. Als besonders ambiges Beispiel nennt [Bisle-Müller 91]:

(19) Das ist *sein Bild*.

mit den Interpretationen:
- Das ist das Bild, das ihm gehört.  
  b) (i)
- Das ist das Bild, das er geschaffen hat. 
  c)
- Das ist das Bild, auf dem er dargestellt ist. 
  f)
- Das ist das Bild, über das er verfügt. 
  b) (ii)
- Das ist das Bild, auf das er Anspruch hat. 
  b) (iii)

([Bisle-Müller 91:87])

Die Auflösung von solchen Ambiguitäten, die selbst einem menschlichen Rezeptor Probleme bereiten, wird automatisch kaum möglich sein.

Bei der Behandlung von solchen, in der natürlichen Sprache massiv auftretenden **Mehrdeutigkeiten**, nützt es auch nichts, die verwendete Sprache zu normieren mit dem Ziel,

3. Interpretation possessiver Relationen – allgemein und im RADD-NLI

Allgemein kann die Erkennung, um welchen Typ von Possessivität es sich handelt, durch eine feinere Klassifikation der Substantive, als sie anfangs im NLI im System RADD vorgenommen wurde, erreicht werden.

Zunächst wollten wir auf eine besondere Behandlung der Bedeutung der Substantive soweit wie möglich verzichten. Substantive beschreiben Objekte der realen Welt. Für den Datenbankentwurf sind wir nicht interessiert an der Natur dieser Objekte, wichtig ist leidlich deren Abbildung im konzeptuellen Datenmodell. ([Buchholz & al. 95])

Vor allem bei der Disambiguierung possessiver relationen, aber auch anderer sprachlicher Phänomene hat es sich erwiesen, daß man nicht umhin kommt, auch die Semantik der Substantive näher zu betrachten, denn von ihnen hängt die Interpretation anderer sprachlicher Konstrukte in starkem Maße ab.

Üblich ist eine Kennzeichnung relationaler Nomina. Das sind solche Substantive, die eine Argumentstelle und somit eine inhärente Beziehung zu anderen Objekten besitzen. Beispiele für relationale Nomina sind Vater, Präsident, Schwester, Name, Abbildung etc.. Eine Person kann eben immer nur Schwester einer anderen sein, ein Name ist immer Name eines Objekts. Schwierig ist—wie immer—eine eindeutige Einordnung aller Substantive in diese Klassifikation.

Treten relationale Substantive als Referenzobjekt auf, können bestimmte Lesarten der Possessivität ausgeschlossen werden. Dies sind im einzelnen:
- Besitz
- Menge – Element
- Region – Plätze

Je nachdem, ob man Begriffe wie Dach, Tür, Rad, etc. zu den relationalen Nomina zählt oder nicht, kann auch eine Objekt-Komponente-Beziehung ausgeschlossen werden.

Bei der Auszeichnung von Substantiven, die durch Derivation aus Adjektiven entstanden sind (dumm <- Dummheit, groß <- Größe, alt <- Alter), lassen sich auch Attribut-Beziehungen erkennen.

Die Beziehungen in c) (ii) sind dadurch gekennzeichnet, daß das Referenzobjekt ein
sogenanntes Nomen Actionis (Verbalsubstantiv) ist. Dies sind Substantive, die meist von Verben abgeleitet sind und Handlungen und Vorgänge beschreiben. ([Bußmann 90]) Auch die Kennzeichnung solcher Substantive im Lexikon ist möglich.

Der Ausdruck von Besitzverhältnissen ist im allgemeinen nur dann sinnvoll, wenn der Besitzer ein belebtes Objekt — im besonderen ein Mensch — oder eine Menge von belebten Objekten — wie Firmen, Staaten, etc. — ist. Durch Aufnahme des Merkmals „belebt“ in das Substantivlexikon ließen sich hier Besitzrelationen leichter disambiguieren.

3.1. Realisierung im System RADD


Wesentlich ist also die Erkennung der jeweils vorliegenden Form von Possessivität entsprechend der obigen Klassifizierung. Will man mit möglichst wenig Nachfragen an den Nutzer auskommen, wird man um eine möglichst feine Klassifizierung der Substantive nicht herumkommen. Ausgezeichnet sollen (zusätzlich zur bisher im System RADD verwandten Einteilung in Konkreta/Abstrakta, Unika, Kontinuativa/Individuativa)
- relationale Substantive
- Derivate aus Adjektiven
- Nomina Actionis
- belebte Objekte (im weiteren Sinne, also auch „juristische Personen“)
werden.

Damit lassen sich folgende Heuristiken für die einzelnen Formen der Possessivität aufstellen:

a) Teil-Ganzes-Relation
Referenzobjekt und Referenzrahmen müssen vom selben Typ sein.

b) Besitzrelation

c) Erzeugung

d) Attribut
Das Referenzobjekt ist ein Derivat aus einem Adjektiv.

e) soziale Beziehung
Referenzrahmen und -objekt sind belebt. Das Referenzobjekt ist ein relationales Substantiv.

f) Repräsentation
Das Referenzobjekt ist ein relationales Abstraktum.

Die Beziehungen unter a) - c) sowie e) werden im Datenbankschema als Relationships eingetragen, falls es sich bei Referenzrahmen bzw. Referenzobjekt nicht um bereits erfaßte Attribute handelt. In diesem Falle können mit Hilfe possessiver Relationen auch Aussagen über den Typ eines Attributs gemacht werden:


In den Beziehungen d) und f) werden die Referenzobjekte als Attribute zu den Referenzrahmen abgebildet. Dabei sind Attribute nach f) gute Schlüsselkandidaten, da ein Repräsentant oft geeignet ist, ein Objekt zu identifizieren.

Im folgenden soll an zwei Beispielen die Umsetzung possessiver Relationen auf der konzeptuellen Ebene gezeigt werden:

(21) Alle Bestandseinheiten haben einen Titel.

Die semantischen Analyse ergab:
satz(possessiv,r(icaus,indet,all,titel),
     thema(def,all,bestandseinheit),_(_,__).

(Zur Bedeutung der Repräsentation im System RADD siehe [Buchholz & al. 95], [Cyriaks 95])


FRAGE: Kann jede Bestandseinheit nur durch Kenntnis von Titel eindeutig identifiziert werden?

Beantwortet der Nutzer diese Frage mit „ja“, wird „Titel“ Schlüssel des Entities „Bestandseinheit“, ansonsten nicht.

Die konzeptuelle Analyse ergibt also:
entity(bestandseinheit).
attre(bestandseinheit,titel).
keycand(bestandseinheit,titel).

(22) Bücher haben mindestens einen Autor.

Bei der semantischen Analyse wird folgende Struktur generiert:
satz(possessiv,r(\text{causa}(&\text{indef},&\geq1,\text{autor}),
\text{thema}(&\text{def},&\text{all},\text{buch}),_,_),_).


entity(buch).
entity(autor).
relship(besitz/produktion,autor,[buch]).

Im Kontext einer Bibliothek tritt jedoch die Eigenschaft eines Autoren, auch Person zu sein, in den Hintergrund. Hier steht "Autor" lediglich für einen Namen, der zur Identifizierung eines Buches o.ä. dient. "Autor" würde dann in das semantische Lexikon als relationales Abstraktum aufgenommen und analog zu "Titel" im vorherigen Beispielsatz behandelt werden. Damit würde durch die konzeptuelle Analyse folgendes Schema entstehen:

entity(buch).
attre(buch,autor).
keycand(buch,autor).

4. Zusammenfassung


Das aufgezeigte Problem tritt nicht nur im natürlichsprachigen Datenbankentwurf, sondern auch bei anderen Anwendungen der automatischen Sprachverarbeitung, im natürlichsprachigen Entwurf von Informationssystemen unterschiedlichster Art wie z.B. objektorientierter Systeme, in der natürlichsprachigen Validierung von Entwürfen, das
bedeutet **Generierung** natürlicher Sprache, aber auch beim **maschinellen Übersetzen** etc. auf.

Bei der Entwurfsvalidierung muß der Typ der possessiven Beziehung z.B. dann bestimmt werden, wenn ein bestimmtes, die Art der Beziehung näher charakterisierendes possessives Verb als *haben* im generierten Satz verwendet werden soll.

Die am Beispiel possessiver Relationen gezeigten Probleme beim automatischen „Verstehen“ natürlicher Sprache treten so oder ähnlich bei einer Vielzahl anderer sprachlicher Phänomene auf. So konnten wir zeigen, daß selbst so scheinbar „nebensächliche“ Wörter wie die Artikel der deutschen Sprache bei der Wissensacquisition aus natürlichsprachlicher Eingabe einen erheblichen Aufwand bei der Verarbeitung erfordern. [Cyriaks 95] Das gleiche trifft z.B. auf die Behandlung von Konnektoren etc. zu.

## 5. Literatur


[Cyriaks 95] Cyriaks, H.: Analyse der Semantik der Determinantien und ihre Nutzung für ein natürlichsprachiges Interface im Datenbankentwurfssystem RADD. Diplomarbeit, Universität Rostock, Fachbereich Informatik 1995


Reuse of Database Design Decisions

Meike Klettke
University of Rostock
Database Research Group

Abstract

Reuse of available databases could support database design and reverse-engineering of databases. Thereby, design decisions could be derived from already designed and proven databases.

In this article a method for reusing databases is proposed that is similar to the approach in case-based reasoning. We describe how similar databases can be determined, then we enumerate which available design decisions can be reused and how they have to be revised. Finally, we suggest a method to build libraries that can be used for the process.

The method can be implemented and used in tools.

1 Motivation

Database design is a process of determining the structure of a database, semantics and behavior specification. For this process a designer only can use very informal descriptions about the application. Therefore, database design is very difficult and time-consuming. The results often depend on the skill and inspiration of a designer. But the design process is very important because the usability of a database depends on the design.

Re-engineering of a database consists of a reverse-engineering process and a design process. During the design process the derived conceptual schema is evolved. Thereby, the same problems occur that also exist in database design.

It would be desirable to support the database design process by tools. Tools should check design decisions or suggest improvements. Because of the abstraction process that is necessary to design an database it is very difficult to derive meaningful suggestions automatically. Reuse of already designed and proven databases can speed up and improve all tasks of database design.

In this article, a method is shown that supports the reuse of databases. The approach can be employed by tools.
This article is organised as follows:

First, we start with an overview that shows the main tasks of the reusing approach. In the next section related works are enumerated. The main idea in case-based reasoning is similar to the idea of reusing databases. In both applications we first look for a case/database that is similar to an actual problem. If we could find one then we try to reuse and adapt a solution. In section 4 we want to demonstrate a method to find similar databases. In section 5 we derive design decisions from similar databases that we could adapted onto an actual database. We show that a revise process is necessary (section 6). At the end, we will show two methods how libraries for the reuse process could be organised. We close with a summary.

2 Overview of the method

It is widely accepted that the approach of case-based reasoning requires the following four tasks (that originally have been suggested in [AaP94])

1. RETRIEVE the most similar case
2. REUSE the information and knowledge in that case to solve the problem
3. REVISE the proposed solution
4. RETAIN the parts of this experience likely to be useful for future problem solving

We can adapt these tasks onto the reuse of database design decisions because this problem is similar. Then we get the following tasks:

1. RETRIEVE the most similar database or the most similar part of a database that shall be reused in any way
2. REUSE the information in that database to derive design decision for the actual database
3. REVISE the proposed design decision
4. RETAIN the information - building of libraries that are suited to support the process of reuse

Figure 1 shows how the retrieve reuse and revise process work together. For our approach, we assume the following scenario. A set of already designed databases is available. Further, we have an actual database with incomplete structural, semantic and behavior information. The design of this database shall be completed. We try to find in the set of already designed databases a
available databases

actual database

Retrieve

available solution

Reuse/Revise

actual solution

Figure 1: Reuse of database design decisions

similar one and want to adapt design decisions onto the actual database.

Before we declare the method in the following sections we will give an overview on some related works.

3 Related research

Storey/Chiang/Dey/Goldstein/Sundaresan: Database Design with Common Sense Reasoning and Learning. [SCD97] suggest a system that supports database design by using available databases. Main part of the approach is to determine if pairs of attributes, entities, relationships and parts of databases that are "same", "close" or "different". For that comparison information about names are exploited, an ontology helps to determine more complicated similarities such as synonyms. Based on this comparison the learning of commonly valid databases for one application is realized. These databases are used to support the design of new databases. If no database is available then an abstract business model replaces it. This method was tested with sample databases from well-know database literature. The results are very convinced, but the databases are often from the same group of authors, it may be possible that these results cannot be achieved in every case.

Some methods of the approach we will use in this article in a similar way. Those parts are marked in the text.
Castano/DeAntonellis et al.: Schema Indexing, Clustering, Determining of Similar Databases. Several techniques that are relevant for reusing databases are described in numerous publications. In [CAZ92], [CaA94], [CAF95], and [CaA96] a method is suggested that determines the most important parts of a database (schema descriptors) by exploiting of

- number of paths
- number of attributes
- level of hierarchies

of an object. Based on equal and different schema descriptors the similarities of two schemas is calculated ([CAZ92]). In [CAF95] and [CaA96] this method is extended not only on the schema descriptors, but onto all objects of the schema. Schema abstraction on the basis of schema similarity is described in [CaA94], [CAF95], and [CaA96].

Song/Johannesson/Bubenko: Finding Similarities for Schema Integration. [SJB92] deals with the problem to find semantic similarities as a prerequisite for integrating schemas. The authors compare the meaning of entities and relationships by using "integration knowledge" that contains information about synonyms and subset relationships. Attributes and key attributes are also used to compare the meaning of entities and relationships. Further, equal or compatible cardinalities are determined and included. The results are equivalent, compatible and mergable schemas which can be used for the integration process.

The method of determining similar entities and relationships uses all available information of the conceptual databases therefore this approach resembles the identification of similar concepts that is suggested in this article.

Bergmann/Eisenecker: Reuse of Object-oriented Software. The finding of object-oriented software for the aim of reuse is analyzed. The authors notice that reuse is always a kind of case-based reasoning with the tasks retrieve, reuse, revise and retain. Further, they developed a method for reuse object-oriented software that uses structural and semantic information. They found out that a method based only on structural characteristics (names of methods, number and classes of parameters, return value) returns much poorer results than a method that bases on structural and semantic information.

Distance functions in case-based reasoning. There are many publications that try to improve distance functions for several applications. Some of these are: [Ric92], [Pfe96] and [WiB96].
4 Retrieve

If we want to reuse database design decisions, we first have to identify similar databases. This task is not easy because databases can be very complex and difficult to understand.

Without understanding the meaning of a database completely, we only can exploit available characteristics (names, types, integrity constraints, transactions, and so on) of the database.

Comparing of parts of databases is very complex it is easier to realise a bottom-up approach. Thereby more complex concepts base on others. The following table shows the concepts that we want to compare in this section and on which other concepts these comparisons base on.

<table>
<thead>
<tr>
<th>concept</th>
<th>bases on</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>attributes</td>
</tr>
<tr>
<td>entities</td>
<td>attributes, entities</td>
</tr>
<tr>
<td>relationships</td>
<td>attributes</td>
</tr>
<tr>
<td>entities and relationships</td>
<td>entities, relationships</td>
</tr>
<tr>
<td>databases</td>
<td></td>
</tr>
</tbody>
</table>

We start with methods to find similar attributes in two databases.

4.1 Determining of similar attributes

There are some characteristics that can be used to determine similar attributes of two databases. Now, we suggest several heuristic rules for finding those:

Exploitation of structural information

\( H_{a1} \) It is obvious to take same attribute names as a hint for similar attributes. Also, if one attribute name is substring of another then the attributes could have the same meaning and one name could be abbreviated. If synonyms are exploitable, in ([SJB92], [SCD97], and [CaA96] synonym dictionaries are includes) then we can extend this heuristic rules onto the determining of synonyms.

\( H_{a2} \) Equal or similar types and length of attributes can also deliver hints for identical attributes.

\( H_{a3} \) There are further structural information, for instance enumeration types or default values that can deliver hints. If these characteristics are equal then we derive an additional hint that these attributes may be equal or similar, too.
These structural information deliver reliable hints for similar databases. But we don’t believe that the using of these characteristics will determine all similarities because attributes with the same meaning often don’t have the same names. Synonyms are domain-dependent. It is impossible to use a synonym dictionary that delivers for every case correct results (and the extent of such a dictionary would be tremendous therefore it is impossible to build one). Further, in databases often exists homonyms. As an example: Attribute names “name” and “number” are very often used in a database but these attributes always have a different meaning.

Therefore, we believe that names can help to compare databases but it is better to include more kinds of available characteristics.

**Using of semantic information** If there are integrity constraints that are already specified in the actual database we can enumerate further heuristic rules:

- $H_a^4$. If two attributes $A$ and $B$ are both keys of their entities or relationship then we derive that this gives additional hints to identical attributes $A$ and $B$.

- $H_a^5$. Functional dependencies can also be exploited in the search for similar attributes. If the attributes $A$ and $B$ are both on the left side of the same functional dependency or both on the right side of a dependency than this is an additional hint that the attributes are similar.

**Exploitation of available data** It is possible that data exist in entity-relationship models. The comparison of data can deliver further heuristic rules.

- $H_a^6$. If there are available data in the actual database we can compare these data with the data in the available databases to derive equal attributes. Same data values in two attributes could be a heuristic that these attributes are similar.

If there are further available characteristics of a database for instance behaviour information or transactions then we can develop further heuristic rules that compare these information, too.

Now, we have some heuristic rules that deliver hints for similar attributes. We want to compare and weight all these heuristics to determine a measure for similarity of attributes. This measure shall deliver higher results if more heuristic rules are fulfilled.
The following simple estimation can be used for that task:

\[ \text{sim}(A, B) := \sum_{i=1}^{6} w_i \times H_a(i, A, B) \]

- \( H_a(i) \) - result of heuristic rule \( i \in [0..1] \)
- \( w_i \in [0..1] \)
- \( w_1 + \ldots + w_6 = 1 \)

The weights \( w_i \) should be determined in the following way. The table specifies how reliable the results of the enumerated heuristic rules are:

<table>
<thead>
<tr>
<th>Heuristic Rule</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_a1, H_a6 )</td>
<td>very reliable</td>
</tr>
<tr>
<td>( H_a2, H_a3 )</td>
<td>reliable</td>
</tr>
<tr>
<td>( H_a4, H_a5 )</td>
<td>only in combination with other rules relevant</td>
</tr>
</tbody>
</table>

Heuristic rules which are more reliable shall be weighted higher than the other heuristics.

The similarity of an attribute set can be estimated in the following way:

\[ \text{sim}(A_1..A_n, B_1..B_m) := \max \left\{ \frac{2 \times \sum_{i=1}^{n} \text{sim}(A_i, B_j)}{n + m} \right\} \]

\( j \in 1..m \), every \( j \) occurs only once

Based on these similar attribute sets we search for similar entities.

### 4.2 Determining of similar entities

<table>
<thead>
<tr>
<th>( D_1 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 )</td>
<td>( E_2 )</td>
</tr>
<tr>
<td>( A_{11} \ldots A_{1n} )</td>
<td>( A_{21} \ldots A_{2m} )</td>
</tr>
</tbody>
</table>

If we search for similar entities \((E_1, E_2)\) in two database \((D_1, D_2)\) we can employ a method that bases on the rules for determining similar attributes \((A_{11}..A_{1n}, A_{21}..A_{2m})\). There are addition characteristic of the entities that can be included:

**Structural characteristics**

\( H_e1 \) Same entity names, substrings in entity names or synonym entity names are characteristics that indicate similar entities.
**Integrity constraints**

*H*_2 If two entities _E_1 and _E_2 have the *same keys* then these entities can be same or similar.

Both heuristic rules deliver very reliable results, (therefore both could get the same weight.) These heuristics can be taken to estimate a similarity measure for entities:

\[ 
\text{sim}(E_1, E_2) := \frac{1}{2} \sum_{i=1}^{2} w_i \ast H_{i,i}(E_1, E_2) + \frac{1}{2} \text{sim}(A_{11..A_{1n}}, A_{21..A_{2m}}) 
\]

_H_2 - result of heuristic rule \( \in [0..1] \)

\[ w_i \in [0, 1] \]

\[ w_1 + w_2 = 1 \]

Thereby, the estimation of similar attributes enters into this calculation.

### 4.3 Determining of similar relationships

If we search for similar relationships \((R_1, R_2)\) in two database \((D_1, D_2)\) we can use the rules for determining similar entities \((E_{11}, E_{12} \text{ and } E_{21}, E_{22})\) and similar attributes of the relationships \((A_{11..A_{1n}}, A_{21..A_{2m}})\). There are further characteristic of the relationships that can be used:

**Structural characteristics**

*H_*1 *Same relationship names, same parts of a relationship name or synonyms* in the names can give a hint for similar relationships.

**Integrity constraints**

*H_*2 *Same keys* of two relationships _R_1 and _R_2 can indicate that the relationships are similar.
$H_{r3}$ Same inclusion an exclusion dependencies (if available) deliver additional hints to similar relationships.

$H_{r4}$ We want to determine an estimation of a similarity measure. Thereby, we include the similarity of entities. Further, we also want to compare the cardinalities, therefore an additional heuristic rule is added. Same cardinality constraints deliver additional hints for the similarity of relationships.

The following table specifies how reliable the results of the heuristic rules are. The weights $w_i$ of the rules derive from this overview:

<table>
<thead>
<tr>
<th>very reliable</th>
<th>$H_{r1}, H_{r2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>only in combination with other rules relevant</td>
<td>$H_{r3}, H_{r4}$</td>
</tr>
</tbody>
</table>

The heuristics can be weighted in the following way:

$$sim(R_1, R_2) := \frac{1}{4} \sum_{i=1}^{3} w_i \cdot H_{r,i}(R_1, R_2) + \frac{1}{8} \cdot sim(B_{11}..B_{1n}, B_{21}..B_{2m}) + \frac{1}{8} \cdot sim(E_{11}, E_{12}) + \frac{1}{8} \cdot H_{r4}(R_1, E_{11}, R_2, E_{21}) + \frac{1}{8} \cdot sim(E_{21}, E_{22}) + \frac{1}{8} \cdot H_{r4}(R_1, E_{12}, R_2, E_{22})$$

$H_{r,i}$ - result of heuristic rule $\in [0, 1]$

$w_i \in [0, 1] \quad 0..1 \\
0 + w_2 + w_3 = 1$

In this estimation there are two possibilities to compare the belonging entities:

- Comparison of $E_{11} - E_{21}$ and $E_{12} - E_{22}$ or
- comparison of $E_{11} - E_{22}$ and $E_{12} - E_{12}$

the one with the highest similarity measure is chosen.

### 4.4 Comparing entities and relationships

Different structural descriptions of two databases can have the same semantics. To find those cases, we also want to compare entities and relationships of two databases.

$D_1: \begin{array}{c}
\ldots & E & \ldots \\
A_1 & \ldots & \ldots \\
A_n & \ldots & \ldots \\
\ldots & \ldots & \ldots \\
\end{array}$

$D_2: \begin{array}{c}
\ldots & B_1 & \ldots \\
R_1 & \ldots & \ldots \\
B_{m} & \ldots & \ldots \\
\ldots & \ldots & \ldots \\
\end{array}$

**Structural characteristics**

$H_{er1}$ Same entity respectively relationship names, same parts of a entity/relationship name or synonym names are is one characteristic that indicate similar entities and relationships.
Using of integrity constraints

H
ε,2 Same keys of an entity and a relationship can indicate that these are similar.

Both heuristic rules deliver very reliable results. (therefore both could get the same weight.) The following estimation can determine the similarity of those concepts.

\[
\text{sim}(E, R) := \frac{1}{4} \text{sim}(A_1..A_n, B_1..B_m) + \frac{1}{4} \sum_{i=1}^{2} w_i \ast H_{\varepsilon,i}(E, R)
\]

\(H_{\varepsilon,i}\) - result of heuristic rule \(\in [0..1]\)
\(w_i \in [0, 1]\)
\(w_1 + w_2 = 1\)

Also the estimation of the belonging attributes is included in this measure.

4.5 Comparing of more complex parts of databases

In the last paragraph we mentioned that same content can be represented in different structural descriptions. This story is continued.

An attribute of one database can decode the same information that is represented in one entity or relationship of the other database.

Same terms could be contained in database parts with different granularity. For instance it could be fulfilled that the information that in \(D_1\) is represented in one entity is the same that in \(D_2\) is represented in two entities and one relationship.

\[
D_1: \quad \begin{array}{c}
E \\
\vdots \\
A_n
\end{array} \quad D_2: \quad \begin{array}{c}
E_{22} \\
R_2 \\
B_{21} \cdots B_{2m}
\end{array}
\]

We could find those similarity for instance by evaluating the attributes and names. The advantages of the including of such complex comparisons is that much more kinds of similarities can be found.

But it is more difficult to reuse information from such similar database parts because the adaption of design decisions is more complex. And, the search space
increases. The comparisons of only entities or relationships is $O(n \times m)$ whereby $n$ is the number of entities and relationships of one database and $m$ is the number of entities and relationships of the other. If we want to compare much more complex parts of databases we increase the search space.

For some application this can be interesting, therefore, we mentioned it here but do not use it in the further approach.

### 4.6 Building a graph

One typical problem that occur in reuse of information is that one entity or relationship of a database can have similarities to several terms of another database. We now have to determine which similar concept we want to choose for the derivation of further design suggestions. Therefore, we use a method that is known in graph theory: the graph matching. We built a bipartite graph that represents the estimated similarities.

**Definition:** A **bipartite weighted graph** $G$ consists of a non-empty vertex set $V(G)$ that can be divided into two disjoint subsets $S \cup T$ and a set of edges $E(G) \subseteq \{(s, t) | s \in S, t \in T\}$. All edges in $E(G)$ connect one vertex from $S$ with one vertex from $T$. To every edge in the graph a weight is related.

We build a bipartite graph that represents all estimated similarities in the following way:

1. We start with an empty graph.

2. Each determined similarity is added into the graph. For all determined similar entities and relationships $K_1 \in D_1$ and $K_2 \in D_2$ vertices are introduced ($K_1$ in $S$ and $K_2$ in $T$). We draw an edge between these vertices. The weight of the edge is the determined similarity measure.

