SEMANTIC DATA MODELING IN RELATIONAL ENVIRONMENTS
SEMANTIC DATA MODELING
IN
RELATIONAL ENVIRONMENTS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus,
Prof. drs. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op dinsdag 4 juni 1991 te 14.00 uur

door

Johannes Hendrikus ter Bekke

wiskundig ingenieur

geboren te Hengelo (O)
Dit proefschrift is goedgekeurd door de promotor Prof. dr. H.G. Sol

en de leden van de promotiecommissie:

Prof. dr. ir. P.M. Dewilde
Prof. dr. H. Koppelaar
Prof. dr. D.J. McConalogue
Prof. dr. ir. K. van der Werff
Prof. ir. D.H. Wolbers
For Ineke,
Margo, Tom and Susan
Published and distributed by J.H. ter Bekke, P.O. Box 356, 2600 AJ Delft, the Netherlands

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bekke, Johannes Hendrikus ter

Semantic data modeling in relational environments /
Johannes Hendrikus ter Bekke. - [S.l. : s.n.]. - III.
Proefschrift Delft. - Met index, lit. opg. - Met
samenvatting in het Nederlands.
ISBN 90-9004132-X
Trefw.: databanken.

© 1991 by J.H. ter Bekke

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system,
or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording
or otherwise, without the prior permission of the author.
## CONTENTS

### PREFACE

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ix</td>
</tr>
</tbody>
</table>

### 1 FROM STORAGE TO SEMANTICS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Storage</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Programming</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Algebra</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Semantics</td>
<td>4</td>
</tr>
<tr>
<td>1.6 Data models</td>
<td>6</td>
</tr>
</tbody>
</table>

### 2 MODERN APPROACHES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 The entity-relationship approach</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Case study</td>
<td>10</td>
</tr>
<tr>
<td>2.4 The semantic hierarchy model</td>
<td>12</td>
</tr>
<tr>
<td>2.5 The extended relational model</td>
<td>18</td>
</tr>
<tr>
<td>2.6 Semantic networks</td>
<td>23</td>
</tr>
<tr>
<td>2.7 Evaluation</td>
<td>25</td>
</tr>
</tbody>
</table>

### 3 BASIC NOTIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Abstraction</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Sets</td>
<td>32</td>
</tr>
<tr>
<td>3.4 Types</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>37</td>
</tr>
<tr>
<td>3.6 Bibliography</td>
<td>37</td>
</tr>
</tbody>
</table>

### 4 SEMANTIC CONCEPTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Semantics</td>
<td>42</td>
</tr>
<tr>
<td>4.3 Assertions</td>
<td>43</td>
</tr>
<tr>
<td>4.4 Convertibility</td>
<td>44</td>
</tr>
<tr>
<td>4.5 Relatability</td>
<td>48</td>
</tr>
<tr>
<td>4.6 Object relativity</td>
<td>50</td>
</tr>
<tr>
<td>4.7 Semantics in data models</td>
<td>55</td>
</tr>
<tr>
<td>4.8 Case study</td>
<td>56</td>
</tr>
<tr>
<td>4.9 Bibliography</td>
<td>63</td>
</tr>
</tbody>
</table>
5 SEMANTIC OPERATIONS
5.1 Introduction 65
5.2 Selections 66
5.3 Extensions 70
5.4 Modifications 77
5.5 Evaluation 79
5.6 Query conversion 82
5.7 Bibliography 86

6 SEMANTIC INTEGRITY
6.1 Introduction 89
6.2 Constraints 89
6.3 Inherent constraints 91
6.4 Static constraints 93
6.5 Dynamic constraints 95
6.6 Transactions 98
6.7 Design aspects 100
6.8 Conclusion 101
6.9 Bibliography 103

7 EPILOGUE
7.1 Introduction 105
7.2 The closure property 105
7.3 Practical experience 106
7.4 Supporting software 116
7.5 Conclusion 119

REFERENCES 123

INDEX 133

SUMMARY 137

SAMENVATTING 139

CURRICULUM VITAE 141
PREFACE

Data modeling, as the name suggests, is performed by making use of data models. This dissertation introduces a semantic data model and demonstrates its viability and power in a number of examples. Relational data models are frequently used in practice because of the wide availability of relational database management systems. The examples illustrate also the applicability of semantic concepts in relational environments.

Many persons have contributed to the work described in this dissertation. I would like to thank the many students at Delft University of Technology who graduated on subjects related to the database project. I found the discussions with the permanent members of the Xplain project group, Bert Bakker, Rien Bos, Dolf van der Ende and Martin van der Valk extremely stimulating. I would also like to thank Henk Wolbers for his support for the project during all these years. Henk Sol’s constructive criticism has contributed towards the improvement of the material on which my work has been based, and his continuing stimulus led to the production of this dissertation. I highly appreciated Bart de Graaf’s help in achieving this. Finally, I express my heart-felt gratitude to Ineke for her year-long patience and encouragement. It is gratifying to be in a position to thank all these persons from here.

Voorschoten, April 1991

Johan ter Bekke
1

FROM STORAGE TO SEMANTICS

1.1 Introduction

Procedural abstractions were recognized as an important research area early in the history of computer science. Major contributions originated from E.W. Dijkstra, C.A.R. Hoare and N. Wirth in the sixties. Before the seventies, data were viewed as subordinate to information processes, and no comparable contributions to data abstraction dating from this period can be cited. There was no stimulus to such research because the hardware necessary for mass data storage was not available.

From the seventies onwards, however, hardware developments started to pave the way for new opportunities, and the existing requirement for the integration of applications was fulfilled; data therefore gained in importance. Data modeling had been established as an important discipline in computer science.

Many data modeling developments have been announced in the past decades, but the implementation roots in files and records were initially still visible. Although the importance of data structures was generally recognized, programming expertise on the part of the users remained essential [Bachman 73, Coddasyl 78]. The problem of navigation, especially in complex applications as exposed by Bachman, encouraged researchers to identify a firm base: mathematics. Solutions were, however, attempted by applying existing mathematical conventions, as demonstrated by the relational data model [Codd 70]. On the other hand, the suitability of these mathematical concepts for data modeling has been questioned more recently, which has in turn initiated significant developments. It has become clear (e.g. [Kent 78], [Sowa 84]) that the concepts should not only have a mathematical (i.e. formal) base, but should also improve the formalization of the meaning (i.e. semantics) of the data.

This chapter briefly reviews the developments in data modeling during the last two decades; these are subsequently taken as a starting point for the discussion of the most recent developments in the discipline. An analysis of their shortcomings leads to the introduction of a new semantic data modeling approach which is the subject of this dissertation.
1.2 Storage

The development of data structures in the fifties and sixties was mainly determined by hardware limitations. Separate files were defined for each application characterized as: individual file organization per application. There was essentially only one level of abstraction.

Figure 1.1

Files are directly accessed by application programs. As a consequence, the file organizations determine the structures of application programs.

The following organizations are distinguished:

- **sequential**
  Data are processed in sequential order, which can therefore only be applied in bulk processing.

- **index sequential**
  Files are accessed both sequentially and randomly. Regular reorganization is necessary.

- **B-tree**
  Files are accessed both sequentially and randomly in a hierarchically sequential file structure. Reorganization is part of the algorithms associated with this organization type.

The direct application of these organizations does not encourage integration of files and programs. They can only be modified with great difficulty because of this dependence. The desirability of a new approach is therefore obvious.

1.3 Programming

Programming was important in the early days of data oriented approaches. Data were structured in hierarchies or networks, with the following associated concepts:
• **record**
  a logical group of data, structured as a single unit;

• **relation**
  a link between two records.

The application of the above concepts required programming expertise [Bachman 73], and the programmer had to be capable of following the predefined navigation paths.

A significant milestone was reached in this period: while programs hitherto contained both specifications for data and manipulations, these were now separated and two new languages were introduced:

• **the data definition language (DDL)**
  for defining data, including their relationships;

• **the data manipulation language (DML)**
  for interfacing applications with software systems.

However, the objective of separating data logically and physically was not yet achieved. Attempts were therefore made to provide a more effective starting point for data structuring.

### 1.4 Algebra

During the early seventies it became clear that the conceptual dependence on physical data structure was standing in the way of theoretical understanding. The required firm base for the discipline was borrowed from mathematics: the relational theory.

Data were no longer required to be linked perceptibly in the relational model [Codd 70]. Two abstraction levels were distinguished (see also figure 1.2):
the logical level
at which data were defined mathematically;

the internal level
at which data storage was described.

The required autonomy was emphatically expressed in the phrase data independence. The user was enabled to formulate manipulation commands in a mathematically oriented language and was no longer required to have programming expertise, or to be familiar with the implementation. An example of such a language is relational algebra with following operations:

- **union**: $R \cup S$
a set consisting of elements occurring in at least one of the operands $R$ or $S$;

- **difference**: $R - S$
a set consisting of elements occurring in the first operand $R$ but not in the second operand $S$;

- **cartesian product**: $R \times S$
a set produced by concatenating each element of $R$ with each element of $S$;

- **projection**: $P (R : A)$
the projection $P$ of $R$ on $A$ consists of elements of $R$, omitting attributes not belonging to $A$;

- **selection**: $S (R \mid F)$
a set of elements of $R$ satisfying condition $F$.

Although mathematics generally allows various interpretations of one and the same object (e.g. sets sometimes considered as elements), data models based on the relational theory lack this flexibility. A relation cannot be used as an attribute, so relations and attributes, as the relevant interpretations in these data models, cannot be exchanged. These models are therefore inadequate for practical applications: they are too rigid and flat.

### 1.5 Semantics

The three schema database architecture was defined in a standard [ANSI 75] in 1975. Optimal flexibility was introduced by defining following abstraction levels, see figure 1.3:
Figure 1.3

- *the internal level*
  defining data implementations;

- *the conceptual level*
  defining data relationships (i.e. semantics);

- *the external level*
  defining data interpretations.

The essence of this architecture is that one conceptual data description can result in more than one implementation and interpretation; so two types of flexibility are introduced:

- *flexible implementation*
  Data descriptions in terms of implementations are undesirable, particularly so in the relational approach. The associated phrase is data independence, i.e. that modifications of data structure implementations are invisible on the conceptual level. Data independence is consequently paramount in software maintenance.

- *flexible interpretation*
  Data descriptions leading to only one interpretation are undesirable. The required flexibility cannot be achieved by the relational model and can result in new data models. The phrase here is view independence, which accepts new interpretations in addition to the existing ones. Existing interpretations do not require adjustment. View independence is therefore especially useful during data modeling.
Modern data modeling approaches make use of relational model achievements; data independence is the essential ingredient. These methods therefore differ in the way view independence is treated. Approaches satisfying this objective are considered as semantic.

1.6 Data models

The early approaches to data modeling were developed from experience in data storage facilities in available computer systems. Their development was thus based on programming practice; the methods in which these approaches were formalized, were only developed after a number of implementations had been carried out.

One of the achievements of the relational model has been the introduction of the notion of data model. Codd especially emphasized the importance of a data model in his Turing Award Lecture [Codd 81] and stated that the relational model was the first data model consisting of a combination of the following three components:

- a collection of data structure types (the building blocks);
- a collection of operators or rules of inference, which can be applied to valid instances of the data structure types;
- a collection of general integrity rules, which implicitly or explicitly define the set of consistent states or changes of state or both.

A database is a collection of related data. A database is called consistent according to a certain data model if it satisfies the data structure types and general integrity rules of that data model. A database management system (DBMS) is a software system for the use and control of databases.

The following chapter presents an overview of some modern data modeling approaches, with emphasis on data structure types. Not all approaches are data models in the sense above, because of missing operators. That is probably why most approaches require a translation into a relational equivalent for implementation. They may be useful for information analysis, especially in the early stages of determining a preliminary informal description for data modeling.
2

MODERN APPROACHES

2.1 Introduction

E.F. Codd’s research prompted the publication of various proposals for data modeling during the late seventies. These developments mainly concerned logical and semantic aspects of data modeling, without questioning the basic principles of the relational model.

The relational model enhancements concerned the improved transparency of data models. This was achieved by the addition of diagrams in the entity-relationship approach (section 2.2). Section 2.3 includes a case study on the consequences of adopting entity and relationship as basic concepts. Mutual positioning of relationships improved the transparency in semantic hierarchy models (section 2.4). Section 2.5 discusses Codd’s comments on these developments. The influence of developments in the area of object representation manifests itself in the data modeling approaches. From this point of view we will describe semantic networks in section 2.6.

2.2 The entity-relationship approach

The entity-relationship approach is sometimes considered as one of the first useful proposals on the subject of semantic data modeling. The popularity of this approach is mainly attributable to Chen [Chen 76]. Whether we can consider this approach as a data model in its own right is as yet unclear, as it mainly consists of various concepts embodied in certain diagrams. Aspects like manipulation and integrity are barely covered.

Chen [Chen 86] considers the excessive dependence on database specialists in the design process to be the most serious problem we face in data modeling today. Communication between users and specialists is improved by the introduction of special diagrams, called 'enterprise' schemas for certain application areas. These can be developed irrespective of the nature of the database management system in use, followed by the transformation of the schemas into classical database definitions (e.g. relational or network).
E-R diagrams

The entity and relationship concepts are of utmost importance in the diagrams - that is why they are called E-R diagrams. By entity we mean something with objective reality (e.g. a supplier or a product), while entities are connected by relationships (in e.g. 1:1, n:1, n:m relationships). Distinctive symbols have been defined for these concepts: an entity is indicated by a rectangle, a relationship by a diamond. Furthermore, attributes are shown as ovals.

Suppose we wish to record information about employees and their allocated projects; the E-R diagram in figure 2.1 shows the relevant entities and relationships.

![E-R Diagram](image)

**Figure 2.1**

These E-R diagrams are based on entities (employee and project) and relationships (working for). Both entities and relationships have attributes (emp-#, name, office-ext, home-ext, percentage, proj-# and budget), while attributes are related to domains (emp-#, name, extension, %-manpower, proj-# and amount).

In order to meet the requirements of later developments, additional constructs have been introduced in the above E-R diagram [Chen 86]. The need for the ability to represent generalizations (and specializations) arose in the areas of artificial intelligence and data modeling in the late seventies. It was satisfied by the creation of the 'is a' relationships, in which the connections are more closely defined by an indication of direction (the precedence relationship), as shown in figure 2.2.
Finally, constructs were created for set oriented relationships, such as union (disjunct entity types can be combined into a new entity type) and intersection (entity types overlap, the overlap being considered as an entity type). An example is shown in figure 2.3.
People applying the E-R approach for the first time are often overly enthusiastic: data modeling is supported by the use of diagrams as an effective means of communication with the users. The limitations, however, emerge when more complex structures need to be defined: problems crop up because of the entity and relationship concepts; a semantic problem is introduced because it is difficult to interpret a relationship as an entity as well.

With view independence, the outcome of data modeling projects is independent of the chosen development trajectory. We shall demonstrate by a simple example in the following case study that E-R solutions are not unique; they are dependent of the chosen development trajectory.

2.3 Case study

Description

In a club of CD (compact disc) player owners, members may borrow CD’s from each other. CD’s are characterized by title, performer and owner. Identical CD’s may be owned by more than one member. Recorded member data are name and address. Property, borrowing and reservation are to be modeled.

Solution

The above description contains the items relevant to the real life situation to be modeled. Items with relevant properties are CD, member, property, borrowing and reservation, further defined as follows.

CD

CD is interpreted in a number of ways in the description:

- the CD type:
  meaning the combination (title, performer) found on different instances, for which we shall use the phrase CD. This is useful for modeling reservations.

- the CD instance:
  meaning instances owned by the members, which therefore corresponds to property, which is also mentioned. We shall use the phrase property.

CD may be interpreted as an entity with the attributes title and performer. Property will be defined separately.
Member

Members own, reserve and borrow, and it is therefore obvious to model member as an entity with the attributes name and address.

Reservation

CD's are linked to members as requesters for reservations. We therefore model reservation as a relationship, enabling us to record the attribute 'required period'.

Property

Similarly to reservation, property indicates a relationship between CD's and members (as owners). Additional attributes need not be defined.

Borrowing

Borrowing links member (as borrower) to property. We must now consider the way these two were modeled: member was in an earlier stage defined as an entity and property as a relationship. Borrowing can therefore not simply be regarded as a relationship.

A number of solutions can be applied to this problem:

- introduce a new entity, related 1:1 to the relationship property;
- change the interpretation of property into an entity; additional relationships with member and CD have to be introduced to obtain a valid model;
- regard borrowing as an entity; the interpretation of property remains unchanged, but, likewise, a relationship with member must be added.

As the above example demonstrates, the E-R approach solution is ambiguous because of the interpretations of entity and relationship.

Conclusion

Problems were encountered in this case when borrowing was to be modeled, requiring entities and relationships which had been modeled at an earlier stage. The inflexibility of the structuring concepts presented a problem.

Entities and relationships are unsuitable to solve this problem unambiguously. A possibly better approach would be to use only one fundamental structuring concept and consider
entities and relationships as its exchangeable interpretations, as will be demonstrated in the following chapters.

2.4 The semantic hierarchy model

A disadvantage of the relational model is its inadequate treatment of semantics, which has been pointed out especially by J.M. Smith and D.C.P. Smith [Smith 77a, 77b, 78a, 83]. The phrase database abstractions was also coined because abstractions play such a dominant role in their approach. Many other researchers have studied database abstractions: Codd was strongly influenced by these developments as well, as can be seen from his publication on the extended relational model [Codd 79], of which we shall explain the most important properties later on in this chapter.

Abstraction can be defined as 'the omission of all details, other than the characteristics relevant to sound understanding', in other words, 'the omission of all details, excluding the characteristics relevant to the information needs'.

Smith and Smith specify several abstractions, two of which are:

- **aggregation** which abstracts a relationship between several objects to a higher level aggregate object;

- **generalization** which abstracts a number of common characteristics of a collection of objects to a higher level generalized object.

These abstractions will be illustrated by the following examples.

*Aggregation*

Aggregation is often used in natural language to simplify complex relationships. Smith and Smith use the following example based on an American university system, where enrollment is carried out for each course:

'Student ST achieves grade G in a class of a course numbered C#, consisting of CH credit hours, and a description D, by lecturer L during the semester S, in lecture room LR'.

The following relation looks acceptable:

\[
\text{relation } R (\text{ST, G, C#, CH, D, L, S, LR}).
\]

We cannot, however, define the meaning of this relation as a noun. A more detailed
analysis shows that aggregation was used twice in above sentence. The noun course (CO) is an aggregate of the relationship between C#, CH and D. The noun class (CL) is an aggregate of CO, L, S and LR.

The eightfold relation can therefore be reduced to a threefold relation containing class CL, student ST and grade G. This relation can be abstracted to the aggregate object of enrollment.

Thus we see following aggregate objects [Smith 77a]:

\[
\begin{align*}
\text{type enrollment} &= \text{record} \\
  &\quad \text{ST} : \text{student} \\
  &\quad \text{CL} : \text{class} \\
  &\quad \text{G} : \text{grade} \\
  &\quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{type class} &= \text{record} \\
  &\quad \text{CO} : \text{course} \\
  &\quad \text{S} : \text{semester} \\
  &\quad \text{L} : \text{lecturer} \\
  &\quad \text{LR} : \text{lecture room} \\
  &\quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{type course} &= \text{record} \\
  &\quad \text{C#} : \text{number} \\
  &\quad \text{CH} : \text{credit hours} \\
  &\quad \text{D} : \text{description} \\
  &\quad \text{end}
\end{align*}
\]

The resulting objects can be represented in the structure shown in figure 2.4.

It will be obvious that aggregation results in a hierarchy of defined objects, which leads to a better understanding of the data models. For an insight into the enrollment object, for instance, the exact nature of the subordinate object class is irrelevant.

Having defined the objects, we can proceed to allocating primary keys. For example, course is uniquely identified by course number C#.

Smith and Smith show that normalization is independent of many semantic aspects of aggregation, by the following example:

'Student ST has been enrolled in class C and climbed mountain M'.
This is summarized by the relation

\[ \text{relation } R \left( \text{ST, C, M} \right) \]

which is in third normal form because the attributes collectively result in the primary key; we cannot possibly define the meaning of this relation, however. By using database abstractions, we could distinguish the aggregates of \textit{enrollment} and \textit{climbing}, the latter being defined as

\[ \text{type climbing = record} \]
\[ \text{ST : student} \]
\[ \text{M : mountain} \]
\[ \text{end} \]

\textit{Generalization}

We have described a structuring mechanism (aggregation) in the above, of which an 'equivalent' could be captured in the relational model. There is no relational 'equivalent' for generalization, because it is a new structuring component in databases, though earlier known in fundamental programming as 'discriminated union' [Hoare 72].
Smith and Smith define generalization as an abstraction form enabling us to regard classes of individual objects as a single, defined object class.