Remark: For this application it is reasonable to introduce a limit (the same is done in [SCD97] where similarities that are under a limit are not taken into account). The best solution would be not to use a fixed limit, but one that we can adapt on the actual problem. It avoids to derive much "similar" information that is not similar and it simplifies the matching algorithm.

Now, we try to find a matching (a one-to-one relation) of the similar nodes with a maximal sum all all weights. For the database context it means that we search for similar parts of the databases $D'_1 \subseteq D_1$ and $D'_2 \subseteq D_2$ and $D'_1 \approx D'_2$ ($D'_1$ and $D'_2$ are similar).

We will demonstrate the suggested approach with an example. As a running example we will take two different databases that are designed for an university application (figure 2, figure 3).
Figure 2: Sample database university1

Figure 3: Sample database university2

Figure 4 shows the bipartite graph that originates by applying the suggested method onto the two sample databases. Now, we want to determine a matching of this graph. The basic idea is to
construct a cover \( c \) so that \( c \geq w(M) \) (that means the cover is greater or equal to the weight of the matching). We try to find the case where the cover is equal to the weight of the matching. Then we can derive the matching ([Jun94][Wes96]).
It is possible to determine the matching for every independent component of the graph separately. Figure 5 shows the components of the graph and the maximal matching for the similarity graph.

Figure 5: Maximal matching of the similarity graph

For a matching, we can determine the weight, in the example the weight $w(M) = 2.29$. This weight can be used to choose the most similar database to a given one.
The matching results to the following similar parts of the databases (figure 6, figure 7).

Figure 6: Database university1 that is similar to university2 ($D'_1$)

Figure 7: Database university2 that is similar to university1 ($D'_2$)
In figure 8 we represent the part of the database university1 that is not similar to a part of university2.

![Diagram](image)

Figure 8: Different part of the database university1 ($D_1 - D_1'$)

In the same way we encounter the parts of the database university2 that have no similar part in university1 (figure 9).

![Diagram](image)

Figure 9: Different parts of the database university2 ($D_2 - D_2'$)

5 Reuse

Now, we have an actual database and we have determined a similar database or similar part of a database. In this section we want to give an idea which information can be reused for the actual database and how it can be done. Reuse of information from available databases can support the design or redesign process of a database.

5.1 Structural design

First, we will demonstrate, which kinds of structural completions and expansions can be derived.

We assume, that we have determined similar databases $D_1'$ and $D_2'$, $D_1'$ is a part of an available database, $D_2'$ is a part of the actual database. If additional
structural information is available in \((D_1 - D'_1)\) then we can derive the following suggestions for \(D_2\).

- **Additional attributes**
  Are there similar entities or relationships \(E_1\) in \(D'_1\) and \(E_2\) in \(D'_2\) and \(E_1\) contains attributes that are not in \(E_2\) then we can suggest to add those attributes in \(E_2\), too.

- **Additional paths**
  Are there two similar entities or relationships \(E_1, R_1\) in \(D'_1\) and \(E_2, R_2\) in \(D'_2\) and these are connected by a path, then we can also try to add this path information in \(D_2\).

- **Adding of relationships**
  If there exists a relationship \(R_1\) in \(D_1 - D'_1\) and all belonging entities of \(R_1\) are in \(D'_1\) (that means similar entities exists in the actual database and a relation that is defined over these don’t exist) then the relationship \(R_1\) could be added in \(D_2\).

  In figure 8 there is shown such a case. The entities *student* and *professor* of university1 have similar entities in university2, the relationship *supervise* not. Therefore, we could suggest the relationship *supervise* as extension of the database university2.

- **Adding complex database parts**
  It is possible to suggest the adding of complex parts of the database \(D_1\) also in \(D_2\) if there exist similar entities \(E_1\) in \(D'_1\) and \(E_2\) in \(D'_2\).

  We need a node \(E_1\) in \(D'_1\) that has a direct link to nodes in \(D_1 - D'_1\).

  In our example, such a case occurs. In the database university2 exists parts that are not in university1 (figure 9). If the database university1 should be extended then we could for instance suggest to add the concepts *city*, *studies* and *lives* to extent the entity *student*.

  In that way we derive meaningful suggestions for an insight-out design.

### 5.2 Integrity constraints

We can derive integrity constraints from available similar databases.

- **functional dependencies**
  If we determined similar attribute sets \(A_{11}..A_{1n}\) in \(D'_1\) and \(A_{21}..A_{2n}\) in \(D'_2\) and a functional dependency \(A_{1i} \rightarrow A_{1j}, i, j \leq 1..n\) is valid, then the corresponding functional dependency in \(D'_2\) could also be valid.

- **keys**
  If similar entities \(E_1\) in \(D'_1\) and \(E_2\) in \(D'_2\) and similar attributes \(A_{11}..A_{1n}\) and \(A_{21}..A_{2n}\) are derived and if the attributes \(A_{11}..A_{1n}\) are a key of \(E_1\)
then \( A_{21}..A_{2n} \) could also be key of \( E_2 \).
In the same way candidate keys for relationships are determined: If we found similar relationships \( R_1 \approx R_2 \) (\( R_1 \) in \( D'_1 \) and \( R_2 \) in \( D'_2 \)) and similar attributes \( A_{11}..A_{1n} \approx A_{21}..A_{2n} \) and if the attributes \( A_{11}..A_{1n} \) are a key of \( R_1 \) then \( A_{21}..A_{2n} \) could also be key of \( R_2 \).

- **inclusion and exclusion dependencies**
  If we found two databases with similar entities or relationships \( E_{11}, E_{12} \) in \( D'_1 \) and \( E_{21}, E_{22} \) in \( D'_2 \) and similar attributes \( A_{11}..A_{1n} \approx A_{21}..A_{2n} \) and \( B_{11}..B_{1n} \approx B_{21}..B_{2n} \) and a cardinality constraint in \( D'_1 \) \( \big( A_{1i} \subseteq B_{1i} \text{ or } A_{1i} \parallel B_{1i} \big) \) then
  \( A_{2i} \subseteq B_{2i} \) respectively \( A_{2i} \parallel B_{2i} \) in \( D'_2 \) can be valid, too.

- **cardinality constraints**
  Are there two databases with similar entities and relationships \( E_{1}, R_{1} \) in \( D'_1 \) and \( E_{2}, R_{2} \) in \( D'_2 \) and the cardinality constraint \( \text{card}(R_1, E_1) \) is fulfilled in \( D'_1 \), we also can expect that \( \text{card}(R_2, E_2) \) in \( D'_2 \) has the same value.

5.3 Behavior information, sample data, optimisation, sample transactions

There are several further characteristics that can be reused in the same way. We want to give an idea which one it could be:

If behaviour information are specified in a formal way in available databases then we can reuse this specification in similar databases. Further, sample data, suggestions for optimisation, and sample transactions can be reused.

If we determine similar databases and try to reuse several information (for instance structural extensions) then we can assume that if the reuse of one characteristic was successful than the user accepts that the available database was similar. Therefore, it is promising to use this database also for the reuse of further characteristics (for instance integrity constraints or sample transactions).

But for all reusable characteristics we only can derive suggestions. These have to be revised in any way. The next section deals with the revise of the suggested solutions.
6 Revise

The method based on several heuristics therefore suggested design decisions have to be validated in any way.

Some design decisions can be checked without a user, for instance it can be checked if suggested integrity constraints are not in conflict and if they are conform with sample data.

Other suggestions have to be discussed with the user. If the reuse approach is used to derive suggestions for design tools and an user has to confirm the decisions than this demand is fulfilled.

For integrity constraints a method was demonstrated in [Kle97] that acquires candidates for integrity constraints by discussion of sample databases.

7 Retain

We have showed that the tasks retrieve, reuse and revise can be used to suggest design decisions for a database. To apply the method, we had to compare complete databases. In this section, we will demonstrate two methods how we can organise the databases in libraries to support the reuse process more efficiently.

7.1 Determining necessary and optional parts

In [SCD97] the similar parts of databases are stored for the further use.

In our approach we want to store for every field of application

which parts occur in every database

which parts are available in one or several databases

Because the similar parts of the databases shall be stored we have to discuss the following points with a user. A user must decide which of the synonym names of two similar databases is the most suitable one. Further, he has to confirm which attributes of an entity/relationship are relevant for a common case.

The parts of the databases which distinguish may be important for following applications, too.

But we have to extend the derived parts so that all databases are complete. That means if there exist a relationship in the database and not all belonging entities are in this database, then the entities have to be added. For the example:
The databases in figure 8 have to be completed, thereby the database in figure 10 is created.
In the same way the database in figure 9 is extended to the database in figure 11.

The side-effect of this action is that the database parts could be integrated over these entities.
If a user wants to design a database for one field of application then he/she can be supported as follows:

- The part that occurs in every database are discussed. If is is relevant to an actual database then we continue.
- The optional parts are discussed. If they are relevant then they are integrated into the actual database.

7.2 Deriving database modules for the reuse process

Another method to develop libraries is the following one.
We divide a database into modules and store these modules together with the field of application. How can we find these modules? Therefore, we exploit the following available information:

- path information
- integrity constraints (especially inclusion dependencies and foreign keys)
- design process (if known) - which entities and relationships are designed together
- similar names
- transactions
- layout (if it is automatically determined then it represents the path information of the model otherwise nodes that have a low distance can belong closely together

All these characteristics determine how closely entities and relationships belong together. A combination of these heuristic rules can be token to determine clusters in a database. These clusters are stored as units of the database.

The reuse process bases on these units.

8 Conclusion

The method shown in this article bases on several heuristics and an intuitive way of weighting these heuristics. Therefore, we cannot guarantee that it already delivers correct results. But the method is simple, easy to apply and promising for many design decisions. One of the advantages is that many different tools for database design could use the same method.

It is in the nature of reuse that the results of the method improve if more reusable databases are available.

An adaption onto an user and a field of application can also be achieved if databases of the same user or the same field of application are known and reusable.

The method was developed as a part of a tool for acquiring integrity constraints by discussing sample databases [Kle97]. In this context it was one method to derive probable candidates for keys, functional dependencies, analogue attributes, inclusion and exclusion dependencies.

9 Acknowledgement

I would like to thank Bernhard Thalheim because he encouraged me to write this paper.
References


[Dav93] Ted Davis: The Reuse Capability Model: A Basis for Improving an Organization’s Reuse Capability. in Advances in Software Engineering, Selected Papers from the Second International Workshop

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company Inc. 1995


[Wat95] Ian D. Watson (ed.): Progress in Case-Based Reasoning. Springer Verlag, Berlin, 1995

[Wat95a] Ian D. Watson: An Introduction to Case-Based Reasoning. in [Wat95]

Douglas B. West: *Introduction to Graph Theory.* Prentice Hall, 1996

Wolfgang Wilke, Ralph Bergmann: *Considering Decision Cost During Learning of Feature Weights.* EWCBR-96, European Conference on Case-Based Reasoning, 1996
Optimisation and Database Performance Tuning
A Computational Approach for Conceptual Database Optimization

Martin Steeg and Bernhard Thalheim
Cottbus Technical University, Computer Science Institute,
POBox 101344, D-03013 Cottbus, Germany
{steeg,thalheim}@informatik.tu-cottbus.de

Abstract. Traditional database design is not considering operational behavior. For performance reasons, tuning is a often required measure after conceptual, logical and physical design has been finished. This approach is difficult especially if integrity constraints require complex maintenance. Sometimes, integrity enforcement may be so complex that designers are not able to include it or that enforcement requires restructuring and denormalizing schemes. These restructuring actions are not conceptually represented what makes extending databases unmanageable. Based on estimations for operational behavior we present an approach to extend conceptual design such that a conceptual schema is found for optimal behavior of resulting applications.

1 Application of Database Design

Traditional database design is based on waterfall approaches. The designer starts with requirement analysis, designs the conceptual schema and translates it to the logical one. During these phases, semantical information is used to derive well-normalized type structures, i.e. database schemes without certain anomalies. Later, during physical design internal representations are derived. It is done in accordance to implementation techniques of the chosen database management system. This classical approach is represented in Figure 1(a).

However, response time of running applications may be inappropriate. Then, – at best – the conceptual schema is redesigned such that more efficient logical and internal representations can be derived, i.e. the process is started once again. Maybe now, processing requirements (planning, control, operations) are considered more appropriately. If this is not the case then the loop will be repeated constantly.

So, there are several reasons why processing information should be included into the earlier phases of database design:

– Low storage complexity is an objective of conceptual and logical design. Efficiency is the main database processing requirement. These goals are often in contradiction to the other.
– Normalization is the typical approach to minimize storage complexities and operation anomalies. But, normalization algorithms may derive several schemes from the same attribute and functional dependency set. (See [Ull82])
– Well-designed schemes which are piped from conceptual to logical design and subsequently are implemented by a database management system often imply operational bottlenecks. For different DBMSs different tuning principles are appropriate. ([Sha92])
– Database structures providing the most efficient implementation often cannot be derived directly from the given. Therefore, external views may not be supported by the ‘tuned’ internal, logical or conceptual schemes.
Therefore, modern database design approaches advice to include processing information as well. E.g., [EN89] argues why processing requirements should be always considered at early design steps: conceptual and logical design often generates a database schema that cannot easily be modified once the database is implemented.

There are certain foundations for transformation from conceptual to logical schemes and their implementation under a certain DBMS. Mostly, the mapping to an advantageous internal representation is based on the according operational support. E.g., retrieval cost can be minimized if we use ‘join-tables’ clustering data of types related one-to-many [CBC93], and different data organizations have different benefits and drawbacks [Wie87, KS91]. Processing requirements can be considered as soon as they are available:

- Approximated tuple numbers or frequencies of sophisticated transactions can be acquired.
- There exists a general mechanism for computing integrity maintaining procedures from data schemes. ([ST94b, SST94])

Figure 1(b) shows how conceptual design is extended by inclusion of processing requirements. With the help of parametric transformation heuristics internal data structures of hierarchical, network, relational or object-oriented type are derived. According to the provided physical data organization (such as Ring, Isam, Btree, dHashing, ...) operational complexity is evaluated. By attachment of preceeding types and integrity constraints the conceptual schema is valuated on base of internal complexities. Therefore, bottlenecks can be detected and subsequently, they are eliminated in the conceptual schema. So, it requires neither to design logical schemes nor to respect the logical or internal representation by the conceptual designer. This supports a natural entry to database design.

However, the complete set of processing requirements cannot be included into conceptual design. This is an approach to include a subset of them. The underlying Higher-order Entity-Relationship Model (HERM) [Tha89, Tha91] is the base of a toolbox for Rapid Application and Database Design (RADD). The tool discussed here consists of two mayor components: first, a behavior estimator and second, a schema optimizer. So, the given conceptual schema
can be evaluated and optimized in accordance to the intended target context. In addition, recommendations can be generated what (type of) DBMS to use or what data organization to choose if the used DBMS supports different ones.

The paper starts with discussing integrity enforcement in Section 2. Section 3 presents a rule-based approach for computation of behavior properties as well as operation complexities. Section 4 presents a scenario optimizing the given conceptual schema. Section 5 figures the automatical restructuring of bad-behaving schemes, and Section 6 finishes with a retrospection to the taken approach, a palette of open problems, and future work.

2 Integrity Enforcement

Databases can be specified by their structures $\text{Struc}$, their operations $\text{Ops}$ and their integrity constraints $\Sigma$, i.e. the database schema is given by the triple $\mathcal{S} = (\text{Struc}, \text{Ops}, \Sigma)$. Often we omit the operations assuming that operations are given generically whenever the structure of the database schema is given, i.e. $\mathcal{S}^* = (\text{Struc}, \Sigma)$. For instance, the definition of relational schemes by their attribute sets implies directly the corresponding manipulation operations of the relation type (select, insert, delete, update), i.e. we are given the pair $(\text{Struc}, \Sigma)$ and assume that $\text{Ops}$ is to be generated on the basis of $\text{Struc}$ ($\text{Ops} = \text{GenericOps}(\text{Struc})$).

Constraints can be classified into static, transition and general dynamic constraints describing correct states, state transitions or state sequences respectively. It is commonly assumed that most of integrity constraints can be defined statically. Consistency is a crucial property of database application systems. There are two approaches to consistency enforcement:

1. Each database instance is checked whether it is correct. This requires constraint checking after each operation or after a certain number of operations.
2. The database instance is assumed to be correct before performing the operation. Then correctness is to be checked according to the effect of the operation.

The first approach is computational infeasible. The second approach uses the correctness of the database instance and is based on integrity enforcement. Then the consistency problem is to guarantee that each specified operation $o \in \text{Ops}$ will never violate any constraint $\alpha \in \Sigma$. In conventional database systems, consistency is preserved either by forbidding operations that violate integrity constraints or by rolling back transactions that produce inconsistent database states. A third way to integrity enforcement is that of postponing faulty transactions by introducing repairing actions which compensate incorrect states. The complexity of correctness inspection depends on functionality of the database management system. Normally, database management systems are limited in their support for consistency. It is argued that DBMS only support key and domain constraints. Several approaches have been developed to consistency support:

1. Normalization aims at the derivation of a new structure of the database, a set of operations obtained by applying normalization transformation $T$ to the operations, and a corresponding set of integrity constraints such that $(\text{Struc}' = T(\text{Struc}), \text{Ops}' = T(\text{Struc}, \text{Ops}), \Sigma' = T(\text{Struc}, \Sigma))$ is equivalent to the schema $(\text{Struc}, \text{Ops}, \Sigma)$ and integrity can be maintained for the new schema.
2. Integrity enforcement aims at the derivation of a new set $\text{Ops}'$ of operations such that $(\text{Struc}, \text{Ops}', \Sigma)$ is equivalent to the schema $(\text{Struc}, \text{Ops}, \Sigma)$ and each operation is consistent.
3. **Integrity restructuring** aims at the derivation of a new set $\Sigma'$ of integrity constraints such that $(\text{Struc}, \text{Ops}, \Sigma')$ is equivalent to the schema $(\text{Struc}, \text{Ops}, \Sigma)$ and a better support to consistency is available in the current DBMS.

The normalization approach is well-known and can be applied to ‘normal’ databases. The integrity restructuring approach is not applicable whenever the constraint set is using constraints like inclusion and functional dependencies together. Different approaches to integrity enforcement and database maintenance have been proposed in literature.

1. The **transaction approach** [Elm92, GR93] requires that whenever the application of an operation to a correct database leads to an incorrect database the operation is rolled back. This approach is inefficient for large constraint sets. **Repairing actions** [CW90, FPT92] can be used for support of rollback. The **specialized transaction approach** of [LST87, Dec87] restricts the set of integrity constraints to be checked in each database state. The **frame transaction approach** [KSS87, Das92] restricts the database to be checked after application of an operation to a subset of the database which is examined on consistency by a subset of $\Sigma$.

2. The **specialization approach** [STSW92, ST93] extends the **trigger approach** [CW90, ZH90]. It is based on the assumption that for any operation applicable to the database schema a specialization of this operation can be developed. This specialization of the operation can be applied to any correct database and the result of this operation is again a correct database.

The first approach has several essential drawbacks. The classical rollback approach is completely ‘pessimistic’. Once an operation or transaction violates one of the integrity constraints the operation cannot be used in general. Integrity enforcement is considered one of the major application fields of rule triggering systems. This approach requires confluence and consistency. The rule triggering approach is limited as well. Analyzing critical paths [ST94a, SST94] it can be shown that only strong hierarchical schemes permit this approach. This case should be considered as the exceptional case for practical applications. Further, it has been shown [SST94] that specialization is a necessary condition which is not derivable from the first two.

The second approach is computationally feasible. As shown in [ST94] the efficient construction of consistent update operations can be based on linguistic reflection. Type-safe linguistic reflection came up with the development of the ADABTPL language which laid a primary interest on the development of correct database transactions [SS90, SS91]. This paper extends this approach to physical schemes. Based on an estimation of the operational complexity bottlenecks in the design can be detected and remedied. Generic update operations are analyzed. **Insert**, **delete** and **update** are generic in sense that they are applicable to each type of the database schema. Relational database management systems as well as most of the important object-oriented DBMSs (Montage, Itasca, O2, ... ) provide such operations. In order to build generator macros for generic update operations we can follow the constructive proof of their existence in [ST93]. This approach to integrity enforcement can be extended to more general integrity constraints as well.
3 A Rule-based Approach to Integrity Maintenance

Recent DBMS implementations offer a range of opportunities for rule triggered integrity corrections (Rule Triggering Systems). The RTS approach may be applied for todays object-oriented and relational DBMSs which are supporting triggers and/or stored procedures. Triggers can be specified in order to correct immediately incorrect database states – or else to raise errors. This allows to implement database structures close to the structure of the underlying data model.

3.1 Type structures, constraints and integrity maintenance

Concepts of conceptual and logical design can be used to describe influence on operational behavior in the usual way:

relationship type $R_i = \{E_1, \ldots, E_m, R_1, \ldots, R_n\}, \{a_1, \ldots, a_o\}, \{k_1, \ldots, k_p\}$:

The referred items $E_1 : x_{k_1}^1, \ldots, R_n : x_{k_n}^n$ must coincide items in the component sets $E_1, \ldots, R_n$ respectively. So, insert into $R_i$ requires insert into (existence in) the component sets, delete from any component set can require delete from $R_i$. Dependent set types of the logical or internal data model, i.e. types with references to others may be interpreted as relationship types too.

functional dependency $R_i^t : A_j \rightarrow A_k$:

Whenever the values of attribute $A_j$ in relation $R_i$ coincide for two tuples, the values of attribute $A_k$ must also be equal:

\[
R_i^t(\ldots, A_j : x_j, \ldots, A_k : x_k^1, \ldots) \land \\
R_i^t(\ldots, A_j : x_j, \ldots, A_k : x_k^2, \ldots) \Rightarrow x_k^1 = x_k^2 .
\]

This extends in the usual way to sets of attributes $X, Y$.

inclusion dependency $R_i^t[A] \subseteq R_j^t[B]$:

The values of attribute $A$ in relation $R_i^t$ always form a subset of the values of attribute $B$ in $R_j^t$:

\[
R_i^t(\ldots, A : x, \ldots) \Rightarrow R_j^t(\ldots, B : x, \ldots) .
\]

Inclusion dependencies generalise the relation between a relationship and her component.

exclusion dependency $R_i^t[A] \parallel R_j^t[B]$:

The sets values of attribute $A$ in relation $R_i^t$ and of attribute $B$ in $R_j^t$ are always disjoint:

\[
R_i^t(\ldots, A : x, \ldots) \land R_j^t(\ldots, B : y, \ldots) \Rightarrow x \neq y .
\]

So, insertion of an item possibly requires deletion of conflicting items in the other set.

Example:

Assume, there is a constraint set for types $R_1 = \{A, C\}$ und $R_2 = \{B, D\}$:

\[
\begin{align*}
R_1^t[A] & \subseteq R_2^t[B] & \text{(inclusion dependence)} \\
R_2^t : D & \rightarrow B & \text{(functional dependence)} \\
R_1^t[C]\parallel R_2^t[D] & \text{(exclusion dependence)}
\end{align*}
\]
So, the following rules can be derived directly:

1. ON INSERT \( R_1(a,c) \): \( \text{IF } a \not\in R_2^t[B] \) THEN INSERT \( R_2(a,?) \)
2. ON DELETE \( R_2(b,d) \): \( \text{IF } b \in R_1^t[A] \land b \not\in R_2^t[B] \) THEN DELETE \( R_1(b,?) \)
3. ON INSERT \( R_2(b,d) \): \( \text{IF } (b',d) \in R_2^t \land b' \neq b \) THEN FAIL
4. ON INSERT \( R_1(a,c) \): \( \text{IF } c \in R_2^t[D] \) THEN DELETE \( R_2(?,c) \)
5. ON INSERT \( R_2(b,d) \): \( \text{IF } d \in R_1^t[C] \) THEN DELETE \( R_2(?,d) \)

Rule (1) and (2) are derived from the inclusion dependence, (3) from the functional dependence, and (4,5) from the exclusion dependence. If we apply INSERT \( R_1(abba,coco) \) to Fig. 2 the database in Fig. 3 results.

\[
\begin{align*}
R_1^t &= \{(atta,baba)\} \quad \text{and} \quad R_1^{t'} = \{(atta,baba)\} \quad \text{and} \\
R_2^t &= \{(atta,caca),(abba,coco),(agga,cucu)\} \quad \text{and} \quad R_2^{t'} = \{(atta,caca),(agga,cucu)\}
\end{align*}
\]

**Fig. 2.** Initial database instance  \hspace{1cm} **Fig. 3.** State after insertion

Hence, INSERT \( R_1(abba,coco) \) results in DELETE \( R_2(abba,coco) \). This is far away from the desired result. Such inconsistent behavior can be determined using the greatest consistent specialization (GCS) algorithm of Schewe and Thalheim [ST94b]. The GCS algorithm evaluates the transaction content based on the integrity constraint set. It finally generates the minimal operation sequence which causes the resulting database state as maximal quasi-consistent specialization. Unfortunately, – as well as the sequence of rule-triggered operation calls – the algorithm may not terminate if the set of integrity constraints is abitrarily constructed from IDs and FDs.

Especially, if the type and constraint set is not strong hierarchical then heavily-terminating database operations can result. (Fig. 4)

---

**Fig. 4.** Non-terminating insert operation  \hspace{1cm} **Fig. 5.** Restricted database population type
3.2 Database populations and operation dependencies

For the situation in Fig. 4, interesting questions may be asked – like: When does such a situation appear? or What is the frequency/probability for her occurrence? In order to evaluate such behavior the current approach breaks operation cycles. So, the evaluation as well as the valuation of properties/complexities is restricted to acyclic database populations (Fig. 5). From sight of subsumption-free operations by use of conventional DBMSs, or non-terminating operations at application of the RTS approach this seems to be a reliable restriction. Then, for consideration of operation dependencies we can refer to cardinality constraints:

A participation constraint \( \text{card}(X,Y)=(m,n) \) specifies that each item in \( Y^t \) must be related to at least \( m \) and at most \( n \) items in \( X^t \). Consequently, cardinality constraints may be used to determine the probability by which an operation \( \text{op}_X \) requires – or otherwise triggers – \( \text{op}_Y \).

For instance, if we refer to the usual implementation by a modern DBMS then we could give the number of necessary repairing actions in the following way:

1. \( \text{card}(X,Y)=(m,\cdot) \).
   - Each time a new instance \( y \) is added to the set \( Y^t \) and the lower bound \( m \) is violated (it means, the number of \( X \)-items related to \( y \) is lower than \( m \)) invoke insertion of \( m-|x \in X^t_{\{x|y^\cdot=y\}}| \) items into \( X^t \).
   - Each time for some but not all items in \( X^t \) which are referenced by some \( y \in Y^t \) deletion is performed and for \( y \) the cardinality falls short of \( m \) invoke insertion of concerning \( X \)-items for compensation. If \( m > 0 \) and deletion of all \( X \)-items related to \( y \) is performed invoke deletion of \( y \).

2. \( \text{card}(X,Y)=(\cdot,n) \).
   - Each time on insertion into \( X^t \) the upper bound \( n \) is violated for some \( y \in Y^t \) invoke deletion of \( |x \in X^t_{\{x|y^\cdot=y\}}| - n \) \( X \)-items for compensation.

3. For estimation the other constraint types could be represented by cardinality constraints respectively: relationship type to his component type by (1,1), FD by (0,1), ID by (1,\cdot), and ED by (0,0).

Now, the probability by which an operation requires another operation is determined as follows. Let \( \rho_1, \rho_2 \) be generic update operations \((\text{INSERT,DELETE,UPDATE})\) und \( T_1, T_2 \) be parameter types of \( \rho_1, \rho_2 \) , respectively. If there is a \( \text{card}(T_2,T_1)=(m,n) \) then:

1. \( \beta_{\rho_1(T_1)\rho_2(T_2)} = \frac{\text{avg}(m,n)\sqrt{m}}{n} \cdot \frac{\sqrt{\text{tuple}\#(T_1)}}{\sqrt{\text{tuple}\#(T_1)+\sqrt{\text{tuple}\#(T_2)}}} \) for \( \rho_1 \in \{\text{INSERT,DELETE}\} \), and

2. \( \beta_{\rho_1(T_1)\rho_2(T_2)} = \text{DefUpdRelCost} \cdot \frac{\text{avg}(m,n)\sqrt{m}}{n} \cdot \frac{\sqrt{\text{tuple}\#(T_1)}}{\sqrt{\text{tuple}\#(T_1)+\sqrt{\text{tuple}\#(T_2)}}} \) for \( \rho_1 = \text{UPDATE} \).

\( \text{DefUpdRelCost} \) is dependent on the kind of update (e.g. destructive) and upon the frequency of key-value updates or data movements on the secondary storage.
We set \( m = 0 \) if the lower bound is not given and \( n = \max(m \times 3, 20) \) if the upper bound is unspecified. The cards for EDs are handled as \((1, 1)\) (\((0, 0)\) isn’t appropriate and would cause a division error). For operations on embedded set types, e.g. \( T_1^s \) of \( T_1 \), we need to choose some special for \( \beta_{\rho_1(T_1)\rho_2(T_2)} \) if \( T_1^s \) and \( T_2 \) were related previously. Hence, the term is multiplied by:

\[
\frac{\text{tuple}\#(T_1^s)}{\text{tuple}\#(T_1)}.
\]

(\text{tuple}\# indicates the expected tuple number; she can be acquired from the data modeler)

Note, that the assumptions are purely heuristic. They may be verified by implementing the database schemes, inputing test data, and running the referred transactions.

### 3.3 Determining operation complexities by linear equation systems

In general, evaluating operation contents may be inefficient as soon as many different cases have to be looked for – especially on large occurrence sets. Therefore, we restrict us to a mapping of operation calls to representing terms. Then, these terms – embedded in transaction graphs – can be used to implement an efficient replacement algorithm. The transaction cost terms are mapped to equations. The solution of the equation system renders the merits and drawbacks of the modeled data context.

First, if we have mutual or cyclic data dependencies (e.g. IDs) then the database may ever remain the empty database (unless nulls are allowed), i.e.: invoked insertion operations are subsumption-free. For modern DBMSs with rule triggering abilities problems may occur when read-locks are required to (partially) write-locked data sets: transaction managers often need much time to detect that a required lock is always held by the same transaction. Enforcement of data model constraints implies that some operation \( Op_1 \) w.r.t. type \( T_1 \) may trigger \( Op_2 \) w.r.t. \( T_2 \) if and only if the database system implements a related constraint, i.e.: some constraint specifies that items of \( T_1 \) are dependent on the (non-) existence of \( T_2 \)-items. So, it seems to be reliable to group a relationship type \( T_1 \) with his component type \( T_2 \) if the data schema specifies a cardinality constraint \( \text{card}(T_1, T_2) = (m, n) \) with \( m \geq 1 \):

**Hierarchical model.** \( \text{card}(T_1, T_2) = (1, 1) \) indicates that \( T_1 \) and \( T_2 \) should be grouped to some new \emph{union}-type in the target schema.

**Network model.** \( \text{card}(T_1, T_2) = (m, n) \) with \( m \geq 1 \) and \( n \leq \text{some upper bound} \) indicates that \( T_1 \) and \( T_2 \) should be grouped to a new type \( T_2' \) containing a repeating group of \( T_1 \).

**Relational model.** \( \text{card}(T_1, T_2) = (m, n) \) with \( m \geq 1 \) and \( n \times \text{length } T_1 \) \( \leq \text{some upper bound} \) proposes that \( T_1 \) and \( T_2 \) should be grouped to some new type \( T_2' \) containing an attribute group of \( T_1 \) which is repeated \( n \)-times.

**Object-oriented model(s).** \( \text{card}(T_1, T_2) = (m, n) \) with \( m \geq 1 \) indicates that \( T_1 \) and \( T_2 \) should be grouped to a new type \( T_2' \) containing a set-typed attribute of \( T_1 \).

These transformation heuristics can be parametrized with explicit criteria indicating whether to group and split types or whether to enforce and restrict integrity. So, an experienced database designer is free to introduce his own concepts – either for the global transformation frame or else for specific types and integrity constraints of his schema.

Then, the behavior estimator infers transaction graphs from the transformed data schema. This process is performed continuously until an operation call of the same type is detected. For reason of termination, then the replacement algorithm is interrupted. So, inconsistencies can be detected – e.g., an insert operation that possibly triggers delete from the same set – like the operation in section 3.1.
The nodes of the so generated transaction graphs contain the cost of the base operation and the probability for the necessary integrity maintaining action. Then, an equation system is built up from the transaction graphs; i.e.: the root node (representing the invoked operation) and his children of first level is mapped to a cost equation. So, the operational complexities can be evaluated by solving the equation system.

The re-mapping to terms of the conceptual schema (to the conceptual operation cost) is enabled by reference of the transformed types and constraints to the conceptual ones. E.g., the cost for a conceptual insert operation that is mapped to a physical update is scheduled by the related operation nodes of the physical and conceptual level respectively. So, if a conceptual type is transformed to a set-typed attribute on the implementation level then the evaluation of according conceptual transaction cost is done appropriately.

3.4 Underlying cost model

Operational complexities are quite different for several operation types. If we follow the approach of [CBC93] then we can use the following frame of cost primitives:

1. locate data item in $R^t \ l_{R^t}$
2. store data item into $R^t \ s_{R^t}$
3. delete data item from $R^t \ d_{R^t}$ (set unused marker)
4. modify data item in $R^t \ m_{R^t}$
5. fetch data item of $R^t \ f_{R^t}$
6. (a) reorganize the related indices of $R^t \ r_{i_{R^t}}$
   (b) reorganize the hash buckets of $R^t \ r_{b_{R^t}}$
7. reorganize the whole storage of $R^t \ r_{s_{R^t}}$

These terms are parametrized by some term $p_r$ according to the type of data organization:

1. indirect
   $p_r = n_d$ (pointer semantic)
2. Isam
   $p_r = n_i * \log_{\text{fanout}}(R^t)$
3. Btree
   $p_r = n_b * \log_{\text{blocksize/keylen}(R)}(R^t)$
4. Hashing
   $p_r = n_h' * (1 + \rho(\text{collision})) * \frac{n_h'' \cdot \text{keylen}(R)}{n_h'''}$
   $\rho(\text{collision})$ itself is dependent upon the chosen bucket size $bc$, the frequency of modification requirements $fm$ and some uniformity $uf$ of 'key' data according to the (maximal) hash prefix: $\rho(\text{collision}) = \frac{fm \cdot uf}{bc}$
   ($fm$ depends on the occurrence set and the operation type - on RETRIEVE it is 0 - , $uf$ depends on the tuple number $uf = n_h''' \cdot \text{tuple}\#(R^t)$, and $bc$ is set to some reliable bucket size such as $bc = 20$)

where $n_d, n_i, n_b, n_h', n_h'', n_h''', fm$ and $bc$ are assumed to appropriate terms. If a new DBMS offers new behavior properties then they may be changed accordingly. Note, that their size is rather unimportant whereas the proportion between them is much more important.

We provide a generic cost model applicable to Heap, Ring, Isam, Btree, dBtree, Hashing, and dHashing data organization. The cost model uses default values for the related data model type and can be parametrised according to the chosen DBMS. In addition, the combination of different data organization types is possible, if the designer specifies different organizations for types $T_1, T_2$ and they are grouped by the transformation process.
For instance, we evaluate the operational complexities for dynamic hashing organized databases in the following way:

1. Cost of insert$_{R^t}(i) = l_{R^t} + \rho_{ib} \ast s_{R^t} + rb_{R^t} + \rho_{is} \ast rs_{R^t}$
2. Cost of delete$_{R^t}(i) = l_{R^t} + d_{R^t} + \rho_{db} \ast rb_{R^t}$
3. Cost of update$_{R^t}(i) =$
   (a) $l_{R^t} + m_{R^t} + \rho_{ub_1} \ast rb_{R^t}$ (non-destructive)
   (b) $l_{R^t} + m_{R^t} + \rho_{ub_2} \ast rb_{R^t} + \rho_{us} \ast rs_{R^t}$ (destructive)
4. Cost of retrieve$_{R^t}(i) = l_{R^t} + f_{R^t}$

(where $\rho_{ib}, \rho_{is}, \rho_{db}, \rho_{ub_1}, \rho_{ub_2}$, and $\rho_{us}$ represent probabilities depending on the tuple numbers)

4 Application Scenario

The base model of our conceptual approach is the *Higher-order Entity-Relationship Model* [Tha89, Tha91]. HERM may be used to model structure, semantic and operations of databases. It extends the traditional entity-relationship approach by the type constructors 'cluster' (refering to union types) and 'higher-order relationship' (relationship types may have relationship types as components). The semantic of a data schema is founded on the constraint types in section 3.1. Operations are looked with regard to the generic update operations, insert, delete and update.

4.1 A given data schema

Figure 6 shows an example HERM data scheme.

For semantical completeness, we can attach the following cardinality constraints:

- $\text{card}(Fnames.Fname, Name) = (1,3)$. The Name of each Employee contains at least one Firstname and at most three Firstnames.
- $\text{card}(Locations.Location, Department) = (1,\infty)$. Each Department is placed on at least one Location.

The behavior estimator is supported by an user interface for acquisition of necessary tuple numbers, e.g.: Employee(760), Department(4), works_for(870), Project(15), and Supervision(570). Those of works_on, manages, and controls can be inferred from these.

This delivers the estimation of operational complexities presented by Table 1.

Now, there is the following perception: the final database user will be confronted with unreliable response times on some operations. So, under certain circumstances

- insertion of works_for, works_on and Supervision,
- deletion of Employee, or
- update of Employee, works_for, works_for and Supervision

may be possible bottlenecks.

Whether this operational behavior is really a bottleneck of the scheme depends upon the frequency and priority of required operations. More specific:

- if the insertion is a often required operation for works_for, works_on and Supervision then the insert operation creates a crucial bottleneck;
Fig. 6. Company Data Schema

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Update</th>
<th>Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>15.13</td>
<td>22.35</td>
<td>13.06</td>
<td>3.63</td>
</tr>
<tr>
<td>Department</td>
<td>20.27</td>
<td>18.39</td>
<td>13.67</td>
<td>2.24</td>
</tr>
<tr>
<td>Project</td>
<td>14.45</td>
<td>13.15</td>
<td>8.11</td>
<td>2.02</td>
</tr>
<tr>
<td>works_for</td>
<td>35.4</td>
<td>9.12</td>
<td>11.84</td>
<td>3.75</td>
</tr>
<tr>
<td>manages</td>
<td>18.32</td>
<td>11.49</td>
<td>8.6</td>
<td>1.9</td>
</tr>
<tr>
<td>works_on</td>
<td>26.05</td>
<td>9.12</td>
<td>9.78</td>
<td>4.08</td>
</tr>
<tr>
<td>controls</td>
<td>30.79</td>
<td>15.45</td>
<td>13.77</td>
<td>2.39</td>
</tr>
<tr>
<td>Supervision</td>
<td>23.13</td>
<td>5.31</td>
<td>11.63</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Table 1. Operational complexities implied by the given Company Data Schema
– if it is necessary to delete an Employee frequently then the delete operation has a higher complexity;
– the update operation is inefficient for the entity type Employee and the relationship types works_for, works_on, and Supervision
– the retrieval operation w.r.t. the relationship types works_for and works_on may be looked as relatively complex.

Therefore, we can derive that

1. if we need frequently the above mentioned operations then
   – the types works_on and works_for could be grouped if optional attributes are allowed
   – the type manages is a subtype of works_for identifying the special role of department managers
   – the relationship type Supervision should be considered as complex in behavior (therefore, we could reason whether this type can be represented as an attribute)

2. if the entity occurrence sets are not changed frequently then above already discussed relationship types should be grouped in the case that their operations get a very high frequency and priority

3. if the insertion of Supervision is rarely used we could remain with the original type.

More general restructuring suggestions are discussed below. Rules for optimization can be developed in accordance to tuning techniques of the chosen DBMS.

4.2 Consequences for conceptual optimization

Although the cost model considers storage complexities we can presuppose (w.r.t. decreasing hardware prices) that consume of storage space is of less interest. In order to the aim of minimizing operational complexities we can propose the following actions to the conceptual database designer:

1. The high complexities for deletion and update of Employee mainly result from the (operational) dependency cycle on path Employee-works_for-Department-manages. Consequently, the designer is hinted to group the relationship types. For reasons of preserving the given semantics, he needs to add an attribute – e.g. IsManager – marking the special role of department managers.
So, the cardinality constraint:

\[
\text{card}(\text{works}_\text{for}'(\text{IsManager}=\text{true}).\text{Department}, \text{works}_\text{for}'\cdot\text{Department}) = (1, 1)
\]

is additionally given. It indicates that the new relationship type \text{works}_\text{for}' sets the identifying attribute for exactly one tuple of each department to true. The attribute \text{StartDate} must be transferred to the new type as an optional one (e.g., \text{MgrStartDate}'\).

2. For the high complexities on insertion of \text{works}_\text{for} and \text{works}_\text{on}, we propose the user to cluster these relationship types. The attributes (i.e. \text{IsManager} and \text{MgrStartDate}) of \text{works}_\text{for}' are moved to the new relationship type \text{works}_\text{on}'. As above, this requires that attributes of the new type must be optional.

3. To drop the high complexities on \text{Supervision}, we can suggest to substitute the relationship type by some new attribute added to type \text{Employee} (e.g., \text{supervised by}'\).

Then, the inclusion dependency:

\[
\text{Employee}'\cdot\text{supervised by} \subseteq \text{Employee}'
\]

must be implemented by a referential constraint.

### 4.3 Result after new evaluation

After performing all suggested modifications the designer is confronted with some surprising result.

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Update</th>
<th>Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>23.75</td>
<td>16.23</td>
<td>17.05</td>
<td>3.76</td>
</tr>
<tr>
<td>Department</td>
<td>10.11</td>
<td>13.87</td>
<td>12.79</td>
<td>2.24</td>
</tr>
<tr>
<td>Project</td>
<td>16.12</td>
<td>18.15</td>
<td>11.05</td>
<td>2.02</td>
</tr>
<tr>
<td>controls</td>
<td>28.63</td>
<td>23.7</td>
<td>15.49</td>
<td>2.39</td>
</tr>
<tr>
<td>works_on</td>
<td>40.25</td>
<td>18.5</td>
<td>23.92</td>
<td>3.88</td>
</tr>
</tbody>
</table>

**Table 2.** Estimation of the modified Company Data Schema

Although some of the trade-offs decreased, in general more worse behavior is implied by the new scheme. Mainly, this bases on the insert operation for types \text{Employee} and \text{works}_\text{on} as well as deletion and update for \text{Department}. It can be reasoned by

- the higher trade-off for basic operations on \text{Employee} which was added the attribute type \text{supervised by}'
- the high priority by which the attribute \text{supervised by}' implies that the insertion and update operation on \text{Employee} require integrity test or repairing actions
- the strong dependence of the update and deletion operation for type \text{Department} on the occurrence set of \text{Employee}.

Whether or not the designer accepts these drawbacks depends upon the context. We think that in this context the high complexities for the operation types on \text{Department} can be neglected because there are only 4 instances. Related operations seem to appear rather infrequently. In contrast, the related insertion operations should become a crucial bottlenecks because there
are large numbers of Employee and works_on tuples. So, we take the attribute supervised_by out of the entity type and picture again the relationship type Supervising. (Figure 7.)

![Diagram of the conceptual data schema](image)

**Fig. 7.** Optimal Conceptual Data Schema

### 4.4 Better behavior

The following evaluation results from the new scheme and presents operation types of lower complexity. So, the data scheme of Figure 7 probably presents a reliable basis in order to implement a well-behaving database system.

### 5 Automatic Optimization

#### 5.1 The implementation concept

The tool 'Reviewer' is implemented in Standard-ML as a collection of non-parametric and parametric abstract data types. For estimation, the conceptual scheme is mapped to SML terms where we use a ‘database’ of persistent data:
Table 3. Reliable Trade-offs for the Optimal Data Schema

<table>
<thead>
<tr>
<th>Type</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Update</th>
<th>Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>22.59</td>
<td>22.06</td>
<td>20.41</td>
<td>3.63</td>
</tr>
<tr>
<td>Department</td>
<td>10.2</td>
<td>14.12</td>
<td>12.88</td>
<td>2.24</td>
</tr>
<tr>
<td>Project</td>
<td>16.15</td>
<td>18.54</td>
<td>11.17</td>
<td>2.02</td>
</tr>
<tr>
<td>controls</td>
<td>26.73</td>
<td>21.86</td>
<td>13.54</td>
<td>2.39</td>
</tr>
<tr>
<td>Supervision</td>
<td>28.11</td>
<td>5.66</td>
<td>14.44</td>
<td>3.56</td>
</tr>
<tr>
<td>works_on</td>
<td>32.56</td>
<td>17.28</td>
<td>20.25</td>
<td>3.92</td>
</tr>
</tbody>
</table>

1. A set of references to the modeled attributes.
   (‘reference’ in SML is different from the notion ‘address’ or ‘pointer’ of procedural languages; it is a value similar to an ‘object identifier’.)
2. A set of references to the entity, relationship and cluster types.
   Their attribute and key schemes are lists resp. sets of the attribute references; relationships and clusters contain a reference set for components, respectively; all type constructors store their history by the reference to the preceding types.
3. A set of integrity constraints of the above mentioned types.
   Similar to the type constructors their history is stored by the reference of the predecessor.

This provides heuristic transformation, evaluation of behavior properties, and appropriate remapping to terms of the conceptual scheme. So, the user is directed to benefits and drawbacks of his design: when the user clicks on a cost term the content of the according transaction (subtransactions) will be displayed.

5.2 Modification of the scheme

Su [Su85] always presented a strategy how to estimate trade-off based on statical and dynamical aspects of types and their connections: frequencies for access and combining instances with others were added to the types and arrows of the conceptual scheme. More recently, Patrick van Bommel [Bom94] has introduced ‘fitness’ functions to determine trade-off. Starting with a base scheme of the NIAM model he tests several variants of grouping the given type set to a new type set. So, he can determine local minima for cost of the transformed type scheme and choose the best one.

Similar to this manner, we will realize the optimization: the attribute set is used to modify and generate distinct data schemes for the same information context. (Figure 8)

By regard to the operational fitness of the (sub)schemes we can determine which part behaves well and which part behaves worse. Then, we can incrementally generate the optimal conceptual scheme by selection of those parts implying the minimal complexities – in accordance to the operation types which are frequently required.

6 Conclusion

There is a large research on physical models and techniques in order to improve operational behavior. However, most techniques are specific for a certain DBMS. The list of articles solving
in general tuning problems is smaller. The techniques discussed in [Sha92] cover most of this research. In addition, cost optimization proposals have already been stated for different data models. E.g., CODASYL may be understood as one of these efforts – or as a result of these efforts. Discussions to find a general strategy which are applicable to items of the source data model are rather rare.

The aim of tuning is the generation of better operational behavior. Conceptual design approaches with their complete and explicite semantic representation often contradict reliable behavior of data intensive applications. So, at least at time of physical implementation bottlenecks must be remedied. Required tuning techniques are often based on undesirable actions such as denormalization. In the current approach, an essential part of structural optimization can be already performed at creation time of the conceptual scheme.

The paper discusses transformation principles, operational dependencies and trade-off as well as restructuring/rearrangement suggestions. These are important for optimization of database performance. The approach has been implemented and is used in the design system RADD (Rapid Application and Database Design). The designer can make decisions which constraint needs to be enforced and which enforcement is not necessary. In this case, the partial enforcement gives the designer more freedom in choosing the best implementation. Summarizing, the presented tool generates better design decisions and displays it for further discussion. We will extend the design system to include also complex transactions and parametric queries into the specification framework of application scenarios.

References


A Computational Approach to Conceptual Database Optimization

Bernhard Thalheim, Martin Steeg

1 Database Design Approaches

Traditional database design is based on waterfall approaches. The designer starts with requirement analysis, designs the conceptual schema and translates it to the logical one. During these phases, semantical information is used to derive well-normalized type structures, i.e. database schemes without certain anomalies. Later, during physical design internal representations are derived. It is done in accordance to implementation techniques of the chosen database management system.

However, response time of running applications may be unappropriate. Then, – at best – the conceptual schema is redesigned such that more efficient logical and internal representations can be derived, i.e. the process is started once again. Maybe now, processing requirements (planning, control, operations) are considered more appropriately. If this is not the case then the loop will be repeated continously.

So, there are several reasons why processing information should be included into the earlier phases of database design:

- Low storage complexity is an objective of conceptual and logical design. Efficiency is the main database processing requirement. These goals are often in contradiction to the other.

- Normalization is the typical approach to minimize storage complexities and operation anomalies. But, normalization algorithms may derive several schemes from the same attribute and functional dependency set.

- Well-designed schemes which are piped from conceptual to logical design and subsequently are implemented by a database management system often imply operational bottlenecks. For different DBMSs different tuning principles are appropriate.

- Database structures providing the most efficient implementation often cannot be derived directly from the given. Therefore, external views may not be supported by the ‘tuned’ internal, logical or conceptual schemes.

Therefore, modern database design approaches advice to include processing information as well. Processing requirements should be always considered at early design steps because conceptual and logical design often generates a database schema that cannot easily be modified once the database is implemented.

There are certain approaches to transformation from conceptual to logical schemes and their implementation under a certain DBMS. Mostly, the mapping to an advantageous internal representation is based on the operational support of the used DBMS. This mapping is based on estimations of tuple numbers and other operational characteristics. A better approach is the computation of integrity maintaining procedures and the estimation of their complexity. This approach can be included into conceptual database design and is more appropriate than usual approach.
According to the provided physical data organization (such as Ring, Isam, Btree, dHashing, ...) operational complexity can be evaluated. Therefore, bottlenecks can be detected and subsequently, they are eliminated in the conceptual schema.

According to this approach a toll has been developed and included into the RADD toolbox [1, 8, 9, 7]. It enables estimation of behaviour and optimization of conceptual schemes. This tool analyzes the complexity of main database operations. Based on the analysis bottlenecks are detected and discussed with the designer using an expert system. Finally, the optimizer proposes different solutions for minimizing operational complexity.

2 The Theoretical Background

Databases can be specified by their structures \( \text{Struc} \), their operations \( \text{Ops} \) and their integrity constraints \( \Sigma \), i.e. the database schema is given by the triple \( \mathcal{S} = (\text{Struc}, \text{Ops}, \Sigma) \). Often we omit the operations assuming that operations are given generically whenever the structure of the database schema is given, i.e. \( \mathcal{S}^* = (\text{Struc}, \Sigma) \). For instance, the definition of relational schemes by their attribute sets implies directly the corresponding manipulation operations of the relation type (select, insert, delete, update), i.e. we are given the pair \((\text{Struc}, \Sigma)\) and assume that \( \text{Ops} \) is to be generated on the basis of \( \text{Struc} \) \((\text{Ops} = \text{GenericOps}(\text{Struc}))\).

Constraints can be classified into static, transition and general dynamic constraints describing correct states, state transitions or state sequences respectively. It is commonly assumed that most of integrity constraints can be defined statically. Consistency is a crucial property of database application systems. There are two approaches to consistency enforcement:

1. Each database instance is checked whether it is correct. This requires constraint checking after each operation or after a certain number of operations.

2. The database instance is assumed to be correct before performing the operation. Then correctness is to be checked according to the effect of the operation.