We clarify generalization by an example. Suppose we were asked to design a model for an organization, in which employees appear as objects. Suppose, in addition, that we have three staff categories: truckers, secretaries and engineers. Employee data are to be recorded on the basis of category:

employee: identification, name, age, profession, ...
trucker: vehicle, ...
secretary: typing speed, ...
engineer: specialization, ...

Applying the above may lead to two relational models:

(1) relation employee (identification, name, age, profession, vehicle, typing speed, specialization)

where we do not distinguish between employee categories, or:

(2) relation trucker (identification, name, age, vehicle)
relation secretary (identification, name, age, typing speed)
relation engineer (identification, name, age, specialization).

In the first case all attributes are not applicable to all instances. This definition does not represent the real world adequately because users keep in mind what attribute is relevant to what employee. This is a source of errors because:

• the description of real world objects is incomplete if relevant values are not specified;

• the description of real world objects is incorrect if irrelevant values are specified.

In the second case the above disadvantages have been eliminated, but a new problem is introduced:
Suppose a user group is only interested in global employee characteristics. The retrieval of these would have been possible by access to just one relation in the first setup, but in the second one the relation employee is no longer available. The retrieval of employees characteristics now requires access to three relations. The same problem occurs with specification of relationships and integrity rules concerning the missing employee relation.

So both solutions have distinct disadvantages.
These disadvantages do not appear when database abstractions are used. The application of aggregation and generalization results in the following model.

The most generic object to be represented is employee, and consequently the model first of all contains the object employee (see figure 2.5).

![Diagram of employee object]

**Figure 2.5**

The next step is to decompose the object in the generalization/specialization plane, when we find the professions of trucker, secretary and engineer. The resulting schema is shown in figure 2.6.

![Diagram of employee, engineer, secretary, and trucker objects]

**Figure 2.6**

Combining generalization and aggregation, we end up with the schema shown in figure 2.7.

The introduction of both aggregation and generalization offers considerable advantages, of which we will describe a few, pointing to the examples in figure 2.8.
Figure 2.7

Figure 2.8
We can define exactly each relationship in this schema, for example:

haulage, as a relationship between trucker and truck; 
visit, as a relationship between engineer and car;

but also:

assignment, as a relationship between secretary and engineer; 
participation, as a relationship between project and employee.

Not all of the above relationships can be represented in the classical relational model [Codd 70] because of the missing abstraction of generalization. This indicates a serious inadequacy of this model.

2.5 The extended relational model

The first article on the relational model by Codd appeared in 1970 [Codd 70], followed by textbooks about this model [Date 86a, Ullman 80]. The relational model slowly evolved into a full-fledged data model: all aspects (structure, manipulation and integrity) have been defined [Brodie 82a], and commercial database management systems have been created following these specifications.

A significant development in data modeling took place in 1977: publications by Smith and Smith [Smith 77a, 77b, 78a] focused attention on the nature of abstraction that people use to gain an insight into reality. These abstractions (aggregation and generalization) had already been firmly established in philosophy and in artificial intelligence.

Many researchers had to adapt their theories to the new developments: for the relational model this resulted in the creation of the Extended Relational Model [Codd 79], called RM/T (the T standing for Tasmania, where Codd stayed during a sabbatical year).

The RM/T found remedies for a number of the problems associated with the classical relational model, particularly:

• an improved structural integration of referential integrity by the introduction of certain simple keys;

• extended structuring capability by the introduction of subtypes (cf. specialization) and supertypes (cf. generalization).

We will now consider the innovations applied to the classical relational model, which,
as far as we can trace, have not yet been implemented on any commercial product.

Entities are represented by E and P relations, familiar from the classical model as n-ary relations. Both have specific functions:

- **E relations** are used to posit the existence of entities;
- **P relations** describe the properties of entities.

RM/T entities belong to one of the following categories:

- **kernel entities**: autonomous entities, not dependent on other entities (examples: employee, department, customer, product);
- **characteristic entities**: entities for the sole purpose of describing other entities (example: printed lines in an invoice, defining the entity invoice);
- **associative entities**: entities defining n:m relationships between two or more entities (example: supply, indicating an n:m relationship between supplier and article).

Properties can be found for all types of entities. For example, employees have a name and salary, invoices have a quantity and an amount, and supplies have a date.

**Surrogates**

In essence, entities need to be defined by unique identifications. Primary keys, consisting of one or more attributes, are used for this purpose in the classical relational model. We shall deal extensively with the disadvantages of this approach later on.

Codd, when designing RM/T, obviously realized the drawbacks of keys based on attributes, as they only play a subordinate role - the disadvantages are removed by making use of E and P relations.

All identifications in RM/T are set up by the use of certain primary keys controlled by the database management system. These single system keys are called surrogates. All relations in RM/T are defined by surrogates: primary and foreign keys, central in the classical model, are thus replaced by single identifiers. Let us now look at the E and P relations.

**E and P relations**

Each unary relation (i.e. a single attribute relation) corresponds to an entity type in the
conceptual model. All surrogates, corresponding to the appropriate entity type, are summarized in this E relation. A number of P relations subsequently correspond to each entity type, each relation being used to represent one or more properties. The database designer determines the required number of P relations.

Let us have a look at an explanatory example. In the classical relational model, the relations for suppliers (S), products (P) and supplies (SP) could be defined as:

- relation S (sup-No, name, town)
- relation P (prod-No, price)
- relation SP (sup-No, prod-No, date, quantity).

The database in the extended model would consist of three E relations (figure 2.9).

![Figure 2.9](image_url)

The above E relations contain the unique, unalterable, single identifiers alpha, beta, ..., lambda. The corresponding P relations can now be defined - but a choice has to be made from the following extremes:

- one P relation per attribute
  resulting in supplier P relations
  SS (S¢, sup-No)
  SN (S¢, name)
  ST (S¢, town)

- one collective P relation for all attributes
  resulting in the supplier P relation
SSNT (SC, sup-No, name, town).

The contents of the collective P relation are summarized in figure 2.10.

SSNT:

<table>
<thead>
<tr>
<th>SC</th>
<th>sup-No</th>
<th>name</th>
<th>town</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>1023</td>
<td>Smith</td>
<td>Leeds</td>
</tr>
<tr>
<td>beta</td>
<td>1024</td>
<td>Johnson</td>
<td>Liverpool</td>
</tr>
</tbody>
</table>

*Figure 2.10*

Kernel entity types are easily represented in this way. The difference with the classical approach is highlighted in the associative entity types - remember the supplies relation in our example. As we have said earlier, the role of foreign keys is transferred to the surrogates, and so the P relation, corresponding to supply in the example, does not have to contain the properties sup-No and proj-No. If only one P relation is selected for supplies, this is what it looks like:

SPSPDQ (SPc, SC, Pc, date, quantity).

It could have the contents shown in figure 2.11.

SPSPDQ:

<table>
<thead>
<tr>
<th>SPc</th>
<th>SC</th>
<th>Pc</th>
<th>date</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>zeta</td>
<td>alpha</td>
<td>gamma</td>
<td>900401</td>
<td>30</td>
</tr>
<tr>
<td>eta</td>
<td>beta</td>
<td>delta</td>
<td>900403</td>
<td>50</td>
</tr>
<tr>
<td>delta</td>
<td>alpha</td>
<td>gamma</td>
<td>900407</td>
<td>10</td>
</tr>
<tr>
<td>iota</td>
<td>alpha</td>
<td>epsilon</td>
<td>900411</td>
<td>20</td>
</tr>
<tr>
<td>kappa</td>
<td>beta</td>
<td>delta</td>
<td>900412</td>
<td>40</td>
</tr>
<tr>
<td>lambda</td>
<td>beta</td>
<td>gamma</td>
<td>900412</td>
<td>70</td>
</tr>
</tbody>
</table>

*Figure 2.11*
Subtypes and supertypes

Assume a database with information on employees and engineers. Here we have a case of generalization and specialization, as every engineer is an employee, but not every employee is an engineer. In RM/T, an engineer is considered as a subtype of supertype employee, and the relationship between these two types is established by surrogates in the following way.

Suppose we have three employees (alpha, beta, gamma) of which two are engineers (alpha, gamma). We would therefore consider the E relations of figure 2.12.

![M and E relations](image)

**Figure 2.12**

The appropriate P relations, assuming one relation for each classical one, are shown in figure 2.13.

<table>
<thead>
<tr>
<th>E¢</th>
<th>salary-No</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>102387</td>
<td>Smith</td>
</tr>
<tr>
<td>beta</td>
<td>106813</td>
<td>James</td>
</tr>
<tr>
<td>gamma</td>
<td>260098</td>
<td>Johnson</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E¢</th>
<th>specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>Computer Science</td>
</tr>
<tr>
<td>gamma</td>
<td>Electrical Engineering</td>
</tr>
</tbody>
</table>

**Figure 2.13**

It therefore follows that the subtype/supertype relationship cannot be retraced in the
structure of the E and P relations. As a consequence, the control over such relationships by a database management system remains unclear.

2.6 Semantic networks

Object oriented systems enable us to classify real objects in a structured manner - made possible by 'inheritance links' across objects (which we call modes or frames). These links are often referred to as 'is-a' links, which are characteristic and stable elements of semantic networks.

We will summarize the application of is-a links in object oriented systems in the remainder of this section [Brachman 83].

The principle of is-a links is straightforward. We often come across expressions like 'John is a bachelor' and 'A dog is a mammal' in natural language. Although both sentences contain an is-a link, their meaning differs: the first one relates an individual (John) to a category (bachelor), while the second relates two categories (dog and mammal). Because one is-a link is used in different meanings, we can expect problems in coupling different meanings in a hierarchy of is-a links.

The hierarchical structure of is-a links is especially useful when representing knowledge. Common object properties are ranked high in a hierarchy, which means savings in storage capacity. Property distribution is presented in graphical notation, general categories being placed above more specific ones, and, as a consequence, lower categories inherit properties from the higher ones (see also figure 2.14).

Although is-a links relate dissimilar objects (collections, categories, predicates, prototypes, images, individuals etc.), we can distinguish between two fundamentally different links: the link between two generic objects (categories), and the link between a generic and an individual object. We shall concentrate on both in the following (see also [Brachman 83]).

Generic/generic relationships

When we relate two categories by an is-a link, we imply that one category is less general than the other one. Differences in interpretation arise from the categories we wish to consider. If both categories consist of definite numbers of individual objects, this relation is comparable with the subset/superset relations from set theory. In the case of a relationship between predicates, this relationship is comparable with generalization/specialization, Brachman’s preferred type of link in semantic networks.
Figure 2.14

Generic/individual relationships

An is-a link between an individual object and a generic object implies that an individual object may be defined by a general description; the relationship is often called 'instantiation'.