The first approach is computational infeasible. The second approach uses the correctness of the database instance and is based on integrity enforcement. Then the consistency problem is to guarantee that each specified operation \( o \in \text{Ops} \) will never violate any constraint \( o \in \Sigma \). In conventional database systems, consistency is preserved either by forbidding operations that violate integrity constraints or by rolling back transactions that produce inconsistent database states. A third way to integrity enforcement is that of postponing faulty transactions by introducing repairing actions which compensate incorrect states. The complexity of correctness inspection depends on functionality of the database management system. Normally, database management systems are limited in their support for consistency. It is argued that DBMS only support key and domain constraints. Several approaches have been developed to consistency support:

1. Normalization aims at the derivation of a new structure of the database, a set of operations obtained by applying normalization transformation \( T \) to the operations, and a corresponding set of integrity constraints such that \((\text{Struc}' = T(\text{Struc}), \text{Ops}' = T(\text{Struc}, \text{Ops}), \Sigma' = T(\text{Struc}, \Sigma))\) is equivalent to the schema \((\text{Struc}, \text{Ops}, \Sigma)\) and integrity can be maintained for the new schema.

2. Integrity enforcement aims at the derivation of a new set \( \text{Ops}' \) of operations such that \((\text{Struc}, \text{Ops}', \Sigma)\) is equivalent to the schema \((\text{Struc}, \text{Ops}, \Sigma)\) and each operation is consistent.
3. Integrity restructuring aims at the derivation of a new set $\Sigma'$ of integrity constraints such that $(\text{Struc}, \text{Ops}, \Sigma')$ is equivalent to the schema $(\text{Struc}, \text{Ops}, \Sigma)$ and a better support to consistency is available in the current DBMS.

The normalization approach is well-known and can be applied to ‘normal’ databases. The integrity restructuring approach is not applicable whenever the constraint set is using constraints like inclusion and functional dependencies together. Different approaches to integrity enforcement and database maintenance have been proposed in literature. The specialization approach [5, 2, 3, 4] extends the trigger approach. It is based on the assumption that for any operation applicable to the database schema a specialization of this operation can be developed. This specialization of the operation can be applied to any correct database and the result of this operation is again a correct database. This approach is computationally feasible. As shown in [6] the efficient construction of consistent update operations can be based on linguistic reflection. Type-safe linguistic reflection came up with the development of the ADABTPL language which laid a primary interest on the development of correct database transactions. This paper extends this approach to physical schemes. Based on an estimation of the operational complexity bottlenecks in the design can be detected and remedied. Generic update operations are analyzed. Insert, delete and update are generic in sense that they are applicable to each type of the database schema. Relational database management systems as well as most of the important object-oriented DBMSs (Montage, Itasca, O₂,...) provide such operations. In order to build generator macros for generic update operations we can follow the constructive proof of their existence in [2, 4]. This approach to integrity enforcement can be extended to more general integrity constraints as well.

References


The Tool Box (DB)² Database Design by Beta
The Database Design System \((DB)^2\)

Bernhard Thalheim

Computer Science Department
University of Rostock
Albert-Einstein-Str. 21
D-O-2500 Rostock, FRG
thalheim @ informatik.uni-rostock.dbp.de


Abstract

Database design tools should facilitate database modeling, effectively support database processing and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970’s, various tools for database design have been introduced. Most of them, however, are dependent on the knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The proposed system \((DB)^2\) does not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The Entity-Relationship Model is extended to the Higher-order Entity-Relationship Model (HERM) by relationships of higher degrees and relationships of relationships. This model is used for a high-level database design system \((DB)^2\) (DataBase Design by Beta). The system supports an efficient translation to nested relational, relational, network and hierarchical schemes according to different environments. At present, the system is widely applied in database design.

1 Introduction

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

- to meet all the information (content) requirements of the entire spectrum of users in the given application area;
- to provide a "natural" and easy-to-understand structuring of the information content;
- to conserve the whole semantic information of the designers for a later redesign;
- to achieve all the processing requirements and achieve a high degree of efficiency of processing;
- to achieve the logical independence for query and transaction formulation on this level;
- supporting bad schemes and good for the manipulation of schemes that violate normalization, nonredundancy, etc.;
- coping with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user;
- designing for consistency if the same type of information appears in several design phases.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional complexity to the database design process in such a formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and
number of items included in the database scheme, and further by the semantics defined for the database and the operations.

The design system $(DB)^2$ captures a lot of information about schemes under design. It has a data dictionary in which schema information can be expressed and transformed and is based on a database model that can support all phases of the design process. Nowadays, the design process is understood to capture the structural design and the modelling of the semantics, as well as the description of the behavior of the database, especially the operations defined for the database. We evolved the classical entity-relationship model to the higher-order entity-relationship model (HERM) which can support design in any of the main classical data models and higher order data models and also translation among them. It supports integrity constraints. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated. At present the system $(DB)^2$ is widely used for database design. There is a large number of active groups which are using the system.

The paper is divided into two parts. First we introduce the object-oriented entity-relationship model, the higher-order entity-relationship model. Then we show how this model and the underlying methodology is to be used in the design system $(DB)^2$. For that, the system $(DB)^2$ is explained and the results of the design process using the system are outlined by one example.

## 2 The Object-Oriented Entity-Relationship Model

### 2.1 The Object Model

Objects are to be handled and modelled in databases. They can own an object identifier and are to be characterized by values and references to other objects, i.e.

$$o = (i, \{(s, v), \{ref\}\})$$

The value characterization is to be bound to an already defined structure $s$. Characterized properties of objects are to be described by attributes which forms the structure of the object. Objects have furthermore a special semantics and a general semantics. Operators are to be associated to objects. These operators have a behavior. Object which have the same structure, the same general semantics and the same operators are be collected in classes. The structure, the semantics and the operations of a class is represented in types. Modelling of objects includes in this case the association of objects to classes $C$ and their corresponding value type $T$ and reference type $R$. Therefore, objects are to be represented by $o = (i, \{(C, T, v), \{(C, R, ref)\}\})$.

The known design methodologies vary in the scale of information to be modelled in the types. If objects in the classes can be distinguished by their values then the identifiers can be omitted and we use value-oriented modelling. In the other case, we use an object-oriented approach. In the object-oriented approach different approaches can be distinguished. If all objects are identifiable by their value types or by references to identifiable objects the database is called value-representable. In this case, the database can be modelled by the value-oriented approach too and a mapping from the value-representable scheme to a value-oriented scheme exists. If the database is not value-representable then we have to use object identifiers. It is well-known that in this case either the identifier handling should be made public or the databases can not be updated and maintained. Therefore, value-representable databases are of special interest.

Normally, objects do not exist in a database independently. An objects is to be called kernel objects (or independent) if its existence in the database is independent of the existence of any other object in the database. An object is called characteristic if it describes some other object. Objects can perform a superordinate role in interrelating other objects, in which case they are called associative. The exists associations among objects. Associations can be also objects.

Kernel objects are to be described by entities in the valued-oriented approach. All other object can be entities or relationships. Kernel objects can be distinguished by their values of some attributes. These attributes are called key. In value-representable databases objects are kernel objects if they are identifiable by their values. These objects are represented by entities. All other objects are to be represented by relationships.
The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [10]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types. The clear distinction of independent objects from dependent objects is another advantage of the HERM model and the system \((DB)^2\). This simplifies the modeling since we do not need additional constructs for subtyping.

2.2 The Higher-Order Entity-Relationship Model

We introduce now the higher-order entity-relationship model. Besides the existence of a strong theoretical basis there are several other advantages of the HERM approach:

HERM-schemes are much simpler and are easier understandable than ERM-schemes.

HERM-schemes support abstraction in a simple but comprehensible manner.

HERM-schemes can be translated together with the corresponding constraints, with the corresponding user-defined operations and with the generic operations to normalized relational, hierarchical or network schemes.

The HERM type consists of information on the structure, (static) semantics, operations and behavior (dynamic semantics), i.e.

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior}.
\]

This notation can be generalized to the more general which is out of scope for the purposes of this paper

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior} + \text{Environment}.
\]

The higher-order entity-relationship model has the following modelling constructs:

**Simple attributes** For a given set of domains there are defined attributes and their corresponding domains.

**Nested attributes** Using basic types complex attributes can be defined by means of the following constructors:

- Tuple constructor. Using nested attributes a new nested attribute is defined by the cartesian aggregation.
- Set constructor. Using a nested attribute a new nested attribute is defined by the set aggregation.

Additionally, the bag and the list constructors can be used. For the sake of simplicity we use here only the tuple and set constructors.

**Entities** Entity types are characterized by their attributes. Entity types have a set of attributes which can be used as the identifier of the type.

**Clusters** The union of types is called cluster.

**First-order relationships** First-order relationships types are defined to be associations between entity types or clusters of those. They can be additionally characterized by attributes.

**Higher-order relationships** The relationship type of the order i is defined as an association of relationship types of order less than i or entity types and can be additionally characterized by attributes.

**Integrity constraints** A corresponding logical operator can be defined for each type. A set of logical formulas using this operator can define the integrity constraints which are valid for each instance of the type.

**Operations** Each type can have a set of (conditional) operations and query forms. Furthermore, generic operations like projection, restrictions, insert, (tagged) clustering are predefined for each type.

Examples for the constructs are the following:

A name of a person is defined by the cartesian aggregation

\[
\text{Name}(\text{First}, \text{Fam}).
\]

The membership in societies can be defined by the set aggregation attribute

\[
\text{Membership}\{\text{Member}(\text{Society}, \text{Since})\}.
\]
The address of persons is usually defined to a complex attribute, for example
\[ \text{Address}(\text{State}, \text{City}(\text{Code}, \text{Town}), \text{Street}(\text{Name}, \text{House}(\text{Numb}, \text{Appartm}))) \].

A person can be characterized by its social security number, its name, its address and the sex and has the set \{ SSN \} as the identifier, i.e.
\[ \text{Person} = \{ \{ \text{SSN}, \text{Name}(\text{First, Fam}), \text{Adr}(\text{Zip, Town, Street}(\text{Name, Nr})), \text{Sex} \}, \{ \text{SSN} \} \} \].

The relationship marriage of persons can be modelled by
\[ \text{Marriage} = (\text{Person, Person, From, To}) \].

An example of an integrity constraint over marriage is the following predicate
\[ \text{CorrectMarr} := \text{Marriage}(\text{Person}(\text{Sex}), 1) \neq \text{Marriage}(\text{Person}(\text{Sex}), 2) \]
which indicates that the sex of the first person on Marriage must be different from the sex of the second person in Marriage. We can also use labels instead of the (Person,1).

The function
\[ \text{FamilyName}(x) := (\text{Person}(x)[\text{Name}])[\text{Fam}] \]
is an example of an operation defined for the type Person.

The function
\[ \text{Spouse}(x) := \text{Marriage}(\{\text{Person}(\text{Name})=x, 1\})[\{\text{Person}(\text{Name}), 2\}] + \text{Marriage}(\{\text{Person}(\text{Name})=x, 2\})[\{\text{Person}(\text{Name}), 1\}] \]
is an example of an operation which involves more than one type. Such operations can be used to define roles of relationships.

Operations can be defined using preconditions and postconditions. If, for instance, a type
\[ \text{Student} = \{ \{ \text{SSN}, \text{Name}(\text{First, Fam}), \text{Adr}(\text{Zip, Town, Street}(\text{Name, Nr})), \text{Sex}, \text{StudentNumber} \}, \{ \text{StudentNumber} \} \} \]
is defined and any student is a person which is stored in the database then the insertion operation for the type Student can be defined with one precondition or with a postcondition based on the operation Insert:
\[ \text{Add1(Student, (x,y,z,u,v)) := if Person(x,y,z,u) then Insert(Student, (x,y,z,u,v))}; \]
\[ \text{Add2(Student, (x,y,z,u,v)) := Insert(Student, (x,y,z,u,v)) if not Person(x,y,z,u) then Insert(Person, (x,y,z,u))}. \]

The HERM differs from the classical ERM and from extended ERM in several constructs.

1. Constructs which can be found in different extensions of the classical ERM are: nested attributes, first-order relationships of higher arities, clusters and some integrity constraints like the complexity.

2. The following constructs are new: higher-order relationships, integrity constraints including the generalized complexity, operations and conditional operations.

3. Since weak entity types and Is-A-relationships can be represented directly in the HERM there is no direct need for these constructs in the HERM.

4. Kernel objects are distinguished from other objects and differently represented by entity types. All other objects are to be represented by relationship types. For this reason the schemes in HERM are simpler several times than ERM schemes.

The entity-relationship (ER) model is one of the most popular database design models. Despite numerous positive features of the ER approach there still exists a strong need for a theoretic basis. This theory must be able to define sets of semantically well-formed ER schemes for particular user-specific ER-techniques as well as for subproblems as scheme design, view integration, query generation, and scheme transformation. Codd [4, p. 477] states even that the entity-relationship model "is clearly the winner in terms of its lack of precise definitions, lack of clear level of abstraction, and lack of mental discipline". In [16] the entity-relationship model is extended and it is shown that a precise, formal definition exists. This is based on the HERM-methodology which is using an abstract-data-type-approach.

3 HERM and the system \((DB)^2\)

3.1 The Design System

The system \((DB)^2\) (Data Base Design by Beta: DBDB = \((DB)^2\)) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. \((DB)^2\) supports database design from requirements
analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual
design, thus offering a choice between the relational, network, or hierarchical models for the logical design.
Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be
generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with
already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the
new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can
be used for creating query compilers which are more productive, effective and forceful.

One of the most important advantages of the system is the interactive user-driven input of information. It
is used an easily intelligible, comprehensible and understandable "fill in the blanks” input procedure which is
required in the literature as the ideal input form.

The tools developed in $(DB)^2$ can be divided into the following groups:

1. Analyzers (for reports, schema checking, the normalization, access evaluation tools).
2. Transformers (for new schemes which are information content equivalent).
3. Heuristic, user-dialog-driven tools (for solving view cooperation problems).
4. Translators appropriate for the target data model (relational, network or hierarchical) and system.
5. Query definition module (which is also used for behavioral normalization).

The general system architecture is represented in figure 1 where thicklines denote the data flow and thinlines
the control flow.

The system is based on the following HERM methodology. In database modeling three different perspectives
can be identified. Different models stress only some of those. The structure-oriented, semantic perspective
focuses on what kind of data are stored in the database, what constraints apply to these data, and what kinds
of data are derivable. The process-oriented perspective is concerned with the processes or activities performed
in the application area. The behavior-oriented perspective is concerned with how events in the real world
trigger actions in the database systems. Database modeling should consider all three perspectives. The HERM
methodology supports this approach and is an extension of [1, 3, 8, 9]. We start with meaningful examples (the
first step), develop the structure and the semantics of the database under design (the next four steps), generate
the operations and model the behavior of the database (the next two steps). Throughout the procedure, checks
are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational
design can require for performance reasons the redesign of the structure. The eight steps can be defined as
follows:
1. transform examples in elementary entity and relationship types, analyze the structure;
2. graphical representation of types and their associations, population check;
3. eliminate surplus, derivable types, determine roles;
4. modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. normalize the types;
6. modeling of operators over the types and the behavior;
7. optimizing types and deriving views;
8. check the completeness, consistency of the design with the examples.

This methodology is directly supported by the design system \((DB)^2\).

The HERM and \((DB)^2\) methodologies generalizes different approaches presented in the literature (see for instance \([5, 9, 13]\)). The most important classes of integrity constraints used in the database modeling system [14] are the functional dependencies, the generalized functional dependencies, the multivalued dependencies, the inclusion dependencies, the exclusion dependencies. In \((DB)^2\) there are implemented three different choices for the translation of Is-A- and generally (0,1)-relationships. The design system \((DB)^2\) is able to change from one representation to the other.

The system can be used for the support over the complete lifecycle of database systems. Most known methodologies are not well adapted to the lifecycle because the design information is not used after the design process. Design is a far more iterative process as captured in the straight forward lifecycle model of these methodologies. Using \((DB)^2\) during the whole system development the complete design information is usable for restructuring the database. This makes it possible to restructure the database and during restructuring to recompile in accordance to the old and new structures the programs which are specified in the HERM algebra.

The behavior can be specified using generic operations like insert and deriving restrictions to the behavior from the semantical information. The maintenance complexity is one of the most important behavioral database design criterias. The database designer can design the structural, semantical, operational and behavioral information on the application area and the system generates in accordance to the DBMS's type and the distribution the corresponding data structures and transactions.

Database design is at present considered as top-down design. This approach is only useful for single-designer support systems. However, in the case the designer can detect only in a restricted manner similarities between the concepts. This is especially useful if the design is made by several designers. In this case the view integration is one of the most difficult problems which has to be solved for an efficient management of the whole database system. This task can not be solved using only methods of top-down design. The window manager approach in \((DB)^2\) can be used to detect similarities. Moreover, designing the process information at the same time adds the possibility to adapt the structure to the process efficiency. For this reason, new normalization algorithms are used in \((DB)^2\) which prohibit normalization if this contradicts the process efficiency.

3.2 An example

Now we demonstrate the utilization of the \((DB)^2\)-system. Firstly, an example is proposed. Secondly, the scheme is to be design in the frame-based fill-in-form language. The system generates the graphical representation. The mode of the representation (with/without semantics, in detail/without details) can be chosen. During the last step, the design is to be translated to the language of a database management system taking into account the specific properties of the DBMS.

3.2.1 The University Example

Let us design a HERM scheme for a simple university application covering the following information:

1. A catalogue of persons working or studying in the university. A person has a social security number or person’s number which uniquely identifies this person. Persons have a name (first and last names and titles), and an addresses (with a town, a zip, and a street).
2. A catalogue of students which are characterized by their students numbers. A student is also a person. Students have a major and a minor specialization. They are supervised by a professor.
3. A catalogue of professors (with their specialisation). Professors are persons. Each professor is associated with a department.

4. A catalogue of courses, given in the university and characterized by a unique course number, and names. A course can have different prerequisites.

5. A catalogue of course offerings with the course, the semester (year and season), the room (room number and building) and the professor.

6. A catalogue of projects characterized by a unique project number, the begin and the end, and the title.

7. A catalog of students activities. Students can enroll a certain course in a term. They obtain a final grade.

3.2.2 The structural description

This example can be modeled by the following entity and relationship types:

Person = ( { Person's number, Name (LastName, FirstName, { Title } ) }, Address (Zip,Town, Street(Name,Nr) ) , Person's number),
Course = ( { CNu, CName } , { CNu } ),
Project = ( { Num, Begin, End, PName } , { Num } ),
Room = ( {Nr,Building } , { Nr,Building } ),
Student = ( Person, { StudNr } ),
Professor = ( Person, { Speciality } ) ,
Department = ( { DName, Director, Phones { Phones } } , { DName } ),
Major = ( Student, Department, 0 ),
Minor = ( Student, Department, 0 ),
In = ( Professor, Department , 0 ),
HasProject = (Project, Leads:Professor, Professor + Person , 0 ),
Prerequisite = (Course, Course, 0 ),
Supervisor = ( Student, Professor, { Since } ),
Lecture = (Professor, Course, Room, Semester, { Time(Day,Hour) }),
Semester = ({ Year, Season }, { Year, Season }),
CourseEnrollment = (Student, Lecture, { Result }).

Figure 2 represents the diagram without additional integrity constraints like cardinality constraints, exclusion and inclusion dependencies and generalized functional dependencies.

The path inclusion dependency
Student-Enroll-Lecture-Course-1-Prerequisite-2-Course:
Enroll.Student.StudNr, Prerequisite.(Course,2).CNu
⊆ Student-Enroll-Lecture-Course:

expresses the restriction that a student can enroll lectures on courses only if he/she completed the prerequisites of this course.

The example shows that the proposed extension of the HERM solves some of the drawbacks of the relational and the entity-relational database modeling like uniformity of relations, view definition, problems in deterministic normalization algorithms, problems in query languages, set inadequacy, sequence modeling. This model maintains also the advantages of the relational and the entity-relationship model like availability of database design theory, query language, access path propagation, optimization, user-friendliness, visibility, understandability, and data dependency propagation.

3.2.3 The Relational Representation

The system supports the translation of the schemes, the semantics and the operations to specified types of database management systems. Since each of the known database systems has several implementational restrictions which have to be considered during the translation process and this can be done automatically the system (DB)$^2$ computes using the input the corresponding schemes.
Figure 2: \((DB)^2\)-Diagram of the University Database
Any entity-relationship database can be represented by a collection of relations. For each entity set and for each relationship set in the database, there is a unique relation which is assigned the name of the corresponding type. Each relation type has a number of columns which, again, have unique names. The semantics of the scheme has to be translated together with the scheme. For instance, the entity-relationship scheme contains implicitly also inclusion dependencies. These are important especially for the maintenance of the scheme. If a relationship is embedded in a scheme then this relation scheme contains a foreign key. The translations used in [12] are incomplete. The complete translation theory is developed in [15].

The following translation demonstrates the translation for systems which supports views like Ingres 6.3. During translation we obtain, for instance, the following relation schemes:

- **Person** = \{ Person’s number,Name(First,Fam,Title), \\
  Addr(Zip,Town,Street(Name,Nr)) \}, \; key = \{ Person’s number \} ,

- **Course** = \{ CNu,CName \}, \; key = \{ CNu \} ,

- **Room** = \{ Nr,Building \} , \; key = \{ Nr,Building \} ,

- **Student** = \{ StudNr, Person’s number, Minor, Major \}, \; key = \{ StudNr \} ,

- **Professor** = \{ Persons’s number,Speciality \}, \; key = \{ Persons’s number \} ,

- **Prerequisite** = \{ (Course,1). CNu,(Course,2). CNu \} , \; key = \{ (Course,1). CNu,(Course,2). CNu \} ,

- **Lecture** = \{ Professor. Person’s number, Course. CNu, Semester(Year,Season), \\
  Room. Nr, Room. Building, Time(Day,Hour) \}, \; key = \{ Professor. Person’s number, Course. CNu, Semester(Year,Season) \} ,

- **Enroll** = \{ Student.StudNr, Professor.Person’s number,Course. CNu, \\
  Semester(Year,Season), Result \}, \; key = \{ Student.StudNr, Course. CNu, Semester(Year,Season) \} .

Furthermore, in the translation process we obtain other dependencies like inclusion dependencies, exclusion dependencies, and functional dependencies [6]. The above presented path inclusion dependency is to be translated to the algebraic dependency

\[ \text{Enroll} \ni \text{Enroll.Course.CNu} = \text{Prerequisite.(Course,1).CNu} \]

\[ \text{Prerequisite} \ni \text{Enroll.Student.StudNr, Prerequisite.(Course,2).CNu} \ni \]

\[ \subseteq \text{Enroll.Student.StudNr, Enroll.Lecture.Course.CNu}. \]

Using the theory [16], the translated schemes are automatically simplified. For instance, the attribute identifiers are simplified by synonyms. Furthermore, the behavior of generic operations is to be computed and used for the maintenance of the schema. For instance, the maintenance simplicity implies by the last dependency that the insert and update of prerequisites is restricted to only some simple cases whereas the delete operation on enrollments is allowed only for the whole bulk of students data.

### 4 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system \((DB)^2\) was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and system have the following advantages:

1. The model is easy understandable, simple and perceivable.

   The model can be used as a basis of database design tools [14, 17]. The system \((DB)^2\) is used at present by more than 95 user groups.

   The modeling approach is simple to use even for large problems.

   Since the model uses graphical representations the modeling result is easier to understand and visible.

   In an experiment, 20 novice or end-user database designers learned the HERM methodology and later designed different database schemes in different areas. Our experience was that that the methodology was easily accepted, led to more accurate, to less redundant schemes and to a correct modeling of complex relationships.
Using query forms [18] the user can specify application transactions and programs on the top of the HERM which reduces substantially the complexity of application programming.

2. The results of the design are much simpler than in other approaches.

We have used the the model for modeling also some more complex applications. One observation is that the obtained schemes are from three to five times simpler than those obtained by other models. The example of [13] is simplified by four times and can be placed on one page or one screen. In other examples, the simplification makes it possible to find a model. Using this modeling approach, an airport counter application was modelled by less than 40 entity types and less than 120 relationship types whereas the original solution with more than 150 entity types and more than 400 relationship types was unacceptable by users because of complexity and non-transparency.

The simplification leads also to a better understanding of the application and makes normalization easier to perceive.

The schemes avoid additional redundancy. Using HERM, the normalization and the minimalization of schemes can be considered together.

3. The utilization of \((DB)^2\) is more natural and simpler. Only necessary facts are to be expressed.

The model supports a direct translation to the three classical database models. This translation preserves normal forms. Since a direct translation to relational, network and hierarchical schemes can be used the design decisions directly could be used to obtain schemes in normal forms. The translation theory can be used for a multimodel- and multisystem-support [18] and presents a practical solution to interoperability of systems.

The HERM algebra is used for query definition. The corresponding relational, network or hierarchical queries can be automatically generated.

The model supports a rich set of constraints. These constraints are used for the development of the scheme equivalence. Although the excessive number of fact-encoding mechanisms means that the same semantic unit can be declared in many syntactically different and compatible ways, the information described is equivalent. This equivalence theory can be used for automatic modification of schemes [2].

The database maintenance procedures can be derived using the design information.

Using a knowledge base previous and system-provided design decisions can be reused or partially reused what simplifies the design task. Furthermore, similarities in the design can be detected and used for simplification of the implementation.

Using the whole design information the retranslation of application programs can be used for the adaption of existing database systems to changing environments.

4. The model and the design system have a strong theoretical basis.

The model is based on a multitype logic which is equivalent to the first-order predicate logic. For this reason, results known from discrete mathematics and relational theory [15] can be used.

The model covers the complete modeling information. The structure, static semantics, generic and user-specified operations and behavior of an application can be described by the model.

The theory is simplified and cleaned up. Sequences, subsets and powersets of objects can be modeled directly. Is-A-Relationships are treated in a better way. Weak entity types can be avoided completely. A normal form theory is developed for the HERM. Using this normal form theory, we can obtain normalized schemes like in the classical theory.

5. The theory is applicable to practical needs.

Based on the theory a multi-paradigm, robust design methodology is developed [17, 18] which incorporates approaches known in object-oriented modeling [11], modular programming [14] and programming in large.

Using this modeling approach, a view cooperation concepts was developed. Since full view integration is not decidable and not axiomatizable view cooperation is the only applicable approach.

The approach can be used for reverse engineering. Systems and programs developed for one management system can be recompiled and adapted to other management systems.
References


Design with the Database Design System $(DB)^2$

Bernhard Thalheim
Computer Science Department, Rostock University
Albert-Einstein-Str. 21, D-O-2500 Rostock, FRG
thalheim @ informatik.uni-rostock.dbp.de

Abstract

Database design tools should facilitate database modeling, effectively support database processing and transform a conceptual schema of the database to a high-performance database schema in the model of the corresponding DBMS. Since the late 1970’s, various tools for database design have been introduced. Most of them, however, are dependent on the knowledge, comprehension and experience of the database analyst and their knowledge in normalization theory. The proposed system $(DB)^2$ does not require the user to understand the theory, the implementational restrictions and the programming problems in order to design a database scheme. A novice designer can create a database design successfully using the system. The Entity-Relationship Model is extended to the Higher-order Entity-Relationship Model (HERM) by relationships of higher degrees and relationships of relationships. This model is used for a high-level database design system $(DB)^2$ (DataBase Design by Beta). The system supports an efficient translation to nested relational, relational, network and hierarchical schemes according to different environments. At present, the system is widely applied in database design.


1 INTRODUCTION

The problem of database design can be stated as follows: Design the logical and physical structure of a database in a given database management system to contain all the information required by the user and required for an efficient behavior of the information system.

The implicit goals of database design are:

• to meet all the information (content) requirements of the entire spectrum of users in the given application area;
• to provide a "natural" and easy-to-understand structuring of the information content;
• to conserve the whole semantic information of the designers for a later redesign;
• to achieve all the processing requirements and achieve a high degree of efficiency of processing;
• to achieve the logical independence for query and transaction formulation on this level.

While on the one hand the inputs to the process are so informal, the final output of the database design is a database definition with formal syntax and with qualitative and quantitative decisions regarding such problems of physical design like physical placement, indexing and organization of data. This adds additional
complexity to the database design process in such a formal design must be turned out from, at times, extremely informal available information. The main complexity of the design process is already given by the complexity and number of items included in the database scheme, and further by the semantics defined for the database and the operations.

The design system \((DB)^2\) captures a lot of information about schemes under design. It has a data dictionary in which schema information can be expressed and transformed and is based on a database model that can support all phases of the design process. Nowadays, the design process is understood to capture the structural design and the modelling of the semantics, as well as the description of the behavior of the database, especially the operations defined for the database. We evolved the classical entity-relationship model to the higher-order entity-relationship model (HERM) which can support design in any of the main classical data models and higher order data models and also translation among them. It supports integrity constraints. Constraint declarations include: attribute data types, non-null attributes, attribute combinations forming primary and candidate entity keys, functional dependencies, multivalued dependencies, and inclusion dependencies. They also include relationship cardinalities and other dependencies. The chosen constraint set is powerful enough to capture the constraints in each schema, and to support the generation of equivalent schemes. Without constraints, there are only trivial equivalences between schemes. Without equivalence, it is impossible to justify transformations as provably preserving the information content of a schema. Furthermore, using the design information procedures for the maintenance of the database can be generated.

The published and known literature underlies many parts of the database design process. In particular, the following theories can be used for the database design [ElMasri89],[Heuer89],[Hull89],[Nijssen89],[Rishe88],[Thalheim91]:

- theory of data models;
- theory of normalization of relational databases;
- theory of scheme equivalence.

But the published work needs also several extensions and adaptations in several respects [Thalheim91'], [Thalheim92]:

- supporting bad schemes and good for the manipulation of schemes that violate normalization, nonredundancy, etc.;
- coping with wrong or contradictory results of the design process, incomplete, unreliable, incorrect input from the user;
- designing for consistency if the same type of information appears in several design phases.

In database modeling three different perspectives can be identified. Different models stress only some of those. The structure-oriented, semantic perspective focusses on what kind of data are stored in the database, what constraints apply to these data, and what kinds of data are derivable. The process-oriented perspective is concerned with the processes or activities performed in the application area. The behavior-oriented perspective is concerned with how events in the real world trigger actions in the database systems. Database modeling should consider all three perspectives. The following strategy supports this approach and is an extension of [Abiteboul84],[Chen83],[Nijssen89],[Rishe88]. We start with meaningful examples (the first step), develop the structure and the semantics of the database under design (the next four steps), generate the operations and model the behavior of the database (the next two steps). Throughout the procedure, checks are performed to ensure that no design decision is erroneous or contradicts design decisions before. Operational design can require for performance reasons the redesign of the structure. The eight steps can be defined as follows:

1. transform examples in elementary entity and relationship types, analyze the structure;
2. graphical representation of types and their associations, population check;
3. eliminate surplus, derivable types, determine roles;
4. modeling of semantics (uniqueness constraints, mandatory role, subtype, equality, exclusion, subset, path, functional,...), check that each entity type can be identified;
5. normalize the types;
6. modeling of operators over the types and the behavior;
7. optimizing types and deriving views;
8. check the completeness, consistency of the design with the examples.

This strategy is directly supported by the design system \((DB)^2\) which is to be described in section 3. The strategy is one refinement of the general design methodology described in section 3.

Database design is based on one or more data models. A large number of conceptual data models have been proposed. However, actual experience with the use of these models as a basis for implementing a generalized DBMS is very scant. While most models have been proposed primarily for stand-alone database management systems and are adapted to implementational restrictions in database systems, it is increasingly apparent that data models will be from one hand side incorporated directly into programming languages and a variety of tools (e.g. CAD/CAM, expert systems, knowledge bases) and from the other hand side have to be extended to interoperating environments and multisystem- and multimodel-paradigms. Nearly all early commercial DBMS implementations were based on the hierarchical model such as IMS and SYSTEM-2000 or the network model such as IDS and IDMS or the relational model such as INGRES, DB2. The relational data model was proposed as a simple and theoretically well-founded representation of data, and it has soon become the most important model for database systems (see for example [Paredaens89],[Ullman89]). The primary virtues of the model are its rigorous mathematical foundation and the correspondence of a relation with the notion of a table. However, research efforts have highlighted a large number of drawbacks to the relational model. Rather than abandon the relational paradigm because of these disadvantages, we are interested in extending relational languages in a way that incorporates useful ideas from alternative language paradigms but allows the retention of most, if not all, of the advantages of the relational approach.

The entity-relationship (ER) model is one of the most popular database design models. Despite numerous positive features of the ER approach there still exists a strong need for a theoretic basis. This theory must be able to define sets of semantically well-formed ER schemes for particular user-specific ER-techniques as well as for subproblems as scheme design, view integration, query generation, and scheme transformation. Additionally, the formalism has to be suited for computer aided software engineering tools. In [Yaseen89],[Thalheim89] the suitability of the HERM approach for the solution of database design problems is shown. One reason for the huge variety of extensions of the ER model is that a well-founded theory is still under development. Codd [Codd90, p. 477] states even that the entity-relationship model ”is clearly the winner in terms of its lack of precise definitions, lack of clear level of abstraction, and lack of mental discipline”. In [Thalheim92] the entity-relationship model is extended and it is shown that a precise, formal definition exists. This is based on the HERM-methodology which is using an abstract-data-type-approach.

The paper is divided into two parts. First we introduce the object-oriented entity-relationship model, the higher-order entity-relationship model. Then we show how this model and the underlying methodology is to be used in the design system \((DB)^2\). For that, the system \((DB)^2\) is explained and the results of the design process using the system are outlined by one example.

## 2 The Object-Oriented Entity-Relationship Model

### 2.1 The Object Model

Objects are to be handled and modelled in databases. They can own an object identifier and are to be
characterized by values and references to other objects, i.e.

\[ o = (i, \{(s, v), \{ref\}\}) \]

The value characterization is to be bound to an already defined structure \( s \). Characterized properties of objects are to be described by attributes which forms the structure of the object. Objects have furthermore a special semantics and a general semantics. Operators are to be associated to objects. These operators have a behavior. Object which have the same structure, the same general semantics and the same operators are to be collected in classes. The structure, the semantics and the operations of a class is represented in types. Modelling of objects includes in this case the association of objects to classes \( C \) and their corresponding value type \( T \) and reference type \( R \). Therefore, objects are to be represented by \( o = (i, \{(C, T, v), \{(C, R, ref)\}\}) \).

The known design methodologies vary in the scale of information to be modelled in the types. If objects in the classes can be distinguished by their values then the identifiers can be omitted and we use value-oriented modelling. In the other case, we use an object-oriented approach. In the object-oriented approach different approaches can be distinguished. If all objects are identifiable by their value types or by references to identifiable objects the database is called value-representable. In this case, the database can be modelled by the value-oriented approach too and a mapping from the value-representable scheme to a value-oriented scheme exists. If the database is not value-representable then we have to use object identifiers. It is well-known that in this case either the identifier handling should be made public or the databases can not be updated and maintained. Therefore, value-representable databases are of special interest.

Normally, objects do not exist in a database independently. An objects is to be called kernel objects (or independent) if its existence in the database is independent of the existence of any other object in the database. An object is called characteristic if it describes some other object. Objects can perform a superordinate role in interrelating other objects, in which case they are called associative. The exists associations among objects. Associations can be also objects.

Kernel objects are to be described by entities in the valued-oriented approach. All other object can be entities or relationships. Kernel objects can be distinguished by their values of some attributes. These attributes are called keys. In value-representable databases objects are kernel objects if they are identifiable by their values. These objects are represented by entities. All other objects are to be represented by relationships.

The classical entity-relationship model uses entity types for the representation of kernel and other objects which are not associations. Only associations are represented by relationships. The recently developed standard drops partially this strictness [Spencer91]. The HERM-approach uses the weakest form of the distinction between entities and relationships which is theoretically sound. Kernel objects are to be described by entity types. All other objects, especially existence dependent objects like characteristic objects are describable by relationship types.

### 2.2 The Higher-Order Entity-Relationship Model

We introduce now the higher-order entity-relationship model. Besides the existence of a strong theoretical basis there are several other advantages of the HERM approach:

- HERM-schemes are much simpler and are easier understandable than ERM-schemes.
- HERM-schemes support abstraction in a simple but comprehensible manner.
- HERM-schemes can be translated together with the corresponding constraints, with the corresponding user-defined operations and with the generic operations to normalized relational, hierarchical or network schemes.

The HERM type consists of information on the structure, (static) semantics, operations and behavior (dynamic semantics), i.e.

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior}
\]

This notation can be generalized to the more general which is out of scope for the purposes of this chapter

\[
\text{HERM-Type} = \text{Structure} + \text{Semantics} + \text{Operations} + \text{Behavior} + \text{Environment}
\]

The higher-order entity-relationship model has the following modelling constructs:

**Simple attributes** For a given set of domains there are defined attributes and their corresponding domains.
Nested attributes Using basic types complex attributes can be defined by means of the following constructors:

- Tuple constructor. Using nested attributes a new nested attribute is defined by the cartesian aggregation.
- Set constructor. Using a nested attribute a new nested attribute is defined by the set aggregation.

Additionally, the bag and the list constructors can be used. For the sake of simplicity we use here only the tuple and set constructors.

Entities Entity types are characterized by their attributes. Entity types have a set of attributes which can be used as the identifier of the type.

Clusters The union of types is called cluster.

First-order relationships First-order relationships types are defined to be associations between entity types or clusters of those. They can be additionally characterized by attributes.

Higher-order relationships The relationship type of the order i is defined as an association of relationship types of order less than i or entity types and can be additionally characterized by attributes.

Integrity constraints A corresponding logical operator can be defined for each type. A set of logical formulas using this operator can define the integrity constraints which are valid for each instance of the type.

Operations Each type can have a set of (conditional) operations and query forms. Furthermore, generic operations like projection, restrictions, insert, (tagged) clustering are predefined for each type.

Examples for the constructs are the following:

A name of a person is defined by the cartesian aggregation
\[
\text{Name}(\text{First}, \text{Fam}).
\]
The membership in societies can be defined by the set aggregation attribute
\[
\text{Membership}\{\text{Member}(\text{Society},\text{Since})\}.
\]
The address of persons is usually defined to be a complex attribute, for example
\[
\text{Address}(\text{State}, \text{City}(\text{Code}, \text{Town}), \text{Street}(\text{Name}, \text{House}(\text{Numb}, \text{Appartm}))).
\]
A person can be characterized by its social security number, its name, its address and the sex and has the set
\[
\{\text{SSN}\} \text{ as the identifier } , \ i.\ e.
\]
\[
\text{Person} = (\{\text{SSN}, \text{Name}(\text{First}, \text{Fam}), \text{Adr}(\text{Zip}, \text{Town}, \text{Street}(\text{Name}, \text{Nr})), \text{Sex}\}, \{\text{SSN}\}).
\]
The relationship marriage of persons can be modelled by
\[
\text{Marriage} = (\text{Person}, \text{Person}, (\text{From}, \text{To})).
\]
An example of an integrity constraint over marriage is the following predicate
\[
\text{CorrectMarr} := \text{Marriage}(\text{Person}(\text{Sex}),1) \neq \text{Marriage}(\text{Person}(\text{Sex}),2)
\]
which indicates that the sex of the first person on Marriage must be different from the sex of the second person in Marriage. We can also use labels instead of the (Person,1).

The function
\[
\text{FamilyName}(x) := (\text{Person}(x)[\text{Name}])[\text{Fam}]
\]
is an example of an operation defined for the type Person.

The function
\[
\text{Spouse}(x) :=
\text{Marriage}((\text{Person}(\text{Name})=x,1))[\{\text{Person}(\text{Name}),2\}]
+ \text{Marriage}((\text{Person}(\text{Name})=x,2))[\{\text{Person}(\text{Name}),1\}]
\]
is an example of an operation which involves more than one type. Such operations can be used to define roles of relationships.

Operations can be defined using preconditions and postconditions. If, for instance, a type
\[
\text{Student} = (\{\text{SSN}, \text{Name}(\text{First}, \text{Fam}), \text{Adr}(\text{Zip}, \text{Town}, \text{Street}(\text{Name}, \text{Nr})), \text{Sex}, \text{StudentNumber}\}, \{\text{StudentNumber}\})
\]
is defined and any student is a person which is stored in the database then the insertion operation for the
type Student can be defined with one precondition or with a postcondition based on the operation Insert:
\[
\text{Add1(Student,}(x,y,z,u,v)) := \text{if Person}(x,y,z,u) \text{ then Insert(Student,}(x,y,z,u,v))
\]
\[
\text{Add2(Student,}(x,y,z,u,v)) := \text{Insert(Student,}(x,y,z,u,v))
\]
\[
[\text{if not Person}(x,y,z,u) \text{ then Insert(Person,}(x,y,z,u))]
\]
The HERM differs from the classical ERM and from extended ERM in several constructs.
1. Constructs which can be found in different extensions of the classical ERM are: nested attributes,
   first-order relationships of higher arities, clusters and some integrity constraints like the complexity.
2. The following constructs are new: higher-order relationships, integrity constraints including the gener-
   alized complexity, operations and conditional operations.
3. Since weak entity types and Is-A-relationships can be represented directly in the HERM there is no
   direct need for these constructs in the HERM.
4. Kernel objects are distinguished from other objects and differently represented by entity types. All
   other objects are to be represented by relationship types. For this reason the schemes in HERM are
   simpler several times than ERM schemes.

The model is well-founded [Thalheim92]. Problems known in nested relational models are omitted. Since the
aggregate operations are based on the kind of equality concept used and since the key concept is based on
the equality concept the user can specify different equality concepts. Based on this decision the operations
can be specified. Queries can be specified on this operational environment.

Further, in [Thalheim92] and [Yaseen89] a normalization theory was developed. The main criterion of
normalization is maintenance simplicity which consists of two parts: storage and access simplicity and opera-
tional simplicity. Since the two parts of maintenance simplicity are conflicting the user should specify his/her
preferences. On the basis of integrity constraints several normal forms could be choosen. For instance, if
there are specified only functional dependencies then the elementary key normal form (for nested relations)
is the goal of normalization. Relationship types inherit the keys of the underlying component types. The
decomposition of keys of these types leads to an increase of the arity of the relationship type. Therefore,
a decomposition can be rejected for performance reasons. Integrity constraints can be used for the decision
whether a decomposition is rejected or not.

2.3 OPERATIONS AND BEHAVIOR

The generic definition of manipulation operations is one of the benefits of the relational model. This benefit
is inherited by the HERM. The operations Insert, Delete and Update are defined if a rational tree exists. If
the definition of these operations leads to infinite cycles the operation is not defined for the corresponding
type.

The user can specify its own operations [Yaseen89] using a logical DBPL-like syntax [Schmidt90].

Since a database schema can be considered as a small database itself it is natural to model schema
evolution as database evolution. Therfore, we use meta-operations for the schema evolution. This approach
is not implemented in \((DB)^2\) but at present used in RAD [Bachmann90].

The behavior modeling is an inherent part of HERM modeling. The (dynamic) behavior can be character-
ized by sequences of database (states). It can be expressed by temporal logical formulas which are interpreted
in such state sequences [Lipeck89]. Several classes of formulas are simpler noted by special integrity con-
straints. Behavior is specified according to the following layers:

The interface layer specifies the visibility of parts of databases and restricts the updatebility of the database.
The operational layer specify the behavior in terms of operations which are allowed to modify the database
contents.
The dynamic constraint layer specifies the admissible database sequences.
An important task for a database design tool is the automatic translation of constraints from the dynamic constraint layer to pre- and postconditions in the operational layer. Operations and dynamic integrity constraints might violate dynamic integrity constraints. If such a violation occurs, the corresponding transaction or operation should be rejected afterwards. For this purpose, an integrity browser is to be implemented in RAD [Bachmann90]. The browser is able to detect any violation of a constraint, no matter whether static or dynamic. There are several classes of dynamic integrity constraints which are directly supported on the basis of high-level constructs: transition constraints (e.g. the salary does not decrease), insert-delete-restrictions (e.g. the insert operation is not allowed for a class), future-related classical constraints (e.g. if a certain value exists then another value appears sometimes in the future).

The user can specify the type of integrity constraint enforcement. Integrity constraints can be enforced directly (without delay) or with a certain delay (deferred ... until ...). In the last case the operations are to be modeled by transactions which commit after checking of delayed integrity constraints. The default mode is immediate).

Further, for operations and integrity constraints the exception actions can be specified.

3 HERM AND THE SYSTEM \((DB)^2\)

3.1 THE DESIGN SYSTEM

The system \((DB)^2\) (Data Base Design by Beta; \(DBDB = (DB)^2\)) is purposing to produce a graphics-oriented, PC-based prototype system for the database designer. \((DB)^2\) supports database design from requirements analysis through high-level physical design, using the higher-order entity-relationship model for the conceptual design, thus offering a choice between the relational, network, or hierarchical models for the logical design. Within the framework of progressive refinement and iteration, the system allows interchangeable designs to be generated, displayed, manipulated, analyzed, and transformed. Each iterative design step is compared with already predefined abstract queries. Using this comparison, a new set of predefined queries is generated for the new schema. Using a query improving procedure, the set of predefined queries is optimized. These tools can be used for creating query compilers which are more productive, effective and forceful.

One of the most important advantages of the system is the interactive user-driven input of information. It is used as an easily intelligible, comprehensible and understandable "fill in the blanks" input procedure which is required in the literature as the ideal input form.

The tools developed in \((DB)^2\) can be divided into the following groups:

1. Analyzers which produces different types of reports. They include the schema checker, the normalizer, the logical and physical access evaluation tools, and the index selection tools.

2. Transformers produce a new scheme with information content equivalent to the input. The transformers handle scheme translation as described in normalization theory. They should cope with missing data if need be. The large translations among data models depend on a set of smaller transformations such as removing attributes, declaring keys, and create link records.

3. Heuristic, user-dialog-driven tools produce a new scheme not equivalent to the input. They are invoked early in the design process, and make best-guess decisions to quickly produce a usable, editable result. The view analyzer looks for homonyms and synonyms in two schemes. Synonyms, i.e. attributes having approximately the same meaning and domain and used in different relationships, and homonyms, i.e. attributes having the same spelling but a different meaning in different relationships, can be detected by tools and user-driven identified or distinguished.

4. After partial view integration, a translator produces a scheme appropriate for the target data model. The translation to the physical level adds suggested indexes and clustering.
5. After partial view integration, a query definition module is used for the definition of query-forms. These query forms are to be used for the normalization of the translated schemes. The end-user is able to define queries on the HERM level.

6. The scheme and the defined queries are translated into relational, network or hierarchical schemes and queries. In the case of the relational translation there are implemented four different alternatives which can be chosen according to the properties of interrelationships and according to the support of provided by the DBMS.

The general system architecture is represented in figure 1 where thicklines denote the data flow and thinlines the control flow.

---

3.2 Methodologies and Design-By-Units

The system is based on the HERM methodology. The HERM and \((DB)^2\) methodologies generalizes different approaches presented in the literature (see for instance [ElMasri89],[Rishe88],[Teorey89]). The most important classes of integrity constraints used in the database modeling system [Thalheim89] are the functional dependencies, the generalized functional dependencies, the multivalued dependencies, the inclusion dependencies, the exclusion dependencies. In \((DB)^2\) there are implemented three different choices for the translation of Is-A- and generally \((0,1)\)-relationships. The design system \((DB)^2\) is able to change from one representation to the other.

The system can be used for the support over the complete lifecycle of database systems. Most known methodologies are not well adapted to the lifecycle because the design information is not used after the design process. Design design is a far more iterative process as captured in the straight forward lifecycle model of these methodologies. Using \((DB)^2\) during the whole system development the complete design information is usable for restructuring the database. This makes it possible to restructure the database and during restructuring to recompile in accordance to the old and new structures the programs which are specified in the HERM algebra.

The underlying methodology can be adapted to the three different views of the design process:
1. the structural view is concerned with the description of the structure and of the static semantics and with tasks like integrity;

2. the behavioral view which contains the description of the operations, usual transactions and the behavior of the database;

3. the implementational view which captures the distribution of the data as well as the storage and management of the data.

Normally, in databases these three views are to be handled independently. The behavior can be specified using generic operations like *insert* and deriving restrictions to the behavior from the semantical information. The implementational view depends from the underlying systems and from the distribution of the database parts. The strict separation of these three views from each another is the reason for the mismatch between the abstract given database information and the maintenance of the database during the lifetime. The maintenance complexity is one of the most important database design criterias. The underlying principle of decomposition and normalization in relational databases is the simplification of the maintenance. In order to design a database with the full understanding of the actual complexity of the data and the full understanding of the semantics of the data, the database designer should be supported by a system which captures the structural view as well as the behavioral view and which is able to derive from this information and from the implementational view the corresponding implementation of the database. This task was shown to be solvable using the system $(DB)^2$. The database designer can design the structural, semantical, operational and behavioral information on the application area and the system generates in accordance to the DBMS’s type and the distribution the corresponding data structures and transactions.

The clear distinction of independent objects from dependent objects is another advantage of the HERM model and the system $(DB)^2$. This simplifies the modeling since we do not need additional constructs for subtyping. Is-A-relationships are represented directly if they are subtyping relationships. The ER model is often criticized for the unconstrained use of types instead of other types [Codd90],[Nijssen88]. The help tool of $(DB)^2$ can be used for the solution of the problem whether a given concept should be represented by a relationship type or by an entity type or by an attribute. Since this decision depends from the application area no general method exist to get this distinction only from the model information. By answering questions on the meaning of the concept the designer can decide to which kind the given concept belongs.

Database design is at present considered as top-down design. This approach is only useful for single-designer support systems. However, in the case the designer can detect only in a restricted manner similarities between the concepts. This is especially useful if the design is made by several designers. In this case the view integration is one of the most difficult problems which has to be solved for an efficient management of the whole database system. This task can not be solved using only methods of top-down design. The window manager approach in $(DB)^2$ can be used to detect similarities. Moreover, designing the process information at the same time adds the possibility to adapt the structure to the process efficiency. For this reason, new normalization algorithms are used in $(DB)^2$ which prohibit normalization if this contradicts the process efficiency.

Figure 2 shows the dependence graph for the design information. The designer can move from one design part to the other design part according to his skills and his understanding of the design process and the application area. This freedom in choosing the personal design methodology is further supported in RAD [Bachmann90] by a special user adaption tool.

In [Yaseen89],[Thalheim91],[Thalheim92] another new design methodology was developed: Design-by-units. Most of the well-known design methodologies are based on the relational approach. But each of the database models should have its own methodology. It is surprising that most of the proposed models do not have its own design methodology. If the model is getting richer in construct the methodology should be deepen. One of the database models with its own methodology is the ER model. However, there is still a little agreement in which cases objects from the real world should be modelled by attributes, entities or relationships. A part of the problems of view integration is caused by this modelling problem. And this contradicts the belief of experienced database designers. Those assume that the views of an enterprise can be integrated since there is an internal integration in the enterprise. The reason for this mismatch is that
Methodologies are not supporting abstraction in an efficient manner. The new design methodology can be understood as a step towards a well-integrated methodology.

The proposed methodology is truly object-oriented and at the same time also theoretically based what supports the implementability. This methodology support also extensibility since using this methodology an existing design and implementation can be extended without introducing changes to it. It promotes reuse and inheritance as well as behavioral extension. To some extend, this approach is similar to modular design known in software engineering. The orientation is different only. We are first interested in the data representation part and then in the processing part since a part of the processing part is based on generic operations which are defined according to the structure.

This approach has further other advantages: it is easier to detect similarities among design units and to reuse parts of design units in other units; changes to the scheme and to parts of units are directly reflected in all other units which are using the changed. The new methodology supports directly the distinction between kernel and dependent object discussed in [Thalheim91]. This is especially useful, if abbreviation techniques [Sciore91] are used in query forms [Yaseen89]. It is a recursive method. This methodology is not following the classical waterfall model with iterations but rather supporting a high level inside-out-strategy [Batini92]. Experience in utilization of \((DB)^2\) has shown that this methodology was the most often choosen for practical design.

**Design-by-units**

1. **Basic step.**
   Design the types for the independent kernel object classes.

2. **Recursion step.**
   Repeat until the schema is not changed.
   Either reification:
   - Refine the units introducing subtypes (in HERM represented by unary relationship types).
   - Refine the relationship types according to the subtypes.
or generalization of units:

- If there are associations among units then introduce a new unit containing the associated units and
  the relationship type according to the association.
- Add other relationship types if there exist new associations.