Differences in interpretation are found here as well, depending on the nature of the category considered. If the category has extensional properties (i.e. defined by individual objects), the relationship is interpreted as a set membership relationship. If the category has intensional properties (i.e. defined by predicates), we have a type-instance relationship.

One problem in knowledge representation for semantic networks is the sometimes recurring necessity to ignore inherited properties. Figure 2.14 shows an example: although birds can generally fly, this does not apply to the ostrich. It is this type of exception that renders major flaws in this approach.
2.7 Evaluation

We have seen a general overview of some modern data modeling approaches in this chapter. Although views and solutions vary widely, abstractions and generalizations (or synonyms) appear to be playing a predominant part. Let us try to confirm this point by briefly evaluating these approaches separately.

The E-R approach mainly consists of conventions geared to the diagrammatic representation of models; the need for two separate concepts (entity and relationship) interferes with the exchange of interpretations which is fundamental to the semantic approach. The extensions required to define generalizations are clearly introduced on an ad hoc basis.

Database abstractions introduce the concepts of aggregation and generalization, as viewed from the classical relational model. Abstractions (relation names) are not assigned the importance they merit; this is a disadvantage, because only (key) attributes are essential in the relational model. The aggregation and generalization abstractions (i.e. the use of nouns) is not formally realized.

The RM/T model is overloaded with concepts, which impedes the exchange of interpretations (also refer to [Earl 86]). Single identifications (surrogates) are evidently considered as important in the unambiguous definition of relationships. P relations appear to represent an aspect belonging to the description of a database implementation. The existence of subtypes and supertypes emerges from an analysis of the values in a database. These notions are not expressed in the data model, which can be considered as a shortcoming of RM/T.

Semantic networks are exclusively based on the is-a connection, which can be interpreted in more than one way. These connections in derivations are therefore ambiguous.

Conclusion

Many data modeling approaches are based on fixed interpretations of database objects: for instance relations and attributes in the relational model - they cannot be substituted by each other. The same holds for other data modeling approaches.

Semantic models, however, must allow flexible interpretation of objects: while some consider a certain object as an attribute, others may regard it as a type. The object of color, for example, is often taken as an attribute, but there are circumstances when color is preferably regarded as a type, because it has some properties in common (consider, for example, paint manufacturers).

Many researchers have emphasized the importance of this type of flexibility (amongst others [Tsichritzis 82] and [Date 86a]). The following chapters are devoted to a new
proposal that fulfills the flexibility of interpretation and results in a new kind of data model. The usability of this new data model will be demonstrated with some practical examples.
3

BASIC NOTIONS

3.1 Introduction

When we want to model the dynamic aspects of reality, we can look for a number of aspects, that retain their identity even under continuous change; what we are really trying to do is to identify invariant properties. So, invariants are fundamental to models of the real world and therefore also to data processing. Invariants consequently constitute the main body of the semantic model in this dissertation.

Invariants appear when we use the set concept. However, the set concept is sometimes incompletely defined in computer science, or explained by a short description, followed by a handful of examples and a construct of theorems and definitions, consistent in itself.

The set concept is used to define definite collections. An example of a perfectly acceptable set is the one of natural numbers, of which the elements are defined: today’s natural number set is certainly not different from tomorrow’s - the definition would be useless if it were. Application of this set concept in data processing leads to fundamental problems because of the dynamic character of the relevant collections. Should one attempt to evade this difficulty by defining 'a set of all sets', one would just have displaced the problem. The solution we propose in this chapter is a fundamentally different one.

The first step in the search for appropriate formal structures must be an accurate analysis of existing concepts. We will therefore start this chapter by analyzing the set concept. The study of specific properties of data in a data processing environment leads to the definition of the type concept - with a surprising result: the type concept is complementary to the set concept. Operations on types are closely related to abstractions considered to be relevant in language sciences and philosophy. The application of the set concept as well as the type concept enables us to describe database invariants adequately.
3.2 Abstraction

We find so many different details in the real world, that the only way to gain an overview is to leave out a selection of irrelevant details and thus retain a workable collection. The resulting descriptions of the real world are called abstractions. These abstractions constitute an extremely important area of research in data modeling.

As a consequence of model invariance, descriptions of the real world, to be included in a model (i.e. all abstractions), must retain their validity over time. The question of which aspect to consider as invariant, depends on the application area. An example should clarify the above.

In the data modeling area we might be faced with the problem of designing a conceptual model for a certain company, where we could expect a wide variety of data. For example, if we wished to describe the range of available articles, we would be tempted to make use of the set concept - however, it is not as simple as that.

An article range can be described by listing all articles available at a certain moment (i.e. its extension), but then we face the draw-back of continuously updating the model because other sets come into play. And furthermore, we are unable to predict how many sets there will be and what their nature will be.

The only firm statement we can make is that the type article plays a role in the company, where 'article' is always considered as an object with a fixed number of properties, e.g. description, weight and price. Here the elements as such are irrelevant to the modeling process; what matters are their descriptive properties (i.e. its intension). This interpretation corresponds with [Sowa 84].

At first sight the traditional set concept appears directly applicable to the description of data in a database, because it represents an abstraction defined as 'the collection into a whole of definite, distinct objects of our intuition or thought' [Fraenkel 76]. The whole is called a set and the collected objects are the elements contained in it. A set is determined by its elements; this is called the axiom of extensionality. Since set theory is based on the set concept, all elements should be sets also; these sets are called hereditary sets. A simple example to clarify the concepts follows.

Suppose we wish to collect three objects, to be identified by the letters A, B and C, into a set. The mathematical notation is

\{A, B, C\}.

In this set we use names for the objects, themselves representing sets according the axiom of extensionality, and so we require a name for the whole as well. If we name this new
set $S$, we have the following definition:

$$S = \{A, B, C\}.$$  

For our purpose we shall use the following more suitable notation

$$set \ S = A, B, C.$$  

This definition is considered as an assertion, i.e. a positive statement. An assertion consists of two parts: the subject and the predicate. In our example the left hand part represents the subject (i.e. the new set $S$) and the right hand part represents the predicate (i.e. the existing sets $A$, $B$, $C$).

Because a set is defined as a collection of definite, distinct objects, it must be possible to decide unambiguously for each object whether or not it is an element of the set. In this case we have no problem at all. For example, it is trivial to say that $B$ is, and $X$ is not, an element of the set.

Let us now analyze the collection

$$G = \{A, B, \{D\}\}$$

somewhat more closely. At first sight this collection consists of three elements, defined by

$$set \ A = \ldots$$

$$set \ B = \ldots$$

$$set \ ... = D$$

which means that two elements are known by their names and one element by its description. So, the ... in these definitions represent the undefined parts. We find that the elements are sometimes indicated by a subject but also by a predicate. This way of defining sets leads to decision problems and ambiguities, as we shall demonstrate.

If the collection $G$ consists of three different elements, it is simple to decide whether any of the elements is or is not contained in the collection. In our example, however, we can only make this decision if the element has been completely defined, i.e. by subject and predicate. For instance, we can only say that element $C$ belongs to the collection, if it has been previously defined as

$$set \ C = D.$$  

We cannot make that decision if the predicate is missing from the definition above.
On the other hand, the given definition of the set $G$ also results in ambiguities: the number of elements in $G$ can not be determined using the given definition. In case of the definition

\[
\text{set } C = D
\]

$G$ has three elements, while the definition

\[
\text{set } B = D
\]

results in a set $G$ with two elements.

If we allow subjects and predicates to indicate elements, we have to supply a complete description of every element. This would imply a never-ending description of the real world, which is clearly not possible.

In order to avoid ambiguities we allow only definitions which collectively either at least contain the subject (i.e. a set) or the predicate. So, predicate definitions of the type

\[
\{\{X\}, \{Y\}, \{Z\}\}
\]

are certainly valid, because each element is defined by a predicate. The relevant question now is: can a collection of predicates be considered as a set as well?

The notation

\[
\text{set } ... = \{X\}, \{Y\}, \{Z\}
\]

could be used for a set. According the axiom of extensionality this results in the following definitions for the three elements:

\[
\begin{align*}
\text{set } ... &= X \\
\text{set } ... &= Y \\
\text{set } ... &= Z.
\end{align*}
\]

Also these elements should be sets, so:

\[
\begin{align*}
\text{set } X &= ... \\
\text{set } Y &= ... \\
\text{set } Z &= ... .
\end{align*}
\]

This results in unacceptable situations where both subjects and predicates are regarded as sets. Complete definitions are required here, as earlier.
We could end up in a never ending specification of the real world. A remaining solution is to consider the collection of a number of predicates as a predicate in its own right. This is what we call a type, which abstraction we denote as

\[ \text{type } T = X, Y, Z. \]

This discussion leads us to two analogous disjunct concepts: set and type.

\[\text{set employee}\]

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>town</th>
<th>department</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>Brewer</td>
<td>6, Knot</td>
<td>Shingles</td>
<td>stores</td>
</tr>
<tr>
<td>502</td>
<td>Fisher</td>
<td>23, Stream</td>
<td>Selling</td>
<td>manufacturing</td>
</tr>
<tr>
<td>503</td>
<td>Taylor</td>
<td>12, Lane</td>
<td>Guiding</td>
<td>manufacturing</td>
</tr>
<tr>
<td>504</td>
<td>Stewart</td>
<td>3, Bush</td>
<td>Wimblon</td>
<td>sales</td>
</tr>
<tr>
<td>505</td>
<td>Potter</td>
<td>67, Market</td>
<td>Washing</td>
<td>accounting</td>
</tr>
<tr>
<td>506</td>
<td>Butcher</td>
<td>2, Place</td>
<td>Selling</td>
<td>manufacturing</td>
</tr>
<tr>
<td>507</td>
<td>Brewer</td>
<td>7, Crescent</td>
<td>Guiding</td>
<td>purchase</td>
</tr>
</tbody>
</table>

\[\text{Figure 3.1}\]

We can, by applying operations, create new sets from existing sets. In a similar way we could create new types from existing types. In the case of sets the usual operations are union, intersection and difference. For types we introduce similar operations viz. specialization, generalization and differentiation.

It should by now be clear why sets are being defined by objects (identified by subjects) and types by properties (identified by predicates). We could, for example, define employee as a type with the properties name, address, town, department and salary. We call the set of employees staff, which is defined by the totality of individuals in an organization (i.e. employee identifications). This is shown in figure 3.1.