At present the system $(DB)^2$ is widely used for database design. There is a large number of active
groups which are using the system. Since this system was developed as a prototype several extensions and
improvements are necessary [Bachmann90] and to be developed in the RAD system [Bachmann90]. This
tools provide also a better understanding of design task like abstraction (e.g. analysis of concepts), editing (e.g.
correctness, consistency, complexity), refinement (e.g. context representation) and transformation (e.g.
prototyping, population or test data generation).

### 3.3 An Example

Now we demonstrate the utilization of the $(DB)^2$-system. Firstly, an example is proposed. Secondly, the
scheme is to be design in the frame-based fill-in-form language. The system generates the graphical representa-
tion. The mode of the representation (with/without semantics, in detail/without details) can be choosen.
During the last step, the design is to be translated to the language of a database management system taking
into account the specific properties of the DBMS.

#### 3.3.1 The University Example

Let us design a HERM scheme for a simple university application covering the following information:

1. A catalogue of persons working or studying in the university. A person has a social security number or
   person’s number which uniquely identifies this person. Persons have a name (first and last names and
titles), and an addresses (with a town, a zip, and a street).

2. A catalogue of students which are characterized by their students numbers. A student is also a person.
   Students have a major and a minor specialization. They are supervised by a professor.

3. A catalogue of professors (with their specialisation). Professors are persons. Each professor is associated
   with a department.

4. A catalogue of courses, given in the university and characterized by a unique course number, and names.
   A course can have different prerequisites.

5. A catalogue of course offerings with the course, the semester (year and season), the room (room number
   and building) and the professor.

6. A catalogue of projects characterized by a unique project number, the begin and the end, and the title,
   further their project leaders and team members.

7. A catalog of students activities. Students can enroll a certain course in a term. They obtain a final
   grade.

#### 3.3.2 The Structural Description

This example can be modeled by the following entity and relationship types:

\[
\begin{align*}
Person & = \{ \text{Person's number}, \text{Name} (\text{LastName}, \text{FirstName}, \{ \text{Title} \}), \\
& \quad \text{Address} (\text{Zip}, \text{Town}, \text{Street} (\text{Name}, \text{Nr})) \} , \{ \text{Person's number} \} , \\
\text{Course} & = \{ \text{CNu}, \text{CName} \} , \{ \text{CNu} \} , \\
\text{Project} & = \{ \text{Num}, \text{Begin}, \text{End}, \text{PName} \} , \{ \text{Num} \} , \\
\text{Room} & = \{ \text{Nr,Building} \} , \{ \text{Nr,Building} \} , \\
\text{Student} & = \{ \text{Person}, \{ \text{StudNr} \} \},
\end{align*}
\]
Professor = ( Person, { Speciality } ),
Department = ( { DName, Director, Phones { Phones } } , { DName } ),
Major = ( Student, Department, ∅ ),
Minor = ( Student, Department, ∅ ),
In = ( Professor, Department , ∅ ),
HasProject = (Project, Leads:Professor, Professor + Person , ∅ ),
Prerequis = (Course, Course, ∅ ),
Supervisor = ( Student, Professor, { Since } ),
Lecture = (Professor, Course, Room, Semester, { Time( Day, Hour )}),
Semester = ( { Year, Season } , { Year, Season } ),
Enroll = (Student, Lecture, { Result } ).

Using this structure we get in the system (DB)^2 entity-relationship diagrams. Figure 3 represents the dia-
gram without additional integrity constraints like cardinality constraints, exclusion and inclusion dependencies
and generalized functional dependencies.

The example shows that the proposed extension of the HERM solves some of the drawbacks of the re-
lation and the entity-relational database modeling like uniformity of relations, view definition, problems in
deterministic normalization algorithms, problems in query languages, set inadequacy, sequence modeling.
This model maintains also the advantages of the relational and the entity-relationship model like availability
of database design theory, query language, access path propagation, optimization, user-friendliness, visibility,
understandability, and data dependency propagation.

3.3.3 Semantical and Behavior Description

The HERM design includes the description of semantics and behavior as well. In (DB)^2 the designer can use
(generalized) cardinality constraints, (path) functional, (path) inclusion, (path) exclusion and multivalued
dependencies. Further, he can specify synonym and homonym relationships. These are used for the simpli-
fication of translations. Using the specified constraints and the constraints given by the structure, (DB)^2 is
able to generate the generic functions Insert, Delete and Update with pre- and postconditions for each type.

Some useful integrity constraints in the university environment are the following:
Student.StudNr → Person.Person’snumber;
→ Enroll.Lecture.Professor, Enroll.Lecture.Room,
Student ⊥ Professor ;

The first constraint expresses that the student number is determining the number of this person. The
second functional dependency is expressed also by cardinality constraints. The presented exclusion dependen-
cies states that students and professors are different persons. But there can be expressed also other integrity
constraints. Path dependencies are such constraints. In the university, a course enrollment may be forbidden
if a student did not take all the prerequisites of a course. This can be expressed by the following path depen-
dency:
Student-Enroll-Lecture-Course-1-Prerequis-2-Course:
Enroll.Student.StudNr, Prerequis.(Course,2).CNu
⊆ Student-Enroll-Lecture-Course:

Another path dependency expresses that the minor and the major department of students should be dif-
ferent: Student-Minor-Department ≠ Student-Major-Department.
The constraint that a student can enroll a course only once in a term is expressed by the path functional
dependency
Figure 3: \((DB)^2\)-Diagram of the University Database
Student-Enroll-Lecture-Course:

\[ Enroll.Student, Enroll.Lecture.Semester(Year,Season), Enroll.Lecture.Course \]
\[ \rightarrow Enroll.Lecture. \]

The first path inclusion dependency restricts the use of generic functions. If the user specifies that the maintenance of the type Enroll should be not influenced by other types like Course, Prerequis or if the maintenance has to be simple then the Insert and Update operations are defined for Prerequis only for those courses which are not used in Enroll.

The user can overwrite the generic operations by his own operations. One example would be for instance the Insert-operation for the relationship HasProject.

\[ Insert(HasProject, \langle pro_num,leads_pers_num,member_{<pers_num}> \rangle) \]

where the last argument expects a non-empty list of members.

In the same manner, user-specific operations are specified. Since static integrity constraints can be translated to dynamic integrity constraints, the behavior of each operation is also described by these dynamic integrity constraints.

Further, the following integrity constraints can be modeled. If a person’s first name can not be changed then this can be modeled by the following update restriction:

\[ \forall p \in Person: p.Name.First = f \rightarrow always(p.Name.First = f) \]
\[ ( No_Update(Person.Name.First) ) . \]

The constraint that the result of student’s enrollment does not disappear once the student finished the course with success is modeled by the following transition constraint which limits the applicability of the delete operation on enrollment:

\[ Student - Enroll - Lecture - Course Result_{\geq D} [Student,Course] \subseteq always \]
\[ Student - Enroll - Lecture - Course Result_{\geq D} [Student,Course] . \]

The following future-related constraint states that if once a student has choosen a department for his major (s)he will eventually have a supervisor.

\[ Major-Student [Student] \subseteq sometimes Supervisor-Student [Student] \]

This constraint has the meaning of the more complex formula of temporal logic:

\[ \forall s \in Student \forall maj \in Major \exists sup \in Supervisor : maj.Student = s \rightarrow sometimes(sup.Student = s) . \]

### 3.3.4 The Relational Representation

The system supports the translation of the schemes, the semantics and the operations to specified types of database management systems. Since each of the known database systems has several implementational restrictions which have to be considered during the translation process and this can be done automatically the system \((DB)^2\) computes using the input the corresponding schemes.

Any entity-relationship database can be represented by a collection of relations. For each entity set and for each relationship set in the database, there is a unique relation which is assigned the name of the corresponding type. Each relation type has a number of columns which, again, have unique names. The semantics of the scheme has to be translated together with the scheme. For instance, the entity-relationship scheme contains implicitly also inclusion dependencies. These are important especially for the maintenance of the scheme. If a relationship is embedded in a scheme then this relation scheme contains a foreign key. The translations used in [Teorey89] are incomplete. The complete translation theory is developed in [Thalheim91].

The following translation demonstrates the translation for systems which supports views like Ingres 6.3. During translation we obtain the following relation schemes:

- **Person** = ( \{ Person’s number, Name(First,Fam,\{Title\}), Adr(Zip,Town,Street(\{Name,Nr\})) \}, key = \{ Person’s number \} ),
- **Course** = ( \{ CNu,CName \}, key = \{ CNu \} ),
- **Project** = ( \{ Num, Begin, End, PName, HasProject.Leads:Professor.Person’snumber\}, key = \{ Num \} ),
HasProject = ( {Project.Num, Person.Person's number},
key = {Project.Num, Person.Person's number} )

Room = ( { Nr, Building }, key = { Nr, Building } ),

Student = ( { StudNr, Person's number, Minor, Major }, key = { StudNr } ),

Professor = ( { Person's number, Speciality }, key = { Person's number } ),

Prerequis = ( { (Course,1). CNu, (Course,2). CNu },
key = { (Course,1). CNu, (Course,2). CNu } ),

Lecture = ( { Professor. Person's number, Course. CNu, Semester(Year,Season),
Room. Nr, Room. Building, Time(Day,Hour) },
key = { Professor. Person's number, Course. CNu, Semester(Year,Season) } ),

Enroll = ( { Student. StudNr, Professor. Person's number, Course. CNu,
Semester(Year,Season), Result },
key = { Student. StudNr, Course. CNu, Semester(Year,Season) } ).

Furthermore, in the translation process we obtain other dependencies like inclusion dependencies, exclusion dependencies, and functional dependencies [ElMasri85]. Using the theory [Thalheim92], the translated schemes are automatically simplified. For instance, the attribute identifiers are simplified by synonyms.

On this basis, maintenance mechanisms can be generated for the relational translations.

It should be noticed that during relational translation several integrity constraints and behavioral constraints known in the original design are 'lost'. They can not be expressed in several implementations. This property is the reason for the non-retranslatability of such relational translations to other models. However, using the original HERM design the translation between different translations is possible.

### 3.3.5 Representation in Other Models

Other database models use different type systems. For instance, the implementation of network models is based on lists and pointers. Therefore the translation to a network model should be based on these concepts. Since pointers are restricted by existence constraints the hierarchical structure of the database scheme is computed first. For each relationship type which is not one-to-one another type is created by cartesian products. Then the new types and the kernel object types are connected to the central System type. In the final step, some integrity constraints and the operations are generated. In the discussed example, the kernel types are connected to the System type. E.g., using additional labels we obtain the type

\[ \text{Course} = \langle \text{CNu, CName, Offering: Lecture, Requires: Prerequis, Required: Prerequis} \rangle \]

and additional integrity constraints like the following path constraint

\[ \text{Course} - \text{Requires} - \text{Prerequis} - \text{Required} - \text{Course} : \]
\[ \{ \text{Requires.Course, Required.Course} \} \rightarrow \{ \text{Prerequis} \} \]

which expresses the set property of the list (If a course is a prerequisite of another course then this prerequisite is determined by the two courses.). The two system types

\[ \text{Department} = \langle \text{Major: Student, Minor: Student, Phone\{Phones\}, ..., Worker: In} \rangle \]
\[ \text{Professor} = \langle \text{Name\{\ldots\}, Adr\{\ldots\}, Person's number, Speciality, Works: In, Gives: Lecture} \rangle \]

are connected via the type In.

Since the HERM representation allows a different representation of hierarchies the system \((DB)^2\) can switch between these representations. It uses these representations for translations.

The translation of the design to object-oriented models can be defined in a similar manner. The system \((DB)^2\) is translating HERM-schemes to O2-schemes. This translation depends on the type of system we are using. If for instance pointers are unique for elements then integrity constraints like the above mentioned are added.
4 Conclusion

The goal of database modeling is to design an efficient and appropriate database. Some important criteria are performance, integrity, understandability, and extensibility. We have developed an extension of the entity-relationship model. Based on this extension a new approach to database design has been developed which is effective in meeting these goals. Based on the theory and the methodology the design system \((DB)^2\) was developed. This approach shows that a strong theory can be developed and applied for important practical problems. The history of database management systems demonstrates that a lacking theoretical basis leads to poor and difficult to apply technologies. The presented model and system have the following advantages:

1. The model is easy understandable, simple and perceivable.
   - The model can be used as a basis of database design tools [Thalheim89],[Thalheim91]. The system \((DB)^2\) is used at present by more than 100 user groups.
   - The modeling approach is simple to use even for large problems.
   - Since the model uses graphical representations the modeling result is easier to understand and visible.
   - In an experiment, 20 novice or end-user database designers learned the HERM methodology and later designed different database schemes in different areas. Our experience was that that the methodology was easily accepted, led to more accurate, to less redundant schemes and to a correct modeling of complex relationships.
   - Using query forms [Yaseen89] the user can specify application transactions and programs on the top of the HERM which reduces substantially the complexity of application programming.

2. The results of the design are much simpler than in other approaches.
   - We have used the the model for modeling also some more complex applications. One observation is that the obtained schemes are from three to five times simpler than those obtained by other models. The example of [Teorey89] is simplified by four times and can be placed on one page or one screen. In other examples, the simplification makes it possible to find a model. Using this modeling approach, an airport counter application was modelled by less than 40 entity types and less than 120 relationship types whereas the original solution with more than 150 entity types and more than 400 relationship types was unacceptable by users because of complexity and non-transparency.
   - The simplification leads also to a better understanding of the application and makes normalization easier to perceive.
   - The schemes avoid additional redundancy. Using HERM, the normalization and the minimalization of schemes can be considered together.

3. The utilization of \((DB)^2\) is more natural and simpler. Only necessary facts are to be expressed.
   - The model supports a direct translation to the three classical database models. This translation preserves normal forms. Since a direct translation to relational, network and hierarchical schemes can be used the design decisions directly could be used to obtain schemes in normal forms. The translation theory can be used for a multimodel- and multisystem-support [Yaseen89] and presents a practical solution to interoperability of systems.
   - The HERM algebra is used for query definition. The corresponding relational, network or hierarchical queries can be automatically generated.
   - The model supports a rich set of constraints. These constraints are used for the development of the scheme equivalence. Although the excessive number of fact-encoding mechanisms means that the same semantic unit can be declared in many syntactically different and compatible ways, the information described is equivalent. This equivalence theory can be used for automatic modification of schemes [Bachmann90].
The database maintenance procedures can be derived using the design information.

Using a knowledge base previous and system-provided design decisions can be reused or partially reused what simplifies the design task. Furthermore, similarities in the design can be detected and used for simplification of the implementation.

Using the whole design information the retranslation of application programs can be used for the adaption of existing database systems to changing environments.

4. The model and the design system have a strong theoretical basis.

- The model is based on a multitype logic which is equivalent to the first-order predicate logic. For this reason, results known from discrete mathematics and relational theory [Thalheim91] can be used.
- The model covers the complete modeling information. The structure, static semantics, generic and user-specified operations and behavior of an application can be described by the model.
- The theory is simplified and cleaned up. Sequences, subsets and powersets of objects can be modeled directly. Is-A-Relationships are treated in a better way. Weak entity types can be avoided completely. A normal form theory is developed for the HERM. Using this normal form theory, we can obtain normalized schemes like in the classical theory.

5. The theory is applicable to practical needs.

- Based on the theory a multi-paradigm, robust design methodology is developed [Thalheim91'], [Yaseen89] which encorporates approaches known in object-oriented modeling [Schewe91], modular programming [Thalheim89] and programming in large.
- Using this modeling approach, a view cooperation concepts was developed. Since full view integration is not decidable and not axiomatizable view cooperation is the only applicable approach.
- The approach can be used for reverse engineering. Systems and programs developed for one management system can be recompiled and adapted to other management systems.

References


The Tool Box RADD (Rapid Application and Database Design)
Die Entwicklung einer Datenbankentwurfsumgebung der dritten Generation:
RADD
Rapid Application and Database Development *

Bernhard Thalheim†
BTU Cottbus
Institut für Informatik
Postfach 101344
D-03013 Cottbus

Meike Albrecht‡
BTU Cottbus
Institut für Informatik
Postfach 101344
D-03013 Cottbus

Zusammenfassung


1 Einführung

Der Datenbankentwurf basiert auf einem oder mehreren Datenbankmodellen zur Modellierung der Anwendung. Im Laufe des letzten Jahrzehntes haben sich semantische Datenbankmodelle als ausreichendes Mittel zur Modellierung von komplexen Anwendungen erwiesen. Dabei wurde insbesondere das Entity-Relationship-
Modell benutzt. Wir legen unseren Arbeiten ein erweitertes, theoretisch-wohlbasiertes Entity-Relationship-Modell (HERM) zugrunde, mit dessen Hilfe sowohl die strukturelle, als auch die semantische und operationale Information über das Anwendungsgebiet dargestellt und betrachtet werden kann.


Im folgenden werden die ersten implementierten Komponenten des Systemes vorgestellt. Abbildung 1 zeigt die Architektur der RADD Toolbox mit Steuerfluss (dünne Linien) und Datenfluß.

2 RADD-Komponenten

Die Vorarbeiten zu semantischen Datenmodellen, zum System (DB) 2 und zum Datenbankentwurf werden in den Rostocker und Cottbuser Arbeitsgruppen zu folgenden Teilkomplexen fortgeführt:

- Modellierung von Struktur, Semantik und Verhalten einer Datenbank
- Nutzermodellierung und Nutzerberatung
- Modellierung von statischer und dynamischer Semantik in Datenbanken
- Natürlichsprachiges Interface für den Entwurf
- Bewertung und Optimierung des Verhaltens
- Theorie semantischer Datenmodelle
- Grundlagen des Entwurfs von (verteilten) Datenbanken

2.1 Graphisches Interface

Es ist allgemein akzeptiert, daß graphische Darstellungsmittel im Entwurfsprozeß unerläßliche Hilfsmittel vor allem für die Kommunikation mit den Entwerfern sind. Für das semantische Datenmodell wurden bisher Ansätze entwickelt, die die Struktur einer Klassendiagramm (Repräsentationsgraph), in Zusammenhang mit anderen Klassen (Referenzgraph) und unter Weglassung von Details (Identifikationsgraph) abbilden.
Fig. 1. Architektur von DB² - RADD - CODE


Es wurden Module geschaffen, die z. B. die in der AG entwickelte modulare Entwurfsmethodik Design-by-Units effizient unterstützt.

Für das Projekt RADD wurde das data dictionary für den Datenbankentwurf ausgearbeitet. Es wird ebenso wie die Grundkomponenten der GraphEd-Erweiterung von allen beteiligten Partnern benutzt. Außerdem wurde die Kommunikationsoberfläche geschaffen, mit der sowohl der Entwurf in verschiedenen Strategien, verschiedenen Teilen der Anwendung, verschiedenen Darstellungen als auch unterschiedlicher Detailierung möglich ist.

2.2 Akquisition von Integritätsbedingungen


Die formale Angabe semantischer Constraints ist eine der schwierigsten Aufgaben im Datenbankentwurf. In dem hier vorgestellten Zugang werden die semantischen Constraints auf die im Datenbankentwurf am häufigsten verwendeten Constraints (Schlüsse, funktionale Abhängigkeiten, Inklusions- und Exklusions-abhängigkeiten, Kardinalitäten) eingeschränkt. Die Schwierigkeit der Erfahrung dieser Constraints ist auf folgende Probleme zurückführbar:


- Semantische Constraints, z. B. Kardinalitäten, können auf verschiedene Weise interpretiert werden. Arbeiten mehrere Entwerfer an einem Datenbankentwurf, so muß Einigung über die Interpretation semantischer Constraints bestehen. Ebenso muß bei Entwürfen, an denen über einen langen Zeitraum gearbeitet wird, eine Interpretation der Semantik durchgängig verwendet werden.

- Fast alle Entwurfsmethoden erwarten von den Entwerfern, daß sie eine vollständige Angabe der Semantik. Normalisierungsalgorithmen, z. B. „hier nur dann eine korrekte Normalform, wenn die Menge der funktionalen (und mehrwertigen) Abhängigkeiten vollständig angegeben wurde und alle nicht angegebenen Abhängigkeiten sich entweder aus der angegebenen Menge ableiten läßt oder nicht gelten.

Oft werden aber einige semantische Constraints vergessen, obwohl sie offensichtlich sind, oder semantische Constraints sind zu kompliziert, besonders, wenn sie über mehreren Attributen definiert sind, und werden deshalb entweder gar nicht oder falsch angegeben.

Wünschenswert wäre also eine Unterstützung bei der Semantikakquisition, bei der dem Entwerfer semantische Constraints vorgeschlagen werden. Bei der Unterstützung der Suche nach semantischen Constraints treten weitere Probleme auf:

- Insbesondere kompliziertesemantische Constraints, die über mehreren Attributen definiert sind, sind schwer zu entscheiden. Um ungeübte Entwerfer diese Aufgabe zu erleichtern, ist es günstig, wenn diese Unterstützung informal erfolgt.

- Es müssen potentiell exponentiell viele mögliche semantische Constraints untersucht werden, um die vollständige Menge semantischer Constraints zu finden.

Einer Forderung nach Vollständigkeit ist deshalb nur bei kleinen Schemata und kleinen Relationen nachzukommen.

Als Lösung dieser Probleme bieten sich zwei Ansätze an:
1. Eine starke Einschränkung des Suchraumes kann durch einen intelligenten Zugang, der Heuristiken zur Beschleunigung der Suche verwendet, erreicht werden.


Das Problem der Semantikerfrage ist NP-vollständig. Es müssen also Heuristiken zur Suche nach Schlüsseln, unabhängigen Bereichen und funktionalen Abhängigkeiten verwendet werden. Diese zeichnen Schlußfolgerungen aus der Struktur der Relation (Attributnamen,-typen, Anzahl der verschiedenen Werte, die ein Attribut annehmen kann, Reihenfolge der Attribute), aus den Beispielen sowie aus den bereits ermittelten Constraints.

Ebenso werden Algorithmen des schnellen Abstiegs einbezogen, es werden zuerst Constraints erfragt, aus denen sich viele weitere ableiten lassen.

Eine weitere wichtige Heuristik ist die Arbeit mit Attributgruppen, wobei in der Relation Attribute gesucht wird, die ähnliche Eigenschaften aufweisen. Für diese können dann die Constraints im Zusammenhang erfragt werden.

Der Dialog wird wissensbasiert gesteuert, es wird versucht, die Methoden zur Suche nach Constraints so anzuwenden, daß effizient Wissen über die Semantik erfragt wird.

In analoger Weise lassen sich Informationen über Inklusions- und Exklusionsabhängigkeiten, sowie Kardinalitäten erfragen. Auch für die Erfassung dieser Constraints können Beispiele verwendet werden. Viele der verwendeten Heuristiken sind dabei übertragbar. Dieser Programmteil befindet sich jedoch noch in der Entwicklung.

2.3 Natürlichsprachiges Interface


Der Entwerfer einer Datenbank kann über die Analyse natürlichsprachiger Eingaben erstellt werden. Aus natürlichsprachlichen Sätzen, die ein Anwendungsgebiet beschreiben, lassen sich Informationen über die Struktur der Datenbank und Kandidaten für die Semantik und das Verhalten ableiten.


zu ermitteln.

Die Kandidaten für semantische Constraints werden gewichtet, um zuerst die Kandidaten zu validieren, die am plausibelsten erscheinen. Im Dialog mit dem Entwerfer kann dann anhand von Beispielen eine Validierung der Kandidaten erfolgen. Da alle erkannten Kandidaten, die ja vage sind, validiert werden, liefert dieses Vorgehen eine Menge richtiger Constraints.

Die einzelnen Komponenten der Methode werden in den folgenden Abschnitten ausführlich beschrieben.


Die Semantik und das Verhalten beim Datenbankentwurf kann man nicht losgelöst von strukturellen Informationen betrachten. Gerade in natürlichsprachigen Äußerungen tritt eine starke Verquickung der drei Elemente auf. Die Interpretation der Entwurfsaussagen, d.h. die konzeptionelle Semantik, stellt deshalb das Kernstück der natürlichsprachigen Komponente dar.

2.4 Benutzerberatung und Entwurfsstrategien


- den Fähigkeiten, Fertigkeiten und der Ausbildung des Entwerfers,
- dem zugrundegelegten Entwurfsmodell,
- der Kompliziertheit der Anwendung, insbesondere der Strukturierbarkeit, Modulierbarkeit und der Komplexität der Semantik und der Operationen, und
- der Größe der Entwicklergruppe.


wurfsprozeß als einen Prozeß der Revision und Verfeinerung auffassen.


- Komposition: Für einen Typenkonstruktor und Komponententypen wird ein neuer Typ generiert.
- Dekomposition: Ein Typ wird mittels eines Typenstruktors in neue Typen zerlegt.
- Erweiterung: Es wird ein neuer Typ konstruiert.
- Hinzu kommen noch Steueroperationen und Hilfsoperationen.

2.5 Mechanismen zur Adaption an den Benutzer


Die Auswahlkriterien für eine Entwurfstrategie beeinflussen sich untereinander. Einerseits beeinflußt die Größe der Anwendung die Größe der Entwicklergruppe. Andererseits verfügt eine Gruppe mit wenig Entwicklern, die aber viele gemeinsame Erfahrungen und Fachkenntnisse in einem festen Entwurfsrahmen hat, über genügend Fähigkeiten, große und komplexe Anwendungen zu entwickeln. Erfahrungen im Datenbankentwurf mit speziellen Entwurfsmustern können für einen Benutzer von Vorteil oder von Nachteil sein. Das Entwurfsbewusstsein des Benutzers ist Bestandteil der Benutzercharakteristik und ist wie folgt unterteilt:

- positive Entwurfsbewusstseine bezgl. der Anwendung
  - konzeptuelle Entwurfsprinzipen
  - von Entwurfsstrategien
  - unterschiedlicher Datenbankentwurfsmodelle
- negative Entwurfsbewusstseine
  - in der Erfahrung und Fehlerbeseitigung
  - in der Entwurfsplanung
  - bei Entwurfsentscheidungen
  - bei der Entwurfsbewertung
  - bei der Konstruktion eines Entwurfes
  - in der Konzeptualisierung (Abstraktions-techniken)
  - bzgl. Effektivität des Entwurfes
  - bzgl. Erweiterbarkeit des Entwurfes
  - bzgl. Kooperation
Die Charakteristik eines Benutzers ist durch ein explizites Benutzermodell repräsentiert. Die Akquisition der Benutzercharakteristik kann explizit vollständig in einem Dialog erfolgen oder der Benutzer wählt ein vorgegebenes stereotypisches Benutzermodell und ändert eventuell einzelne Werte bzgl. der Benutzercharakteristik.