Having introduced sets and types, we shall elaborate on these in the following sections.
3.3 Sets

The set concept is given ample attention in this section, in order to stress its difference from the type concept. The definition of set will therefore be followed by some practical examples.

The set concept is defined as follows:

A set is a definite collection into a whole of distinct objects. An object from the collection is called an element of the set.

We can deduce following characteristics from above definition:

- a set is the collection into a whole, and thus the creation of a set is equivalent to the generation of an abstraction;
- a set consists of a definite collection of objects, and so the number of elements is fixed;
- a set consists of a collection of distinct objects, which implies that the elements differ;
- object properties are irrelevant, which stresses the variable nature of elements;
- the ordering of elements within the collection is irrelevant.

It should strike us that sets are completely characterized by a fixed number of elements - that is why we call this the invariant set property. Object properties are irrelevant, which is also shown in following examples:

1. Consider the symbols a, b, c, ..., z. The set consisting of these symbols is called alphabet.

2. Consider the individuals goal-keeper, quarter-back, inside-left, ..., center forward. The corresponding set is a football team.

3. Consider all individuals working in an organization. The set is called staff.

4. The following sets can be defined in mathematics:
   - the set of prime numbers between 2 and 6;
   - the set of odd numbers between 2 and 6.
   These descriptions both indicate a set consisting of the elements 3 and 5, and thus object properties are irrelevant.
In addition we need operations to create new sets from existing ones; these operations make up set algebra. Note that in defining these operations we keep making use of the invariant set property.

The relevant set operations are

1. the union of sets A and B is the set of elements contained in A or B;
2. the intersection of sets A and B is the set of elements contained in both A and B;
3. the difference of sets A and B is the set of elements contained in A and not in B.

We can illustrate these operations by making use of what is generally called Venn diagrams (see figure 3.2). The extensions of the sets A and B are represented by circles here, while the effect of an operation can be simply visualized by shading.

\[ \text{union} \quad \text{intersection} \quad \text{difference} \]

\[ \text{A} \quad \text{B} \quad \text{A} \quad \text{B} \quad \text{A} \quad \text{B} \]

\text{Figure 3.2}

3.4 Types

Although the type concept is indispensable in data processing, it is nevertheless often misunderstood. Irrelevant details are emphasized, while the ones that really matter are almost ignored - generally misusing the set concept. Let us briefly show why.

Assume we wish to construct a conceptual model with descriptions of employees of an organization. Under these circumstances, designers regularly dream up data structures in which information about an individual employee is taken as an element of the set staff. A structure like that emphasizes the set concept. The properties of individual employees are subordinate. It is only found out at a later stage that the properties of the elements play an important part. For example in relational theory we find this out during the normalization steps [Codd 72].
We will take a different approach in this section, based on the justification of the type concept discussed earlier. We consequently emphasize object properties, which are considered as the invariant part of the data in a database. Taking this as a starting point, we define type as follows:

A type is a definite aggregation of distinct properties. Properties from the aggregation are called attributes. An object having properties of the type is called an instance of this type.

The following characteristics can be deduced from above definition:

- a type is the aggregation into a whole, and thus the creation of a type is the generation of an abstraction;
- a type consists of a definite collection of properties, so the number of instances of a type is undefined;
- instances have properties of the type, so instances differ in the values of their attributes;
- objects are irrelevant to a type; two types may be different while they have the same instances;
- the sequence of the properties within the aggregation is irrelevant to types.

The salient point in the type definition is the complete characterization of the type by a collection of properties, which is called the invariant type property. Objects are irrelevant, as will be demonstrated by following examples.

1. Consider the symbols a, b, c, ..., z. The corresponding type is called letter. This type might be defined, like in a dictionary, by the property 'representing a sound', or just 'sound'. It could also be interpreted as a type with irrelevant defining properties. In the latter case, an instance is either a, or b, or ..., or z. This enumeration of instances does, however, not define the type letter - compare the type letter to the set alphabet mentioned above.

2. Consider the properties name, position, football club. The corresponding type is player.

3. Consider the properties name, address, town, department and salary. The corresponding type is employee.
The intension of the set concept indicates a type, which can be defined by the properties abstraction and subject.

The set and type concepts can be generalized into the abstraction concept, which can be defined by the property 'collection into a whole' or just 'membership'. Meaningful abstractions, however, can only be achieved by making use of the specializations set and type.

The above examples illustrate the use of types in various situations. We come across this type of abstraction early on in our lives, like type operations. Some examples are

- generalization (e.g. animal is a generalization of dog, cat, mouse, ...);
- specialization (e.g. cat is a specialization of animal);
- differentiation (e.g. there are other animals besides cats).

We will now introduce these operations more formally. The collection of type operations is called type algebra. Note that the definitions are only concerned with invariant type properties, which implies that instances play no part whatsoever.

The relevant type operations are

- the specialization of types A and B is the type consisting of properties in A or B;
- the generalization of types A and B is the type consisting of properties in both A and B;
- the differentiation of types A and B is the type consisting of the properties in A and not in B.

These operations can be usefully illustrated by Venn diagrams as well. Circles indicate objects with properties X, Y or Z (for example, let us assign to X 'is a road vehicle', to Y 'is a ship' and to Z 'is motorized').

The types A and B are defined as follows:

\[
\text{type } A = X, Z \\
\text{type } B = Y, Z.
\]

Instances of types A and B, so called denotations or extensions, are shown in figure 3.3.
The specialization $S$, the generalization $G$ and the differentiation $D$ of the type $A$ and $B$ are defined by:

\begin{align*}
\text{type } S &= X, Y, Z \\
\text{type } G &= Z \\
\text{type } D &= X
\end{align*}

Denotations of the specialization $S$, generalization $G$ and differentiation $D$ are represented in figure 3.4.
3.5 Discussion

The empty set is defined as the only set not containing any elements; a set like that is obtained, for example, by intersecting two disjunct sets (i.e. sets without common elements).

As opposed to the empty set, the universal set is sometimes introduced as the set containing all sets. As we know, for example from the Russell paradox, this leads to a multitude of contradictions [Ackermans 76, van Dalen 75].

It would seem logical to define a universal type as a type without any descriptive property at all. This implies that a universal type contains all possible observations without corresponding data - the equivalent statement is that the real world (or universe) does not mean anything, really.

Incomplete definitions are sometimes given for the purpose of clarification, which can easily give rise to misconceptions. We must be aware of the fact that A and \{A\} represent different sets. This is obvious as discussed earlier, because in view of the definitions

\[
\text{set } A = \ldots \\
\text{set } \ldots = A
\]

we can find no confirmation for the supposition that these sets are equal - don’t we have two incomplete definitions here, which, in addition, cannot be compared.

Another example involving incomplete definitions is the following: assume that A is an element of set B, and B is an element of set C. Although set C contains an element \{A,\ldots\}, it does not contain set A, for:

\[
\text{set } B = A, \ldots \\
\text{set } C = B, \ldots
\]

So, A does not necessarily have to occur in set C.

3.6 Bibliography

There have been many attempts to formalize the difference between set and type, the main items discussed in this chapter, because of the contradictory opinions encountered in the area of mathematics [Fraenkel 76]. We notice an increasing demand to review mathematical definitions in computer science these days.
Fundamental problems (as the Russell paradox) can be evaded to a certain extent in mathematics, by practicing the 'naive set theory' [Ackermans 76, Halmos 64]. Admittedly, as long as static object collections are considered, this approach yields acceptable results. Things change when collections vary with time, in which case we really want a warning notice to be put up [van Dalen 75].

According to the leading idea of the Von Neumann set theory 'two kinds of individuals are considered, which are distinguished as sets and classes' [Müller 76]. The distinction may there be thought of in this way, that a set is a multitude forming a proper thing, whereas a class is a predicate regarded only with respect to its extension. The two axioms of extensionality are used for the purpose of defining sets and classes. These two axioms are defined as follows:

1. If the set A has the same elements as the set B, then A is the same set as B.
2. If the class C has the same elements as the class D, then C is the same class as D.

The problems of the application of the set concept in computer science are immediate consequences of the axiom of extensionality. All statements about set are synthetic; they are verified by observing extensions.

Different researchers from mathematics and computer science have expressed their doubts about the extensional interpretations of sets and classes (the extensional counterparts of types). Even P. Bernays, one of the leading mathematicians, did not consider classes as real mathematical objects [Müller 76]. Several set theories (e.g. The Zermelo-Fraenkel system with the Axiom of Choice, The Von Neumann-Bernays-Gödel system and The Morse-Kelly system) have been developed to overcome the fundamental problems about sets and classes. In spite of this, none of these can claim to be the right one because they are founded on the same basic concepts [Kunen 80].

In computer science we are faced with constantly changing collections. A changing collection can become empty. In that case the application of set theory leads to weird results, because every type without a corresponding instance (considering the collection as the extension of a set) would have to be equal to an empty set (and, as mathematicians say, there are no more than one of these) [Kent 78]. For example, if neither the registration of ufo's nor the registration of unicorns contained instances, this reasoning would appear to indicate that ufo equals unicorn. This example in fact points us to the essence of the matter. If two types differ, this is not caused by the denotations (extensions), but by the properties in the aggregation. In other words, only type properties are relevant.

Instances are ignored in other areas of computer science as well: in fundamental programming. In the early days of programming much thought has gone into the instances of repetitive commands - remember the period when flow diagrams were in vogue? Real progress was only made after the consideration of the invariant of repetitive
commands [Hoare 69]. Gries has an excellent example [Gries 80].

In the area of data structuring, much work has been done on individual instances, which we find in all foregoing approaches including database abstractions, and in the reported similarity between the operations of union and generalization [Smith 78a]. We have reasoned that these operations are completely different ones (generalization has more in common with intersection). There is no relationship between the operations union and generalization, even if we regard instances as elements (see the Venn diagrams in section 3.4).

Morris [Morris 73] considered the concept of type as used in the area of programming languages. He made the observation that there has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas. He states that although some interesting developments following the extensional approach have been achieved (e.g. the ALGOL-68 system, Scott’s theory and Reynolds’ system) these miss an important aspect of types and concludes that types should not be based on extensional properties.