### 2.6 Bewertung und Optimierung des Verhaltens

Bei der späteren Implementierung der Datenbankstruktur oder aber darauf aufsetzender Applikationen zur Datenein-/ausgabe, wird oftmals eine andere Strategie verfolgt als die, die Typen der externen Benutzersichten (oder Applikationsprogramme) analog zum Schema des unterliegenden Datenmodells abzubilden. Typzusammenhänge des ER-Modells, die ein schlechten Laufzeitverhalten implizieren würden, werden bestenfalls hier erkannt und ummodelliert. Da der ER-Zugang und die verschiedenen logischen Datenmodelle (hierarchisch, Netzwerk, relational und die Objektorientierten) aber unterschiedliche Level der Datenabstraktion und damit der Transformation für die betreffende Abbildung der Dateiorganisation bieten, werden Probleme, die schon auf den konzeptionellen Entwurf zurückgehen, oftmals weder bei der Transformation in ein logisches Datenschema noch bei deren Implementation durch ein DBMS erkannt und behoben. Schlimmstenfalls zeigen also schon auf den Entwurf zurückgehende Probleme erst im laufenden Betrieb ihre Wirkung.


Der gewählte Zugang beruht auf dem Auswertungsmechanismus mittels eines Kostenmodells für Basisoperationen der Typen (insert, delete, update, retrieve) anhand unterschiedlicher Dateiorganisationen (Listen, Heap, Isam, Btree, dBtree, dHashing, ...). Die hoherstehenden Werteste der Entity-Relationship Modells HERM (Tha94) erlaubt die Spezifizierung genesterter Attribute, mehrstufiger und genesterter Relationshiptypen, sowie von Vereinigungstypen (Cluster). HERM basiert 'stark' auf einer Warteschlangen. Als Integritätsbeschrankungen sind insbesondere Schlüssel- und funktionale Abhängigkeiten sowie Inklusions- und Exklusionsbeschränkungen definierbar. Zudem können durch Kardinalitätsbeschränkungen und Pfadabhängigkeiten auch allgemeinere Bedingungen definiert werden, welche ins-
besondere die zulässigen Datenbankpopulationen betreffen.

2.7 Gates zu Datenbanksystemen


3 Theoretische Grundlagen

3.1 Theorie objekt-orientierter Modelle und des Entity-Relationship-Modelles

Das klassische Entity-Relationship-Modell wurde erweitert. Es können komplexere Typen konstruiert werden:


Relationstypen beliebig hoher Ordnung: Beziehungen können nicht nur über Entitätstypen definiert werden, sondern auch über Beziehungen selbst.

Gruppierungen: Verschiedene Typen können zu einer Gruppe mit einem Konstruktor zusammengefaßt werden.


Verallgemeinerungen klassischer relationaler Bedingungen sind verallgemeinerte funktionale, mehrwertige, Verbund-, Inklusions- und Exklusionsabhängigkeiten.

Spezifische Entity-Relationship-Bedingungen sind insbesondere Kardinalitätsbedingungen, Synonym- und Homonymbeziehungen.

Graphische Beziehungen werden über Pfaden und Teilgraphen im Schema definiert.


Es wurde auf diesen theoretischen Grundlagen die klassische Datenbanktheorie erweitert. Verallgemeinerung von Transaktionen, Normalformen, Sichtenintegrationsalgorithmen etc. erlauben, den Entwurf einer Datenbank vollständig im HERM durchzuführen. Damit ist ein Entwerfer nicht mehr gezwungen in mehreren Entwurfssprachen unter Benutzung verschiedener Transformationstechniken den Datenbankentwurf durchzuführen.

Diese Theorie wurde auf objektorientierte Modelle erweitert.

3.2 Entwicklung einer allgemeinen Entwurfstheorie


Entwurfsrichtung: Bottom-up-Entwurf verwendet im wesentlichen Kompositionoperationen, Top-down-Entwurf dagegen Dekompositionsoperationen.

Steuerung des Entwurfs: Inside-out-Strategien gehen von einer Nachbarschaftsfunktion aus, die benutzt wird, um die nächsten zu betrachtenden Konzepte auszuwählen.

3.3 Linguistische Grundlagen


Für die Semantikerfassung können wir eine Reihe von linguistischen/computerlinguistischen Methoden nutzen, die Regeln für die Bedeutungszuordnungen zu natürlichsprachlichen Äußerungen beschreiben.

Prinzip der Kompositionalität. [2, 280] beschreiben das Frege'sche Prinzip als: 'The meaning of the whole sentence is a function of the meaning of the parts.' Die Bedeutung eines Satzes setzt sich somit zusammen aus der Bedeutung der Satzteile. Die Bedeutung dieser wird dann durch die Bedeutung der Subphrasen beschrieben und so weiter; letztlich gilt es, die Bedeutung der Wörter zu untersuchen. Grundlage für die Satzsemantik bildet also eine im Lexikon definierte Wortsemantik.

Betrachten wir dieses Prinzip für unsere Anwendung, also die Interpretation von Datenbanken, so müssen wir klären, welche Satzphrasen eine wichtige Rolle bei der Semantikakquisition spielen und welche Wörter die Basis der...
Phrasen bilden (vgl. auch [2]).

**Bedeutung als Referenz.** Wenn wir das Prinzip der Kompositionality annehmen, dann gilt es auch, die Frage zu erläutern, wie Bedeutungsbeziehungen zwischen Phrasen/Wörtern realisiert werden. Dazu ist eine gewisse Klassifizierung von Wortarten vorzunehmen, um differenziert Beziehungsrelationen aufstellen zu können. Diese Relationen müssen im Lexikon definiert werden. (Zum Beispiel könnte das Verb 'essen' zur Klasse der Tätigkeitenverb gehören, die mit der Substantivklasse Nahrungsmittel korreliert.) Diese Klassifizierungen und Relationen sind nur zum Teil anwendungsunabhängig gestaltbar. Spezielle Informationen zum Domaine sind für alle Anwendungen notwendig. In diesem Sinne wird die Bedeutungsbeschreibung auch als 'feature instantiation' beschrieben.

Gewisse Teile der Fachsprachen sowie Teile der Umgangssprache allgemein zeichnen sich durch bestimmte über die Anwendungsgebiete hinausgehende pragmatische Eigenschaften aus. Diese auf Beobachtungen basierenden Eigenschaften betreffen insbesondere gewisse Lesarten von Wörtern/Sätzen die dem Nutzer plausibler oder 'auf den ersten Blick logischer' erscheinen. Zum Beispiel wird mit dem Wort 'mehrere' eine Menge von Elementen, die größer als zwei sind beschrieben; selten ist eine Menge mit zwei Elementen gemeint.

Die Analyse basiert auf einer Klassifizierung und einer Konzeptualisierung von Elementen der natürlichen Sprache:


**Quantoren.** Quantoren bezeichnen i. A. Kardinalitäten. Der Quantor *alle* und seine verschiedenen Negationen beschreibt im Zusammenhang mit Kopula Inklusions- und Exklusionsabhängigkeiten.

Die Erfassung von semantischen Informationen hinsichtlich der zu entwerfenden Datenbank stützt sich auf eine Reihe solcher heuristischen, linguistisch motivierten Annahmen, die in die Wissensbasis integriert sind.

### 4 Ausblick


**Danksagung**

Die Entwicklung dieser komplexen Entwurfsumgebung ist nur durch eine größere Gruppe möglich. Wir möchten uns deshalb hier für die Beiträge von Margita Altus,
Edith Buchholz, Haiko Cyriaks, Antje Düsterhöft, Jana Lewerenz, Holger Mehlau, Martin Steeg und Klaus-Dieter Schewe bedanken.

Literatur


[Tha92] B. Thalheim, Design with the database design system (DB2). In Fourth Int. Conference „Putting into practice methods and tools for information system design“ (ed. H. Habrías), Nantes, France, 1992, 155 – 174.


Eine umfangreiche Bibliographie findet man in [Tha93]. Aus Platzgründen verzichten wir hier auf diese.
The Design of RAD: Towards an Interactive Toolbox for Database Design

Peter Bachmann
Bernhard Thalheim*
Abteilung Mathematik
TU Dresden
Mommsenstr. 13
D-O-8027 Dresden, FRG

Walter Oberschelp
Gottfried Vossen†
Lehrstuhl für Angewandte Mathematik,
insbesondere Informatik
RWTH Aachen
Ahornstr. 55
D-W-5100 Aachen, FRG

December 1990

Abstract

The design of RAD is described, which is intended to become a toolbox for the interactive, object-oriented design of databases, and which is being developed within a cooperation between the universities of Rostock, Aachen and Dresden in Germany. The development of RAD will be carried out in two stages:

In the first, a system will be realized whose goal is to support designers of a database in all stages of the database life cycle, beginning with conceptual design and ending in documentation and provisions and support of redesign after modifications. This will be done interactively meaning that users are guided through the various design steps, can get explanations about the systems itself and about what they have designed already, are advised about reasonable design choices, and may consult a “reviewer” who critically inspects the result of a design phase and makes suggestions for modifications if necessary.

In the second phase, the system shall be turned into a genuine toolbox, which on the one hand provides a number of methodologies and tools for database design, but on the other can be configured according to designers’ needs into a system that is customized for a specific application.

This paper describes the ideas underlying the RAD system as well as its various components.

*Permanent affiliation: Fachbereich Informatik, Universität Rostock, Albert-Einstein-Str. 21, D-O-2500 Rostock, FRG
†Current affiliation: Institut für Informatik, Universität Koblenz-Landau, Rheinau 3–4, D-W-5400 Koblenz, FRG
1 Introduction

The design of databases has been a challenging task ever since the arrival of database systems in commercial applications. Due to the complexity of this task especially in large enterprises, computer-aided database design seems a feasible approach; consequently, a number of systems has been developed which assist designers of various levels of database knowledge in mapping a given application to some database system [2, 8, 4, 6, 13]. However, these systems are either based on special meta-languages for describing conceptual aspects of an application which are difficult or inappropriate to learn, like DATAID [2, 8] or Gambit [6], or consider very specific subaspects of the problem only, like Design-by-Example [13] or RED1 [4]. In addition, these systems are monolithic in the sense that they provide some fixed functionality which cannot be extended, and there is no way to customize them with respect to distinct design applications. In this paper, we report on the design of a new design system, RAD, that is intended to overcome these limitations, and that at the same time goes beyond previous approaches in taking current technology (like graphics on the hardware or object-orientation on the software side) into account.

RAD is based on two basic ideas: First, database design should be a highly interactive process, in which a designer not only uses the system for drawing, say, entity-relationship diagrams on a screen and has them automatically translated into a relational schema; instead, the system should act as a “design workstation” [21] and hence

- provide the user with a reasonable way to describe the syntax, static semantics and behavior of an application, thereby following an object-oriented approach to systems design,

- assist the user in making design choices and in analyzing a (preliminary) design,

- comprise automatic mechanisms for translating a completed design into a specific data model that underlies a given system, and these should not be restricted to the relational model,

- provide means to modify a design even when a corresponding database has been established already.

Second, database design is a task that arises in a variety of applications, like banks, insurance companies or other commercial enterprises, but also in technical domains like CAD or CASE applications; in addition, database design is carried out by people of different backgrounds: specifically trained employees who have a good knowledge of databases, casual users with little a priori knowledge, and people with an “intermediate” degree of expertise. As a consequence, a design system should support

- different levels of user sophistication,

- distinct types of applications and their requirements,

- various hardware systems, ranging from PCs to workstations.
Thus, it does no longer make sense to offer a design system which is perfectly suited for particular types of applications and/or users, but performs poorly for others. The solution, then, is to build a toolbox that is configurable with respect to a variety of aspects, an idea which is similar in spirit to that underlying the ODDESSY project reported in [9, 10].

In this paper, we lay the foundations for such a toolbox approach by first discussing, in Section 2, the preliminary design of RAD and its various components. RAD is based on a prototype implementation [17] that has been carried out at the University of Kuwait, which in turn is based on an extended entity-relationship methodology. In Section 3, we describe the steps a designer may go through when using HERM+, the data model on which RAD is based. Finally, in Section 4 we describe an extension of RAD into the intended toolbox; specifically, we explain how the system has to be modified so that several parts are generated in response to the desire of applying RAD within a given application and users.

2 RAD Architecture

2.1 Overview

In this section we describe the current plans for the architecture of RAD, in particular the components it will consist of, their intended purposes, subcomponents, and how they are intended to interact with each other and the outside world. Figure 1 shows the architecture of RAD, where thin lines represent control flow and thick ones data flow.

Basically, the first phase of the project will result in a system consisting of four major components:

1. HERM+, the interface to the outside world, in particular to the designer and the application he or she wants to design. It will provide a language based on an extension of the entity-relationship model for defining syntactical structures, static semantics and behavior of a given application and will hence support conceptual database design based on an object-oriented methodology [5, 11, 12].

2. The Support System, a component to support users of the system during the various phases of a design process. It has two subcomponents: The User’s Guide will comprise a tutoring system for guiding the user through object-oriented design in general as well as through HERM+ and the database design process in particular; it will also support him or her in choosing design strategies, and provide examples. The Analyzer will allow a designer to get online feedback regarding design decisions, and critically review the result of a design.

3. The Translator, a component for translating the result of some database design written in the HERM+ language into the language of a specific database system that is to be used in the given application. Specifically, it is intended to have the Translator support a variety of concrete data models, including the relational, nested relational, network, hierarchical and complex object models. Additionally,
Figure 1: Architecture of RAD.
it should contain a component for performing translations between any two of these models.

4. The \textit{Modifier}, a subsystem which can be incorporated for modifying a given, already completed design or for altering one or more design decisions. To this end, it will consist of means to modify syntactical structures, static semantics as well as behavior, will use a \textit{Normalizer} for performing normalization on parts of a design or an entire design according to user needs, and will have a \textit{Redesigner} for generating the input for the Translator in case a design has been modified.

The following subsections elaborate further on the motivations behind and the planned functionality of these major RAD components.

2.2 HERM+

In this subsection, we elaborate on the HERM+ component of RAD and describe its interfaces with the other components and the outside world (including designer and application).

Basically, HERM+ will be based on the Higher-Order Entity-Relationship Model (HERM) introduced in [17]; it will go beyond HERM in allowing to model behavior of data objects in addition to their syntactical structure and static semantics. In essence, HERM+ will incorporate an object-oriented approach to database design (while HERM lacks the ability of describing behavioral aspects). Consequently, static semantics as well as behavior will be based on an internal algebraic background, i.e., the meaning of structure is expressed by means of a term algebra. This will allow to use mechanisms like term rewriting and narrowing for deduction of information [1]. In detail, HERM+ will support the following design declarations:

1. \textit{Syntactical structure}, described in terms of
   - simple or complex attributes with corresponding domains, where the definition of an attribute is allowed to use set and/or tuple constructors,
   - purely value-based as well as identity-based entities and relationships, where the former will, for example, be supported by reusable type declarations,
   - entity and/or relationship versions which allow the capturing of a notion of time in a database.

2. \textit{Static semantics} of an application, expressed as domain and integrity constraints, like
   - subdomain constraints,
   - domain ordering and compatibility,
   - relational dependencies (functional, inclusion, exclusion, multivalued),
   - special entity-relationship dependencies (paths, disjunctive etc.).
3. **Behavior** of entities, relationships and other parts of the syntactical structure, described in terms of

- implicit operations, i.e., operations “built-in” to the meta-model (like projection or general navigation),
- explicit, i.e., user-defined operations, which might either be expressed in terms of implicit operations, or defined by means of ADT mechanisms (using pre- and postconditions); explicit operations express behavior which is specific to certain classes of objects,
- transactions, i.e., “programs” consisting of implicit and/or explicit operations which form an atomic unit.

For structural declarations, the user of HERM+ will be able to choose between a textual and a graphical mode; for both, HERM+ will provide specific meta-languages, and in textual mode there will be a choice between as “ordinary” declaration language and a menu-based one. Each declaration is documented in a Declarations Dictionary, as is the result of an entire design. A design process is guided by the Support System described below, which interacts with HERM+ regarding information exchange. Upon completion of a design, HERM+ calls the Translator for translating the design into the specific data model that underlies the DBMS at hand, and supplies it with a complete formal description of the application. Finally, it will be possible to modify a given design at any time, according to changing user needs, systems or other aspects, by incorporating the Modifier. The latter is supplied with the relevant information and returns a new version of the given design.

### 2.3 The Declaration Language

We now outline the declaration language through which a user can directly define entity or relationship types, in a way that is loosely related to what can be done in SQL w.r.t. structural aspects, but which additionally includes more sophisticated possibilities than SQL regarding static semantics, as well as new ones regarding behavior (recall that the underlying idea is to support an object-oriented design paradigm).

First, a name has to be introduced for any newly defined object. This is done via an appropriate version of the following command:

```sql
CREATE { TYPE | ENTITY | RELATIONSHIP } name-1 [, name-2 ... ]
END CREATE
```

Next, it is possible to bind a definition of structure, static semantics, and behavior to a given name in separate steps. For type declarations, already existing ones can be reused in other definitions; type structures may involve ordinary constructors like “tuple” and “set” as well as basic types (“int”, “string” etc.), but may additionally involve ADT facilities. The general syntax of the “binding command” is the following:
CREATE { STRUCTURE | SEMANTICS | BEHAVIOUR }
FOR name
AS definition
END CREATE

2.4 The User’s Guide of the Support System

We now describe the User’s Guide, a subcomponent of the Support System which is intended to aid and to guide the user in performing a design. It will consist of the following four (independent) subcomponents:

1. The Learning System has to provide help functions for using the system and for doing database design. For example, a user will be able to request information on the metalanguage or the functionality of RAD. In addition, design steps, strategies, notions and notations can be explained upon request.

2. The Explanation Component is intended for providing the user with samples that explain the impact of possible design decisions, and that can be reused in a current design. For example, if the user wants to declare a relationship, this component will be able to present possible relationships which differ with respect to semantics, operations and such. In general, explanations shall be obtainable for any of the HERM+ ingredients. To this end, a Samples Base will be available, which will contain various complete designs (like for the university application, for which the usage of HERM+ is partially explained in Section 3), and from which parts can be incorporated into an application being designed. Based on already provided declarations, the Explanation Component will also be able to generate examples for certain semantic aspects of the application being designed; for example, the system might ask a user whether functional dependencies hold between already known attributes, and explain what this means by a small sample.

3. The User Adaption shall be a component of the User’s Guide capable of recognizing the “profile” of a current user and the way in which he or she is likely to use the system. For example, a user might choose to perform a design in a top-down fashion. This implies that general structures will be declared first and later refined. The system will then be able to recognize this and make proposals for the individual proceeding, and to watch out for possible mistakes. Furthermore, this component will be capable of discovering analogies, like among attribute or entity names.

4. Finally, the Strategy Advisor is particularly meant for the casual user who has little or no prior knowledge of database design. In such a case, it is reasonable that the system appears to him or her as a kind of “design expert” which can make detailed suggestions on how to proceed. This seems particularly useful for designers who come with a background on functional or procedural software design, but with little knowledge on the object-oriented paradigm.
2.5 The Analyzer of the Support System

We now describe the Analyzer, another subcomponent of the Support System which is intended to support tasks like the analysis of a schema declaration with respect to “quality”, the recognition and user notification of things like “dangling” entities or relationships, redundancies, or the determination of role information. It will consist of the following three (independent) subcomponents:

1. The **Reviewer** will act like an interactive optimizer capable, for example, of recognizing redundancies within a design, of simplifying given sets of functional dependencies if possible, or of proposing a denormalization for a given relational-like design based on efficiency considerations. To this end, it will interact with the Normalizer subcomponent of the Modifier (described below), which will be primarily responsible for the reverse process. Another task of the Reviewer will be to determine homonyms as well as synonyms, for which the Language Base will be at his disposal.

2. The **Verifier** will be a proof system which operates by means of term-rewriting and narrowing. It allows to verify first-order formulas and to solve equations. For example, the impact of transactions on the validity of integrity constraints can be checked, or valid interleavings of distinct transactions can be determined prior to execution.

3. The **Estimator** is intended to predict the effects of the translation of a HERM+ design into a concrete data model relative to required and expected efficiency. For example, the length of search paths for typical queries can be estimated (so that, in principle, optimization at this early stage becomes possible).

The information the Analyzer has to return to HERM+ may contain a notification to call the Modifier next, in order to change the current design according to problems encountered during the analysis.

2.6 The Translator

The next major component of RAD is the **Translator**, whose basic task will be to take a given HERM+ design as well as information on the target DBMS as input, and to transform this into a collection of declarations in the language of the corresponding data model. To this end, various target models will be supported, including the flat as well as the nested relational model, the network and the hierarchical model, and one or more complex object models. For each of these, the given syntactical structure, static semantics and behavior will be transformed into structure plus operations. Depending on the target model, for which the view modeling capabilities might differ from one to another, the result might vary with respect to the representation of the input information. The Translator will use both standard translation techniques like the ones described in [20, 14] and new ones.
In addition, the Translator will have as a subcomponent the *Heterogenous Translator*, which will be able to provide support for heterogenous databases by its ability to translate from one target data model directly into another.

### 2.7 The Modifier

The general task of the *Modifier* is to generate different design *versions*, which either might arise directly during a design process or come into existence during the lifetime of a database. This in particular will include the possibility to generate distinct versions of *parts* of a design, for instance in order to learn about the implications of different design choices, which we consider a crucial functionality when an object-oriented methodology is used. In detail, the Modifier will have two major subcomponents:

1. The *Normalizer*, a tool for performing normalization; however, it will not only be based on known relational normalization procedures, but also support similar tasks for other data models. It can be called during a design process as a response to an acknowledgement from the Analyser, or during a redesign phase for schema updating.

2. The *Redesigner*, primarily intended to support schema (structure and operation) modifications that might arise after an initial design has been completed and “gone into operation”. Its output includes, among other things, proposals for physically transforming the original database according to the modified schema.

### 3 A Sample Application of HERM+

In this section, we describe, mostly by way of a sample application, the key functionality of HERM+ and how a user can interact with it. To this end, it should first be noted that HERM+ will not be capable of supporting requirements analysis and specification; it is assumed that users consult other aids in this respect. In later releases, the learning system might be enhanced in that direction.

#### 3.1 Introduction

A user of HERM+ will have a choice of three ways to do database design, which interact with each other through the Declarations Dictionary:

1. Through a menu-based textual subsystem, where syntactical structures, static semantics and behavior can be declared interactively; for example, when a user decides to declare a new “entity” object, the system will ask him or her for a name, attributes (properties), keys etc.

2. Through a graphical interface, which allows to draw entity-relationship-like diagrams on the screen directly, and to augment these with a textual description of their structural, static semantical and behavioral details, by clicking the corresponding selection of a pull-down menu.
3. Through an SQL-like declaration language.

It will be possible to switch between any of these design representations; in particular, a design can be begun in the graphics mode, later be continued in textual mode, and finally be shown in the SQL-like “procedural” mode.

3.2 A University Application

We now exemplify the basic design methodology underlying HERM+ by way of a university application, which can informally be described as follows [19]: A university is populated by persons, which are either students or professors. Furthermore, it is divided into departments, to which students are associated either as minor or as major, and in which are professors. In addition, professors advise students, and work on projects. Courses require prerequisites. Lectures are courses which take place at a certain time in a certain room, and are offered by professors. Finally, students enroll in lectures.

We assume here that the requirements analysis phase has at least resulted in a description like the above, augmented with information on the relevant attributes and their potential integrity constraints; a designer should at this point also have an idea of operational requirements to the database under design.

Clearly, a decision is now needed on what are entity types (“classes”) in the application, and what are their relationships. To this end, we assume that the following decisions have been made already. Entity types are the following:

Person, Department, Course, Room, Project.

(Concrete entities over these types are intended to be “well-distinguishable objects” of the real world.) Next, there exists a number of IS-A relationships in our application, like student IS-A person. According to HERM [17], on which HERM+ is based, these are modelled here as unary first-order relationships. As a result, we obtain the following:

Student, Professor.

Note that, conversely, unary first-order relationships always represent IS-A relationships. Additional relationships in this application are:

Major, Minor, In, Works, Advisor, Lecture, Enroll.

Notice that these latter relationships are second-order or even third-order in the sense of HERM (“higher-order” relationships), meaning that they are relationships between relationships or between entities and relationships. Relationships which refer only to entities have order 1, whereas relationships of order \( n + 1 \) involve at least one relationship of order \( n \), and no other relationships of order higher than \( n \). As has been demonstrated in [17], this is a reasonable and necessary extention of the standard entity-relationship model as described, for example, in [20]; details of this modeling approach can be found in [19].

Finally, entity types as well as relationships (more precisely: relationship types) may have attributes, which in turn can be of simple or complex structure. To this end,
HERM+ supports a tuple as well as a set constructor, as demonstrated by the following example: A person is characterized by a name, an address, and a personnumber. A name is composed of a first and a last name as well as a set of titles, while an address is composed of a zip code, a city and a street, where the latter is a tuple consisting of a name and a number. Similar observations apply to the remaining parts of the example; a complete description is shown in Figure 2.

With respect to static semantics, we note that things like the following can be specified:

1. Functional dependencies, like
   \[ StudNo \rightarrow Student, \]

2. path dependencies, which are dependencies which are defined via a path through the diagram; an example is the dependency which expresses that a student can enroll only in a course for which he or she meets all prerequisites, formally written as
   \[ \text{Student-Enroll-Lecture-Course-1.Prereq-2.Prereq-Course: StudNo, CNo} \]
   \[ \subseteq \text{Student-Enroll-Lecture-Course: StudNo, CNo}. \]

Finally, with respect to (dynamic) behavior, a user would have to specify, for example, that an insertion of a new student requires a test whether this student already exists as a person (and an insertion as a person as well if this is not the case).