Sowa [Sowa 84] comes close to the approach in this dissertation by emphasizing the fundamental differences between sets and types. He states that statements about types are analytic; they must be true by intension, while statements about sets are synthetic; they are verified by observing the extensions. Extensions cannot serve as practical definitions, because two expressions with the same extension may have different intensions. Similar observations have been made by Kent [Kent 78] and Sol [Sol 82].
4

SEMANTIC CONCEPTS

4.1 Introduction

When designing databases we aim at the construction of a model of some part of the real world (i.e. a conceptual model) which can be understood by others - in other words, the model should be meaningful to the users. We shall attempt to clarify the word 'meaning' in this chapter.

As a first step, we could of course consult a dictionary, but that would not help us tremendously, because of the many interpretations of the word 'meaning' in most languages.

Semantics is the discipline dealing with the relationship between words and things to which these words refer. So, in fact, we are discussing meanings that can be communicated. Semantics is important in all areas of communication. After a long period in which attention in computer science had been concentrated on syntax and pragmatics, more and more emphasis is put on semantics these days [Elmasri 89].

Natural language semantics is extremely complex and we will certainly not study this subject here - and we don't have to. Conceptual models consist of collections of formal definitions, and therefore their semantics is of limited complexity. We will only consider the interrelationships of these formal definitions and the relationships between these definitions and the reality we are modeling. Finally, we will describe the nature of this semantic representation.

We will start off by discussing some fundamental concepts in the formalization of semantics, after which we shall concentrate on two fundamental aspects of semantics applied in conceptual models: convertibility and relatability. At the end of this chapter we will present a direct consequence of conceptual model semantics: object relativity.
4.2 Semantics

We find a multitude of real world phenomena which man has continuously attempted to record for the purpose of analysis. These observations often have limited meaning at the time of recording; they are frequently imperfectly understood.

We use the concept of datum for the representation of a phenomenon, which we take as a primitive concept. This implies that we cannot define it, but have to rely on explaining it by using an example. If we illustrate the concept of datum by 'windsurfer', 'board', 'sail', 'mast' and 'jib', we must conclude that we have not defined any interrelationship at this stage.

We presume from observing the real world that windsurfer, board, mast, sail and jib belong together somehow. There must be some kind of relationship and we attempt to formalize it, perhaps with the following type definition:

\[
\text{type windsurfer} = \text{board, mast, sail, jib.}
\]

By connecting the data we have introduced a meaning. We can now draw the important conclusion that the meaning lies in the connection between the data, as shown in the diagram in figure 4.1.

![Diagram](image)

**Figure 4.1**

The result is fundamentally more that the sum of the separate data: we call this new totality information. So information is generated by linking data with other data.

In the above example, the data board, mast, sail and jib can be considered as the definition of the datum windsurfer; and indeed, a real windsurfer does consist of a board, mast, sail and jib. This definition might be considered as complete by some, although the datum of, say, 'lee-board' is not part of the definition (the lee-board is apparently not included in the particular environment being formalized).

We can now indicate the semantics of data in relation to the real world phenomena as
well. In our example of the datum 'windsurfer', semantics are defined by the connection between the real phenomenon and the datum, as shown in figure 4.2.

Figure 4.2

Also in this example we may use the term semantics, but not the term information, because the phenomenon itself is not being represented. This implies that this semantics (however important) is outside a conceptual model, and we are consequently unable to formalize it.

4.3 Assertions

A conceptual model is a predefined, partial image of the real world. Such an image evidently consists of descriptions of objects that are considered relevant in the part of the real world we are considering. We shall now devote some attention to a few important semantic concepts.

In the definition of a certain entity we call the latter the subject of attention, the subject being identified by a name. This name is just a reference to the appropriate part of the real world and does not formally contain any quality or real object characteristic. This naming property is called object identity.

What we can say about a certain object is called the predicate, which gives a description of the object. The description again consists of names and hyphens. The use of subjects and predicates in a description can be illustrated as follows.

Suppose we wish to discuss 'grass' and for instance mention its property of being 'green'. We would normally say 'grass is green', and in doing so, we use the two names 'grass' and 'green'. The first name is the subject, the second one the predicate, and we see that
names obtain their meanings (or roles) by occurring in the sentence. It is evident that certain names have completely different meanings in other environments. In the sentence 'green is a color', for instance, it is the word 'green' that is now the subject of attention.

As we have seen above, descriptions can be made by using sentences as grammatical constructs. However, the word sentence has a too wide meaning for our purpose. For example, the group of words 'is grass a color?' is a sentence as well, although it certainly does not describe an object.

A sentence may consist of any group of words, but descriptions are only obtained if the subject and predicate can be indicated immediately and unambiguously. In this manner we have, however, only indicated the required grammatical construct. An arbitrary combination of a subject and predicate does not necessarily have a meaning. Consider the message 'the moon believes in intelligent cheese'. Although we appear to have formulated a statement (we identify a subject and a predicate), the meaning is quite obscure and consequently there is no relationship with reality. We can confidently dismiss this as nonsense.

It is not sensible to consider a conceptual model as a collection of meaningless statements (or propositions). The statements we require must be related to the real world that we see around us: semantics are essential.

We call something that corresponds to reality true, so the statements we are interested in should be true. This is what we call a positive statement or assertion. Assertions consist of a subject related to the matter of the statement, and a predicate which positively qualifies the subject.

If we regard the definitions in a conceptual model as assertions, the model itself is meaningful by definition. Because instances obey a conceptual definition, database contents can be meaningful in a similar way. On the one hand we can guarantee, as will be shown later on, the consistence of both the conceptual model and the database, and on the other we can arrive at meaningful derivations. This would be quite impossible if we just applied the concept of statement or sentence, because the truth would then be questionable.

The relationship between the real world and the conceptual model can be indicated schematically as in figure 4.3.

4.4 Convertibility

As we have seen in the previous section, the relationship between a conceptual model and
the real world can be expressed by assertions. In the following we will concentrate on the types of relationships that we find in conceptual models: these either depend on the internal or external assertion structures.

As we have seen before, assertions consist of a subject and a predicate. Taken in combination, they are considered as a description of the object as it appears in reality. The relationship can be represented as

subject — predicate.

The object 'student' as found in student registration, could be defined as e.g.

\[
\textit{type student} = \text{name, address, faculty}.
\]

This assertion can be depicted as in figure 4.4.

---

Figure 4.3

---

Figure 4.4
Name, address and faculty are regarded as a description of student here and therefore the assertion contains the definition of the (new) type student.

This interpretation differs fundamentally from the one we saw in relational theory, where there is no difference between subject and predicate. All characteristics (including the key characteristic) contribute to the object description, and it is thus obvious that there is not much semantics left.

If we regard assertions as definitions of new concepts, each concept should have a unique definition, because the subject (within the part of the real world considered) is exhaustively defined by the predicate.

On the one hand, a given subject must not be linked to another predicate - i.e. the predicate depends upon the subject:

subject → predicate.

On the other hand, no other subject belongs to a given predicate - subjects depend upon predicates as well, so:

subject ← predicate.

We thus have a convertible relationship between subject and predicate:

subject ⇔ predicate.

The convertibility property is regarded as an invariant of the definitions in a conceptual model, i.e. a time-independent property which cannot be invalidated by any modification to the database.

The practical advantages of convertibility are especially apparent in the data modeling phase. Convertibility clarifies the choice of types and matching attributes. Definitions should be attuned completely to the objects appearing in the relevant part of the real world. A thorough understanding of the application area by the designer is therefore essential.

The types relevant to an application area are listed within a conceptual model. So the type definitions represent the invariants of the real world that we wish to consider. We can then observe the real objects satisfying the type definitions of the conceptual model. In doing this, the objects become instances. Voluminous data collections can be safely created in this way by observing the real world.

Convertibility also has consequences for actual data collections in databases. For
example, assume the relevant type is the type student, as earlier defined. Data about individual students give rise to instances of this type. We can represent these instances in a table, as in figure 4.5.

<table>
<thead>
<tr>
<th>student</th>
<th>name</th>
<th>address</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>Smith</td>
<td>Blockhurst</td>
<td>Physics</td>
</tr>
<tr>
<td>1003</td>
<td>McDonald</td>
<td>Stockton</td>
<td>Mathematics</td>
</tr>
<tr>
<td>1004</td>
<td>O'Connor</td>
<td>Blockton</td>
<td>Mechanics</td>
</tr>
</tbody>
</table>

_Figure 4.5_

There is a requirement in conceptual models for each type definition to be compatible with others. Since instances satisfy a type definition, the same applies to instances: each instance must be acceptable to other instances of the same type.

Assume that an instance with following data is added to the table above

(1005, O’Connor, Blockton, Mechanics).

We now have a problem: (O’Connor, Blockton, Mechanics) represents a complete predicate already present, while 1005 suggests a new subject. We have seen earlier that the subject should only identify, but in no way contribute to the characterization of the object. So on the one hand we have to conclude that we have a new object, while on the other we decide that we are dealing with an existing one. This contradiction must therefore lead to a rejection of the last addition.

When 1004 and 1005 point to really different students (for example two brothers), this type definition is incomplete. A solution will be an extension of the number of attributes (for example with first name or birthdate).

As a consequence, each addition should be checked for a unique characterization of the instance by predicate properties. This is what we call a retrospective check, which is done during operational database use.

A completeness check before the event is more effective, although it requires that the designer understand the application area of the database completely and close co-operation with the users is therefore required.
In summary - convertibility is defined as follows:

Convertibility (within certain contexts) indicates a one to one relationship between assertion subject and predicate.

Applying this to the type definitions we find another definition for convertibility:

Each type definition is unique. There are no different type definitions with the same name or the same collection of attributes.

4.5 Relatability

Having discussed the internal structure of assertions in the preceding section, we shall now consider the relationship between the various assertions in conceptual models. We will thereby have covered all relationships in databases - there are no other semantic database structuring means in the model presented.

Type definitions consist of a subject and a predicate, in which properties are used, as demonstrated by the following type definitions:

\[
\begin{align*}
\text{type student} & = \text{name, address, faculty} \\
\text{type result} & = \text{student, subject, mark} \\
\text{type subject} & = \text{description, lecturer.}
\end{align*}
\]

Relatability is applicable to these type definitions: for example, student in the definition of the type result is linked (related) to the definition of the type student, as figure 4.6 shows.

![Figure 4.6](image)

The relatability for types has not been discussed exhaustively yet. The types contained in conceptual models also have instances with similar relationships. The properties at the
<table>
<thead>
<tr>
<th>student</th>
<th>name</th>
<th>address</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>Smith</td>
<td>Blockhurst</td>
<td>Physics</td>
</tr>
<tr>
<td>1003</td>
<td>McDonald</td>
<td>Stockton</td>
<td>Mathematics</td>
</tr>
<tr>
<td>1004</td>
<td>O'Conner</td>
<td>Blockton</td>
<td>Mechanics</td>
</tr>
<tr>
<td>result</td>
<td>student</td>
<td>subject</td>
<td>mark</td>
</tr>
<tr>
<td>60070</td>
<td>1002</td>
<td>23</td>
<td>A</td>
</tr>
<tr>
<td>60071</td>
<td>1002</td>
<td>24</td>
<td>B</td>
</tr>
<tr>
<td>60072</td>
<td>1003</td>
<td>26</td>
<td>B</td>
</tr>
<tr>
<td>60073</td>
<td>1003</td>
<td>23</td>
<td>C</td>
</tr>
<tr>
<td>60074</td>
<td>1003</td>
<td>25</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 4.7

type level determine the acceptability of instances. We could, for example, have the instances for student and result at our disposal as illustrated by figure 4.7.

In this case relatability implies that, at instance level, a student appearing under result is also found as an instance of the type student. In the example given this is true for each instance of result. Conversely, not every instance of student necessarily appears in the description of result.

Comparing the student set with the set of students under result, we find the following relationship

\[
\{\text{result its student}\} \subseteq \{\text{student}\}.
\]

where the expression A its B means the attribute B of type A.

In our example we have

\[
\{1002, 1003\} \subseteq \{1002, 1003, 1004\}.
\]

Because instance relatability is sometimes called inclusion, we may also refer to it as inclusion or subset invariancy.

We clearly distinguish between invariants at the type level (relatability) and the consequences of this invariant at the instance level (subset invariant). These are different names for one and the same semantic concept, which is defined as follows:

Relatability indicates that an attribute is related to one and only one type with the same name. Each type can correspond to various attributes.
As a consequence:
Relatability indicates that an attribute value is related to one and only one instance in the related type.

4.6 Object relativity

In a communication process sender and receiver play a decisive role. Effective communication is only achieved when both participants interpret the communicated matter in the same way - indeed, severe misunderstanding can arise if this is not the case. The reason for these mishaps cannot always be identified unambiguously: incorrect or incomplete perception of reality or differing communications conventions lead to differing interpretations of the same thing.

It is essential for database designers to realize that interpretation differences in communicating with users can cause problems. Designers often have to deal with a multitude of users who can regard details of conceptual models in different ways. We shall discuss a way of rendering the above problems visible: the object relativity principle.

We can demonstrate the principle of object relativity by using a single abstract object: motor vehicle. We can justifiably attach various interpretations to this object at this stage. The concept is essentially meaningless, because it is not related to other objects. The only thing we can say at this stage is that the object is motor vehicle (see figure 4.8).

![Diagram of object relativity](image)

**Figure 4.8**

One is tempted to visualize a type of which many instances occur in the real world, when the expression motor vehicle is mentioned: cars, trucks and motorcycles on public roads are valid instances.

With an eye on the appropriate application area, we may consequently describe a motor vehicle as a type by recording some of its attributes, possibly as follows:

\[
\text{type} \text{ motor vehicle} = \text{ manufacturer, model, year, price, fuel}
\]

of which figure 4.9 is a diagrammatic representation.
We could also regard motor vehicle as an instance. In this way we could, for example, categorize vehicles according to environmental requirements; the pollution index of the categories would be relevant in this case, as expressed by the type definition

\[ \text{type category} = \text{pollution index} \]

In table form we could record the information as shown in figure 4.10.

We conclude from the above that an object can be a type as well as an instance, whereby the interpretation depends completely on the object’s abstraction. But - beware of potential confusion.

Types and instances within the same abstraction are frequently mistaken one for the other: in the first example we had motor vehicle with the characteristics manufacturer, model, ..., etc, while there was also motor vehicle for an arbitrary instance such as 1024, with the characteristics Citroën, XM, ..., etc. This confuses matters no end and can give rise to problems in data modeling. The object relativity principle only indicates that an object can have various interpretations for different abstractions. We will come across more examples in the following.
We can also consider the type motor vehicle in relation with other types: we could categorize a vehicle fleet by applying the disjunct types motorcycle, truck and car. These types are regarded as specializations of the type motor vehicle, which in itself is taken as the generalization of these categories. These relationships are presented in figure 4.11.

In abstraction hierarchies, generalizations are schematically represented by a line connecting facing corners of rectangles, the generalized types being placed below the specialized ones. Generalization's counterpart is called specialization.

In figure 4.11 we placed motorcycle, truck and car one behind the other, while only one line connects to motor vehicle. This is the usual representation of disjunct specializations - i.e. a motor vehicle is either motorcycle, a truck or a car. The combination of a group of disjunct specializations is called a block, so motorcycle, truck and car constitute a block.

The generalization, together with the properties to be added to it, is described in the definition of the specialization. So the type definitions look like

\[
\begin{align*}
type \ motor \ vehicle & = \ldots \\
type \ motorcycle & = [motor \ vehicle], \ldots \\
type \ truck & = [motor \ vehicle], \ldots \\
type \ car & = [motor \ vehicle], \ldots .
\end{align*}
\]

Generalizations are commonly associated with the verb to be. According to above type definitions, a motorcycle is a motor vehicle with some additional properties.

In a similar way we might have regarded motor vehicle as a specialization of the type vehicle, with other types such as sailing boat and bicycle, as depicted in figure 4.12.
Specializations and generalizations can be regarded as similar abstractions. We do not generate a new object by considering a motor vehicle as a specialization or a generalization. The phrase *is a* abstraction has therefore been coined in the area of artificial intelligence. The complement of the *is a* abstraction is called the *is part of* abstraction, which can be clarified as follows.

In defining the type motor vehicle we considered the latter as a combination (i.e. aggregation) of a number of characteristics such as manufacturer, model, ..., etc, as figure 4.13 shows.

Aggregation is normally indicated by a line connecting the centers of two facing rectangle sides, while the aggregate type is placed above its attributes.

We should not use the term *is a* abstraction (because a manufacturer is obviously not a motor vehicle) but rather *is part of* abstraction. What we can say is: manufacturer is part of the motor vehicle definition.
Aggregation has its counterpart similarly to the case of specialization and generalization. We could well imagine a definition in which motor vehicle is an attribute.

Assume that dispatches play a role in the part of the real world we are interested in — dispatches being defined as the transportation of cargo to certain destinations using certain motor vehicles. The following type definition would apply:

\[
\text{type dispatch} = \text{motor vehicle, destination, cargo}.
\]

Figure 4.14 shows the schematic representation.

\[\text{Figure 4.14}\]

It should be clear that motor vehicle within dispatch plays the role of an attribute.

In summary, we have seen that the object motor vehicle can be found in conceptual models with various interpretations.

The object relativity principle says:

- type, instance, generalization, specialization, aggregation and attribute are just different interpretations of one and the same abstract object.

Note that this does not mean that all different interpretations of an object must in fact occur in a conceptual model. It is conceivable that year (of construction) does occur as an attribute in motor vehicle, but that year is not considered as an aggregation.

What we have been trying to convey so far is that objects should not be provided with fixed interpretations, but rather adapted to the view of the individual user. Data models often support fixed interpretations of objects (for example E-R models, see also the case study in section 2.3). Data models requiring fixed interpretations are artificial barriers to
integration, which often cause increasing complexity and rigid models [Date 90].

4.7 Semantics in data models

Relationships between data in a database can show up in two ways: within and amongst type definitions, both of which we have discussed earlier. The convertibility concept is important in the formalization of internal relationships in type definitions, while relatability formalizes the one amongst type definitions.

Semantics describe data relationships. We can determine to what degree semantics play a role in other database approaches by basing ourselves on the convertibility and relatability concepts. In this section we will determine whether the relational approach uses concepts which indicate data relationships, and also whether these concepts are comparable to the semantic concepts we have seen earlier.

Internal structures of relations play a predominant role in the relational approach. Only the key property of a relation was mentioned in Codd’s initial approach, implying that all attributes depend on the key. What we cannot find in the relational approach is that in addition the collection of attributes, within the part of the real world considered, offers a complete object description as well. The following definition would be perfectly acceptable in the relational approach:

relation student (student id, town).

This definition corresponds with:

type student = town.

Due to convertibility at instance level, representation of data about students is now prevented. Rightly because insufficient attributes are specified.

This example shows that the relational theory could be expanded meaningfully with the concept of convertibility.

A key may contain a number of attributes, according to the relational theory - and indeed, keys in various relations may consist of the same attributes. As a consequence we are unable to represent abstractions in a conceptual model accurately as shown in the following example.

Assume a library with a number of branches. Books may be available at more than one branch. Readers can reserve only books available at the branches. In other words: an
unavailable book will not be reserved at another branch. A relational model of the above rules shows

\[
\text{relation branch (branch id, \ldots)} \\
\text{relation book (book id, \ldots)} \\
\text{relation reader (reader id, \ldots)} \\
\text{relation availability (book id, branch id)} \\
\text{relation reservation (book id, branch id, reader id, \ldots).}
\]

The subset invariant, for example, indicates that the book id’s in relation availability are a subset of the book id’s in relation book. In relational theory this results in the requirement that the book id’s in reservation be a subset of book id’s in book. However, this clashes with the earlier description of reality (because (book id, branch id) in reservation must occur in availability). The failure is due to the compound key of relation availability. Unfortunately, we are unable to signal and remedy this in relational theory. The only recommendation is to minimize the use of compound keys; the suggestion is supported by recent relational theory publications [Date 86b, Kent 83].

Developments in relational theory have made the concept of relatability available [Date 81b]. However, because of the degrees of freedom in the choice of keys, we are not always able to indicate relatability correctly. Relatability must be defined separately in the classical relational model - which can slip the mind easily. These additions can also lead to contradictions, and so the only correct solution is to include relatability as an intrinsic characteristic of type definitions, as is done in this chapter.

### 4.8 Case study

We will now apply the semantic concepts discussed so far to the modeling of a public library lending out books [ter Bekke 88a]. Before setting up the semantic model for the library, here is a brief description of the salient points of the case.

#### Case description

The main objective of the library is the provision of books to members. Membership is acquired by a yearly subscription. Members are either senior or junior members. The lending department consists of a senior sub-department and a junior one. The library distinguishes between books and volumes. Books are administrative objects with properties such as ISBN, title, publisher, year of publication and authors, books for juveniles are additionally tagged 'juvenile book'. Volumes are the real objects on the shelves in the reading-room, only defined by volume number (local identification) and ISBN. Multiple
copies of books may be available. Expiration of loan periods results in fines. Information on loans must be retained during a certain period.