### 3.3 Translation into the Nested Relational Model

We briefly describe the result of a translation of the HERM schema from Figure 2 into the nested relational model next, in order to exemplify what the Translator has to do; the following schemata of nested structure are obtained (keys in the sense of [20] are underlined):

\[
\begin{align*}
\text{Person} &= \{ \text{PersonNumber}, \text{Name}\{\text{First}, \text{Last}, \{\text{Title}\}\}, \\
& \quad \text{Addr}\{\text{Zip}, \text{City}, \text{Street}\{\text{Name}, \text{No}\}\}\}\}
\end{align*}
\]

\[
\begin{align*}
\text{Department} &= \{ \text{DName}, \text{Director}, \text{Phones}\{\text{Phone}\}\}
\end{align*}
\]

\[
\begin{align*}
\text{Student} &= \{ \text{Person.PersonNumber}, \text{StudNo}, \text{Major.Department.DName}, \\
& \quad \text{Minor.Department.DName}, \text{Advisor.Professor.Person.PersonNumber,} \\
& \quad \text{Advisor.Since}\}\}
\end{align*}
\]

\[
\begin{align*}
\text{Professor} &= \{ \text{Person.PersonNumber}, \text{Chair}, \text{In.Department.DName}\}
\end{align*}
\]

\[
\begin{align*}
\text{Course} &= \{ \text{CNo}, \text{CName}\}
\end{align*}
\]

\[
\begin{align*}
\text{Prereq} &= \{ (\text{Course},1).CNo, (\text{Course},2).CNo\}
\end{align*}
\]

\[
\begin{align*}
\text{Project} &= \{ \text{Num}, \text{Begin}, \text{End}, \text{PName}, \text{Works.Professor.Person.PersonNumber}\}
\end{align*}
\]

\[
\begin{align*}
\text{Lecture} &= \{ \text{Professor.Person.PersonNumber}, \text{Course.CNo}, \text{Semester,} \\
& \quad \text{Room.No}, \text{Room.Building}, \text{Time}\{\text{Day}, \text{Hour}\}\}\}
\end{align*}
\]
Figure 2: HERM diagram of the university example.
Enroll = \{ \text{Lecture.Course.CNo, Lecture.Semester, Student.StudNo,} \\
\text{Professor.Person.PersonNumber, Result} \}

Since attributes may be inherited from entity or relationship types, the dot notation appears useful to denote their respective origin. The following are examples of inclusion dependencies in this schema:

\text{Student[PersonNumber] \subseteq Person[PersonNumber],} \\
\text{Professor[PersonNumber] \subseteq Person[PersonNumber],} \\
\text{Prereq[(Course,1).CNo] \subseteq Course[CNo],} \\
\text{Lecture[Professor.PersonNumber] \subseteq Professor[PersonNumber]}

4 Future Prospects: The Toolbox

We are currently working on a prototype implementation of RAD, which is based on a HERM implementation that was realized at the University of Kuwait [18].

One of the goals of the overall project is to make RAD a configurable toolbox. To this end, experience from work on database generators like Exodus [7] or Genesis [3] shows that a toolbox consists of three types of components:

- **Fixed** components which appear in the same way in every “version” of the system or every design system generated from the toolbox;

- **variable** ones which a user has to supply himself or herself, by writing corresponding code, translating it and binding it together with the rest of the system;

- **generated** ones which result from “compilers” inside the toolbox according to user specifications.

We plan to adopt a corresponding approach for RAD, adapted to the environment of a design process. Thus, it seems reasonable to consider the HERM+ component as fixed, since this represents the core of our methodology. On the other hand, the system should be capable of being customized for certain classes of applications, target database systems, and types of computers available to a designer. To this end, it will comprise a target-system specification environment, in which standardized languages like SQL (or declaration subsets hereof) are available for describing the things the target system will be able to understand. The remaining components of the RAD toolbox will be the output of corresponding generators.

References


Information Services
Conceptual Design and Development of Information Services

Thomas Feyer¹, Klaus-Dieter Schewe², Bernhard Thalheim¹

¹ Computer Science Institute, Brandenburg Technical University at Cottbus, Box 101344, D-03013 Cottbus
² Computer Science Institute, Clausthal Technical University, Erzstr. 1, D-38678 Clausthal-Zellerfeld

Abstract

Information services are going to be widely used due to the development of the internet and cable nets. On the basis of projects for the development of informations services like regional information services or shopping services we are developing a methodology and a general approach for the creation of information systems and information services which can be used through different nets. Our approach is based on codesign of data, information and presentation. Information units and information containers are main concepts behind our implementations. This paper presents some of the basic concepts for information services: information units and information containers. Information units are generalized views with enabled functions for retrieving, summarizing and restructuring of information. Information containers are used for transferring information to users under consideration of their current needs, the corresponding dialog step and the users environment with its restrictions. If the environment is too restrictive then small containers can be offered to the user. In the case of comfortable environment information containers can contain more and better presented information.

1 Background

‘Internet’ is currently one of the main buzzwords in journals and newspapers. However, the concepts for information presentation, information extraction, information acquisition and information maintenance are in a very early stage. This paper presents the main conceptions we use for the development of integrated information services: information units and information containers and their integration into database-backed infrastructures.

Information Services

Information services especially for the use on the internet are currently developed everywhere in the world. However, there is a number of problems which needs to be solved:

- Conceptual understanding of information services;
- Integration of different information systems;
- Maintenance of information quality;
- Evaluation of information quality;
- User-adequate presentation of information collections;
- Resource-adequate transmission through networks.

The proposals to the solution to each of the problems deserve a separate paper. In our projects we focused on the development of information services platforms which are database-backed. In this case, conceptual understanding of information services is based on conceptual modeling of underlying databases, modeling of functionality and user intentions. In order to model user intentions we
are specifying user dialogs. This approach enables us in integration of different information systems. Careful modeling can increase information quality. This paper is concentrating on two of the main concepts for user-adequate presentation and delivering of information: the information unit and the information container. Information containers are transmitted through the network under consideration of the necessary amount of information. They are transferring only those data which are necessary for the current dialog step. This optimization of data transmission is based on careful integration of data modeling with supplied functions and with dialogs.

Our experience on information services is based on two large cooperation projects in cooperation with a group of industrial partners:

- In the project FuEline [Fehl95, RoTh95] (1993-96) an online database service has been developed for the collection, management, trading and intelligent retrieval of data on industrial and university research facilities. The system is based on client-server architecture. It uses the retrieval-oriented database machine ODARS. The system is currently the basic system used in technology transfer institutes in Brandenburg.

- The project Cottbus net (since 1995) aims in the development of intelligent, simple to capture and simple to use information services which are available through the internet via computers and the cable nets via tv set-top boxes to around 90,000 households in the Lausitian region around Cottbus. So far, a number of information services has been developed together with different industrial partners: travel information service, shopping service, regional information service, industry information services, administration information services, and booking services. Several architectures and solutions have been tested, including multidimensional database architectures [Kimb96, Thom97] which has shown to be too weak and to have too low performance. Currently, two architectures (multi-tier architectures with fat or with thin clients) [RoTh95, Schm98, Schw98] are tested in parallel.

Both projects have shown us that the ad-hoc development of information services which is currently the main approach for most internet-based services cannot be used due to maintenance and development costs. Instead of that we developed a platform in parallel to the system which is in use for Cottbus net.

Database-Backed Information Services

Information, data and knowledge are different. Information and knowledge can be stored or encoded in database systems. The definition of information is still a hot research topic.

For our purposes, we assume that information as it is perceived by the human is data which is formatted, filtered and summarized, which is going to be selected and arranged by the receiver on the basis of his/her interests, experience, intuition, knowledge etc., which is going to be processed by the receiver and integrated into his/her knowledge, and which meets his/her needs and current interests.

Within this understanding of information we can assume that information services are systems which are based on database machines and use a certain communication infrastructure. Thus, information can be extracted from filtered and summarized data collections. Filtration can be computed similar to view generation. Summarization of selected data can be performed on the basis of computational functionality of the given machine. Finally, the information is presented under consideration of the environment and of the user needs.

There is a large number of information service applications which are out of the scope of our research. E.g., travel guidance systems are based on informations which are relatively stable. Such systems are made commercially available on the basis of CD-ROM’s. Database systems are used

\[1\] There is a large number of books on information systems. Each of them uses the concept of ‘information’ in a different understanding. See also [Thal97, Schm98, Schw98] for an extensive discussion of the notion of ‘information’.

2
whenever data has a good update rate and actuality is a substantial aim for the information system. The integration of database systems into information services can be based on middleware solutions. In general, information services can be integrated into DBMS. However, currently there is no full-integration solution available by database vendors. A large number of tools for retrieval and manipulation of database through the internet has been developed. Those tools use specific protocols and are mainly designed for specific DBMS. For this reason, each information service has to be based on several interfaces to databases. The information service itself is using specific databases. Thus, those database can be adapted to the needs of information services. In this case, the design and development of information services subsumes some of the ordinary design and development tasks known for database applications. Additional requirements are implied by the variety of used displays.

Codesign of Information Service Applications

Informations service applications can be based on information systems. Information systems design can be based on database design. Database application design is demanding the design of database structure with corresponding static integrity constraints, the design of database processes with corresponding dynamic integrity constraints and the design of user interfaces. Additionally, distribution is going to be modeled. Often, static constraints are mapped to dynamic constraints. Considering in detail these tasks we find that there are two different dimensions: static versus dynamic on the one hand and global versus local on the other hand. The global, static component is usually modeled by database schemata, the global, dynamic component is usually modeled by processes, transactions or application programs on the implementational layer, the local, static component is often modeled by information units, and, finally, the local, dynamic component is modeled by user interface. The last component is based on the information units and on the processes. Since the local, dynamic side is more complex than the UI and different application layers exist as well we call this component the dialog component. Dialogs consist of dialog steps. Their order depends on the application story or in general on the business processes which are specified for the given application. Thus, dialogs are a generalization of use cases. In general[YuMy94, Lewe98, Thal97], we can model dialogs for groups of actors in dependence of their roles as shown in Figure 1. The metaschema presented is simplistic since there are tight associations among information containers, supplied processes and manipulation requests. Since we are not intending to discuss the complete codesign process we use this picture for the purpose of this paper.

Although views summarize and filter data from the database, the local, static component for information services is more complex. Information units are computed by computational rules, condensed by abstraction and rebuilt rules and at the final end are scaled by customizing and building

![Figure 1: A Segment of the Dialog Model Used for Developing Information Services](image-url)

since there are tight associations among information containers, supplied processes and manipulation requests. Since we are not intending to discuss the complete codesign process we use this picture for the purpose of this paper.
a faceted representation. In Section 2 we will discuss this process. Information units can be the input information to dialogs. There are two approaches to information formation in dialogs: the formation at run-time according the environment and the request and the utilization of predefined information collections. The first approach is more general but seldom computationally feasible, the second approach is simpler and is based on results of conceptual design. Information containers are obtaining by application of formation and wrapping rules to collections of information units. In Section 3 the complete definition of containers is given. Containers are constructed from information units according the user needs and according their environment. The chosen approach to creation of

![Diagram: Information Services Codesign: Data and Process Flow Perspective](image)

Figure 2: Information Services Codesign: Data and Process Flow Perspective

information services is visualized in Figure 2.

Survey of the Paper

In Section 2, the generalization of the view concept is introduced. Information units are based on data which are processed by application of different information creation rules. As discussed in Section 3, information containers can be obtained by wrapping raw containers which are computed by formation of information units. The rules to be applied for formation depend on the dialog, especially on the dialog steps performed before, and on the environment. Finally, the management of information containers is discussed. The rules for manipulation requests are out of the scope of the paper and are not discussed.

2 Modeling Information Units

Information units are generalized views (similar to [?, Motr87]) defined on the database. Since we are only interested in support of information services we are intending to use the most general definition. We restrict the rule system which is used for generating of units from the database to the smallest possible system. The rule system can be extended by inclusion of different analysis systems which enable in detailed analysis of data sets. Other extensions can be included since the rule system is considered to be an open system. In order to define the rule system, we discuss first the modeling process and derive the rule system.
Modeling the Necessary Information

Information units are based on the database schema. They can be modeled on the basis of database schemata. They represent data in a standard, intuitive framework that allow high-performance access.

Information modeling can be considered to be modeling of semistructured data. The data model used for information units and for information containers is based on the concept of the tuple space.

Computational Rules for Information Units

As discussed above, the computation of information units is separated into three consecutive steps according to our needs:

1. Filtration
   
   Thus, the result of the application of a filtration rule to the database is a derived view with its own schema. In the simplest case, this schema is a subschema of the given database.

2. Summarization is the abstraction and construction of preinformation from the filtered data.

3. Scaling is a user-oriented process of custumizing and building a faceted representation of information based upon user interest profiles etc. It uses type-structured queries.

Whether it is necessary to materialize derived views, raw information units and information units depends on the application and the functionality which is going to be attached to the information containers.

For illustration let us consider the following database used to store data on events to be of interest to inhabitants and tourists. We are given the schema display in Figure 3(a) which is a simplification and a reduction of the ER schema used in Cottbus net.

![Figure 3](a) EER subdiagram

![Figure 3](b) raw information unit obtained by filtering and summarizing

Since the filtration is based on view generation mechanisms we describe now in detail only the rules for summarization and scaling. In the example above, filtration is based on selecting a subpart of the schema and filtering on sport events, on companies which are clubs, and on locations which sites are in Cottbus. The filtration rule is similar to a nested `Select * From ... Where ...`. We use for

---

2We use the extended, more compact ER model which allows the definition of relationship types having relationship types as its component, e.g. the type `has_role` is based on the entity type `person` and the binary relationship type `held_on` defined on the entity types `person` and `location`. The relational representation of this type is defined on the relational representation of the three participating entity types in dependence of the specified constraints.
formulation the generalized ER-QBE discussed in [GLL93, Thal98]. A simplified ER-QBE-table for this query is the following one.

<table>
<thead>
<tr>
<th>organizes</th>
<th>trading</th>
<th>promoted_on</th>
<th>belongs_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>company</td>
<td>date</td>
<td>kind</td>
</tr>
<tr>
<td>kind</td>
<td>name</td>
<td>kind</td>
<td>...)</td>
</tr>
<tr>
<td>sport</td>
<td>n</td>
<td>club</td>
<td>hosting</td>
</tr>
<tr>
<td>...</td>
<td>location</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>event</td>
<td>...</td>
<td>site</td>
<td>location</td>
</tr>
<tr>
<td>...</td>
<td>n</td>
<td>Cottbus</td>
<td>1</td>
</tr>
</tbody>
</table>

Abstraction and Rebuild Rules for Information Units

Derived views are used for representation of different aspects in the application. It is often claimed that different processes cannot be consistently represented in the database at the same time. This problem can be solved on the basis of [Thal94]. Derived views presented so far can handle summarization of numeric values. This approach is similar to the constructs incorporated into SQL on the basis of aggregation formulas. However this does not change the schema. On the basis of derived views other views can be defined as well.

The result of application of abstraction and rebuild rules to derived views is a raw information unit which needs to be adapted to the user needs, their requirements and their capabilities. We notice that on the basis of the specification of units a certain database functionality is enabled.

The events database in Figure 3(a) keeps data on ongoing cultural, sport etc. events. Our aim is to define an information unit which can be used to obtain information on sport events to be organized in Cottbus by clubs hosting the event with additional information for picking up tickets and advertisement. Thus, we can summarize

Scaling Rules for Information Units

Information units are obtained by supplement of rules to raw information units and by completion with functions:

- **Measure rules** can be used for translation of different scales used for the domain of values. Measure rules are useful especially for numerical values, prices etc.

- **Ordering rules** are application dependent. The order of objects to be used in the information unit depends on the application scenario. In our example, the preorder relationship is given by

  \[\text{hosting club} \preceq \text{sport event} \succeq \text{selling period} \succeq \text{promotion period} \succeq \text{location}\]

  Ordering rules are useful for deriving the correct order in the presentation during dialogs.

- **Adhesion rules** specify the coherence of objects which are put together into one unit. Adhesion rules are used for derivation of disapproved decompositions. Objects with a high adhesion should be displayed together. In our example, the adhesion of clubs to events is higher than of locations and time to event although two functional dependencies are valid and cannot preferred to each other.

- **Hierarchy metarules** express hierarchies among data. In the example displayed in Figure 3, several hierarchies exist like the time hierarchy (year, month, week, day, daytime), the location hierarchy (region, town, village, quarter, street). Hierarchies can be linear or multiple like the last one. The hierarchy metarules can be used for computation of more compact presentation, of summaries on data.

The following functions are attached to information units. The list is not complete and should be considered to be preliminary.

- **Generalization functions** are used for generation of aggregated data. They are useful especially in the case of insufficient space or for the display of complementary, generalized information after terminating a task. Hierarchy rules are used for the specification of applicability of generalization functions. The roll-up function[AgGS97] is a special generalization function.
- **Specialization functions** are used for querying the database in order to obtain more details for aggregated data. The user can obtain more specific information after he has seen the aggregated data. Hierarchy rules are used for the specification of applicability of specialization functions. The drill-down function used in the data warehouse approach is a typical example.

- **Reordering functions** are used for the rearrangement of units. The pivoting, dimension destroy, pull and push functions [AgGS97], the rotate function are special reordering functions.

- **Browsing functions** are useful in the case that information containers are too small for the presentation of the complete information. Sequentialization functions

- **Linking functions** are useful whenever the user is required to imagine the context or link structure of units.

- **Survey functions** can be used for the graphical visualization of the contents of the unit.

- **Searching functions** can be attached to unit in order to enable the user for computation of add-hoc aggregates.

- **Join functions** are used for the construction of more complex units from units on the basis of the given metaschema. The units are merged into a new composite unit.

Since the set of information units should satisfy also other constraints we specify also integrity constraints on the set of information units. A special kind of such constraints are version constraints which are used to express subsumption, overlapping and time order of units. If, for instance, information containers are too small then subsumed units can be used instead of the original used units.

Finally, we derive an interchange format for the developed information units which is used for the packing of units into containers. Identifiers are used for internal representation of units. The **formal context interchange format** represents the order-theoretic formal contexts of units. The context interchange format is specified for each unit by the unit identifier, the type of context, the subsequent units, and the incident units. In our example, the order is either specified by the scenario of the workflow or by the order of information presentation. For instance, it is assumed that information on possible sport events is shown before information on the history of previous sport events is given.

An advantage of our approach is the consideration of rule applicability to raw units. For this reason, simple, almost similar-looking units are generated. For each of the developed information services we obtain somewhere between 10 and 25 similar looking relationship type based information units which have the same querying and computing environment.

**Differences between external ER-views and information units**

The goals of information units are deducted from the following observations:

- Information needs to be represented in a standard, intuitive framework that enables in high-performance access.

- Large ER schemata defeat intuitive retrieval of data.

- End-users and casual users cannot understand or even remember complex ER schemata.

- End-users cannot navigate through ER schemata. Thus, query specification is getting too a hard task.

- Summarization, compactification should be supported by appropriate software as well as methods for analysis of information.

Thus, for the development of information services we need

- a standard and predictable framework that allows for high-performance access, navigation, understanding, creation of reports, and queries,

- a high-performance ‘browsing’ facility across the components within an information unit and
an intuitive understanding of constraints, especially for cardinality constraints as a guidance of user behavior.

This list of requirements looks similar to requirements which are discussed for database warehouses or for multidimensional databases [Kimb96, Pend97]. However, the requirements for information services are harder to meet. The ER schema breaks down into multiple external views. The ER schemata design technique seeks to remove redundancy in data and in the schema. Multidimensional databases often can handle redundancy but pay the price of being inefficient and of being infeasible hard to modify. Incremental modification is however a possible approach to information units and thus for information containers.

By developing simple and intuitive to use information units user behavior becomes more predictable. We can attach an almost similar functionality to the information units. This advantage preserves the genericity property of relational databases where the operations like `insert` are defined directly with the specification of the relations.

Since information containers are to be composed from units containers also maintain their main properties. Since user behavior is incorporated and additional functionality is added containers have additional properties which are going to be discussed in Section 3 and compared with other concepts in anticipation in the following table.

<table>
<thead>
<tr>
<th></th>
<th>ER schemata with external views</th>
<th>Multidimensional databases</th>
<th>ER-based information units</th>
<th>ER-based information containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>redundancy</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>schema modification</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>navigation through subschemata</td>
<td>(+)</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>relationship-based subschemata</td>
<td>(+)</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>coexistence of several subschemata</td>
<td>(±)</td>
<td>(+)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>additional functionality</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>compositionality</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>genericity</td>
<td>(±)</td>
<td>(±)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>TA-efficiency</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

### 3 Information Containers

It is often assumed in internet applications that pages can be arbitrary large. Psychological studies (e.g. [Hase95]) have shown however that typical users are scanning the part of the page which is directly displayed in their current display unit and are not interested in utilization of vertical browsing facilities enabled through mouse or pointers. This limited use of given functionality is even worse for cable net user since the devises are hard to use for browsing. For this reason, we have to take into our consideration the limitations of display. In order to be flexible, we developed the concept of information containers\(^3\). Containers can be large like in the case of mouse-base browsers of pages or can be very small like in the case of displays.

Similar to approaches used in information retrieval, we distinguish between **logical structure** (*container parameters*), **semantical content** (*instantiation of containers*) and **layout view** (*container presentation*).

\(^3\)This concept generalizes also the concept of d-objects [ScSc96].
Defining Information Containers

The data schema for information containers has to be very general. The data used in information containers can be characterized as semistructured data. Similar to the tuple space definition of [DRFP], the tuple space of containers is defined to be a collection of tuples (or more formally a multiset) where a tuple is a sequence of actual fields, i.e. expressions or values, and formal fields, i.e. variables. Variables are multi-typed. The loading procedure for a container includes assignment of types to variables. The assignment itself considers the display type (especially the size) of the variable.

Pattern-matching is used to select tuples in a tuple space. Two tuples match if they have the same values in those fields which are common in both. Variables match any value of the same display type, and two values match only if they are identical. The operations to be discussed below for information containers are based on this general framework.

Information containers are defined by three parameters:

- **Capacity** of containers restricts the size and the display types of variables in the tuple space of the container. Variables can be used for the presentation of information provided by information units.

- **Loadability** of containers parametrizes the computational functionality for putting information into one container. Functions like preview, precomputation, prefetching, or caching are useful especially in the case when capacity of containers is low.

- **Unloadability** of containers specify readability, scannability and surveyability attached to containers.

Instantiation of information containers is guided by the rules and the supported functions of the information units from which the container is loaded. Whether supported functions are enabled or not depends on the application and on the rules of the units. The operations defined on tuple spaces are used for instantiation of containers by values provided by the information units.

The size parameters limit the information which can be loaded into the containers. In Figure 4 we display three different variants of containers. The "middle" container allows us to ‘see’ the general information on a selected meeting and the information on organizers, selling agents.

Dialogs for information services we are currently developing are still quite simple. Dialog steps can be modeled by graphs, in some cases even by trees. The typing system can be changed if dialogs are more complex or the modeling of complex workflows is intended. The dialog itself can be therefore in a certain state which corresponds to a node in the graph or to a subgraph of the dialog graph. Information containers are used in dialogs. Thus, they are used for delivering the information for dialog states.

The layout view of containers depends on style rules enabled for the container in dependence of the container parameters. Additional style rules can be used for deriving container layout according to style guides developed for different applications.

Modeling the Information Interface for Dialogs

Information containers are developed according to the dialogs and the state in the dialog. For this reason, to each container is attached an additional escort information (see also Figure 4) which is used for guiding the user on the current state, the next possible states and additional background information. Often, this information is displayed in internet pages through frames. Frames are very limited in their expressibility and are often misleading. For this reason, we are currently preferring an explicit display of escort information. This escort information depends on the instantiation of the container. We can use two modes for the display of escort information: complete information displays the graph with all nodes from which the current state can be reached; escort information should at least display the path which is used through the application graph for reaching the current node.
sports
organizations

sports
commercial
provider

meetings

organizer
agent,
information
selling
additional
information

Cottbus information

interest in sport

kinds of sport

time schedule

Figure 4: The subgraph for interest in sport
Formation Rules

Formally, instantiation of information containers is the process of assigning values from information units to variables of the tuple space.

- Operations
- Archiving
- Management

Wrapping Rules

Style rules can be used for wrapping the instantiated information container.

Information containers are based on user profiles, their preferences, their expectations, and their environment. Thus, handling of containers, loading and reloading should be as flexible as possible. Customizing containers to the dialog environment can be done on the basis of customization rules.

4 Concluding

The approach presented so far has shown to be sufficient for our information services projects. In order to meet further requirements we have developed an open architecture. Thus, additional derivation rules can be added to each of the presented steps. Further, the addition of new components to derived views, to raw units, to units and to a certain extend also to containers can be handled in a simple fashion. Thus, the approach is easily extensible in the case of unexpected changes. We are not interested in developing a framework for general handling of information. Our aim has been so far the development of a platform which enables information services like business information services, administration information services, online inhabitants services, educational information services and shopping services.

The complete approach is based on rule systems. The execution semantics of rules is based on lazy evaluation. Conflicts are going to be monitored. In a latter stage, this can be done systematically.

The approach outlined in this paper is based on the derivation of information units which are used in a larger number of dialog steps. Information units are composed to information containers which are the basis for the human interface of our services. Information containers allow a certain functionality. Based on this functionality a user can send manipulation requests to the database.

Since this paper discusses approaches currently used in our information services projects there is a large number of open questions. The approach presented above has been flexible enough for the inclusion of solutions to new requirements. Thus, the presented method can be considered to be one possible approach to the development of database-backed information services.

Acknowledgement

I am thankful to the members of the FuEline and Cottbus net project teams for their stimulating discussions and their effort to implement our ideas.

References