Discussion

The case description is based on an inventory of user requirements, but, in addition to the objectives, it contains further important information.

The list of user requirements is input to the conceptual design, in which the relevant invariants of the real world are determined. The proof of semantic integrity is an important subject in this process. Rather than comparing the database contents directly with reality, the conceptual model is verified in the early stages by applying the two integrity rules for relatability and convertibility.

The abstractions generalization, specialization and aggregation (in this order) are frequently applied during the conceptual design, resulting in a number of explicitly interrelated types, essential to meet user requirements. All relationships are determined by the type definitions, without regard to implementation aspects.

Conceptual design

Members

Because of the division of members into senior and junior ones, a specialization appears appropriate. A more detailed analysis of the case description, however, shows that there are no properties to support this distinction. The following type definitions appear to apply:

\[

type \text{ member} = \text{name, address, birth date, subscription}
\]

\[

type \text{ senior member} = \text{[member]}
\]

\[

type \text{ junior member} = \text{[member]}
\]

A number of properties belong to the type member, but senior and junior members cannot be distinguished by their properties, because only 'being a member' is applicable in both cases. These definitions must however be rejected because they do not comply with convertibility. We therefore find a single aggregation combining both types of member (see attribute category):

\[

type \text{ member} = \text{name, address, birth date, subscription, category}
\]

as also shown in figure 4.15.
Figure 4.15

Books

A similar reasoning results in just one type representing both books for juveniles and adults:

\[
\text{type book} = \text{title, publisher, publication year, authors, category}.
\]

This aggregation is shown in figure 4.16.

Figure 4.16

Although only books for juveniles have a category code, according to the description, we have chosen to provide all books with category codes. The alternative allows missing
data. This makes it possible that books for juveniles are considered as books for adults. The alternative is unattractive because it gives rise to data contamination.

It should be noted that the unique, non-missing, non-modifiable identification (the ISBN) is defined by the type book.

As the library does not require a registration of (potential) authors to function effectively, we will not introduce a corresponding type definition.

The case description does not suggest additional generalization/specialization relationships and we can therefore concentrate on aggregation relationships.

**Volumes**

The case description emphasizes the difference between books and volumes. Books have properties such as title, publisher, etc, but volumes are identified by volume number and the property of referring to the type book. The type definition is therefore:

\[
\text{type volume = book.}
\]

As more than one copy of a book may be available, we have a problem which can be illustrated by referring to possible database contents (see figure 4.17).

<table>
<thead>
<tr>
<th>volume</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>E 125</td>
<td>90 207 1659 X</td>
</tr>
<tr>
<td>E 126</td>
<td>90 207 1659 X</td>
</tr>
</tbody>
</table>

*Figure 4.17*

Volumes E125 and E126 cannot be distinguished by property 90 207 1659 X, and are therefore not convertible. As a consequence, the type volume should be rejected or adapted.

- Does rejection restrict the operation of the library? For instance, is it possible to detect missing volumes?
The introduction of the type volume does not help, because loans not followed by returns are registered under loans, for which the type volume is superfluous. It would only be useful if additional properties such as serial number, purchase date, purchase price and appearance were relevant, in addition to book.

Rejection however, makes it necessary to introduce in the type book an attribute number (of volumes).

- Adaptation of the type volume to:

  \[ \text{type volume} = \text{book, serial number} \]

  also leads to problems. Because of the differentiation between book and volume, books could have been registered without the corresponding volumes being available in the library.

If we require individual local identifications for volumes then we simply modify the type definition of book by including the properties ISBN and serial number:

  \[ \text{type book} = \text{ISBN, serial number, title, publisher, publication year, authors, category.} \]

This amended definition now corresponds to the physical object volume and can be considered as an alternative for the type definition of book.

**Loans**

Certain properties are relevant for loans, such as member, book, loan date and return date:

  \[ \text{type loan} = \text{member, book, loan date, return date, fine.} \]

The corresponding aggregation is given in figure 4.18.

Some properties are provided with default values at the time of lending. The introduction of specializations is not effective here because all properties become meaningful sooner or later.

The description emphasizes that only members of the library can borrow books - obviously, only these loans are legal because of the relatability between loan and member.
Figure 4.18

Total model

Having described all relevant type definitions, we can summarize the interrelationships between book, member and loan in an abstraction hierarchy (figure 4.19).

Figure 4.19

Conversion

The complete set of type definitions for the library is as follows:

- **type book** = title, publisher, publication year, authors, category, number
- **type member** = name, address, birth date, subscription, category
- **type loan** = member, book, loan date, return date, fine.
Keys, well known in relational theory, are not used in semantic models. The relational approach requires keys for all relations. Conversion of type definitions into relations consists at least of the addition of keys to definitions. So, types such as:

\[
\text{type } T = A1, A2, A3
\]

are converted to:

\[
\text{relation } T \text{(TN, A1, A2, A3)}.
\]

In this way, the above type book appears as:

\[
\text{relation book (ISBN, title, publisher, publishing date, authors, category, number).}
\]

The definition for this case in the SQL language is as follows:

```sql
CREATE TABLE BOOK
  ( ISBN (CHAR(10), NONULL),
    TITLE (CHAR(40)),
    PUBLISHER (CHAR(30)),
    PUBLICATION YEAR (INT),
    AUTHORS (CHAR(40)),
    CATEGORY (SMALLINT),
    NUMBER (SMALLINT) )
```

The NONULL specification is used for key attributes, indicating that each table row must have an attribute value. SQL does not define domains other than the standard domains such as CHAR and INT. Relations must contain the domains of all attributes, which leads to significant redundancy in model definitions.

The table descriptions have not yet defined the invariants. The subset invariant can simply not be included in the structure, and keys are specified by the addition of indices, such as:

```sql
CREATE UNIQUE INDEX XB ON BOOK (ISBN).
```

All type definitions are converted to table definitions in this way. References from one type to another result in foreign keys, which cannot be defined using the current SQL standard. The foreign key ISBN in the relation loan can be specified according to the 'pseudo DDL' [Date 81b]:

```sql
```
FOREIGN KEY (ISBN IDENTIFIES BOOK, 
NULLS NOT ALLOWED,
DELETE OF BOOK RESTRICTED,

This specification should be self-explanatory.

4.9 Bibliography

Discussing the subject 'meaning' is not a simple matter, and indeed, the general literature does not help much (see e.g. [Leech 76]). And yet, a discussion on 'data' and 'information' should never be omitted in a dissertation on data modeling. English language textbooks on database (e.g. [Date 86a, Tsichritzis 77, Ullman 80]) rarely discuss the subject; the book by Kent [Kent 78] is an exception, as it gives extremely useful hints, as do the textbooks by Tsichritzis and Lochovsky on data models [Tsichritzis 82], and Teorey and Fry [Teorey 82] on database structure design.

Some authors introduce the fundamental 'datum' concept in such a complicated way that it can hardly be considered as a basic concept. Langefors [Langefors 77], for example, describes elementary data in terms of quadruples <object name, object property, object value, time>. We have conversely introduced the concept of datum as an indivisible object, thereby creating a basic concept. The above quadruple, on the contrary, remains defined by smaller units (and so more primitive concepts).

'Information' was defined as interrelated data in this chapter, and consequently a powerful concept was introduced - the power being comparable to the one of the mathematical set.

Various researchers have initiated studies of semantics in databases since 1975, such as Schmid and Swenson [Schmid 75], Smith and Smith [Smith 77a, 77b, 78a], and ter Bekke [ter Bekke 76]. The subset invariant concept (relatability) appeared for the first time in 1976 [ter Bekke 76], while convertibility was mentioned in 1979 [ter Bekke 79] and 1980 [ter Bekke 80]. Relatability appeared later on under various names in relational theory [Codd 79, Date 81b, Dayal 82].

Schmid and Swenson [Schmid 75] were the first to indicate that no definitive difference can be established between type and attribute, by saying that a type definition can consist of an object as an attribute, while in an other context the same object can occur as an independent type. This can be considered as the first published identification of the object relativity phenomenon. That the idea of object relativity did not result in a new concept was due to the special characteristics of the relational model: relation names acted as type
names, but were not part of the formal theory.

'Object relativity' was defined as a concept in its own right for the first time by Smith and Smith [Smith 78a]. The recognition of object relativity was inevitable because relationships between database abstractions are linked. Kent contributed greatly by his critical views of database abstractions (see [Smith 78a]).

The object relativity concept initially failed to gain much support, because Smith and Smith limited themselves to the relational model. We have chosen an approach such that this concept is integrated quite naturally in the semantic model.
5

SEMANTIC OPERATIONS

5.1 Introduction

Databases are not just used for the convenient storage of data: the retrieval of reliable information from the data is a much more valuable asset. Of course, we require software to carry out such retrievals. The design and implementation of software for retrieval is not a user responsibility - for a variety of reasons.

Modern programming languages provide data structures such as simple variables and arrays, the stored data values belonging to (or rather 'owned by') an application program.

Programs and data structures in the database environment are not exclusively coupled. Databases consist of a collection of common data structures aimed at representing semantics. The preservation of data integrity is an essential requirement, for which the database user can not be made responsible.

Programming languages require database users to have computing knowledge and experience in the design of large application programs. The more so because of the typical control flow nature of most programming languages, forcing the user to specify precisely how the commands should be processed.

An efficient fixed processing route in the database environment can often not be determined because data storage and access may vary with time. That is the reason why control flow techniques are not suitable in a database environment. With data flow techniques things are different, because processing strategies are only determined by the need for data. Therefore, database users doing retrievals only specify what is required, and leave it to the database management system to work out how to do it.

For manipulating data we could make use of mathematical languages like predicate logic or relational algebra. However, firstly these tools are extremely complex and secondly they have few quantifying capabilities (only existential and universal quantifications are available). Finally, the existing mathematical languages cannot handle semantic structures as found in databases.
Because of the shortcomings mentioned above we will introduce Xplain, a non-procedural data manipulation language developed by the author [ter Bekke 88b]. This language is based on semantic data modeling concepts. It is imperative to make use of the structure contained in the Xplain model; in fact, data manipulation capabilities and restrictions are completely dependent on this structure.

We will illustrate the use of the Xplain data manipulation language by many examples, starting off with very simple selections, turning to extensions (crucial to the language) and finally to modification commands. A formal description of the language is given in the next paragraphs.

5.2 Selections

The most important expression of the Xplain language is the one of selection, generally represented as

\[
<\text{type name}> \text{its} <\text{attributes}>
\text{where} <\text{condition}>
\]

The selection expression indicates that we can obtain instances from the database which

- were inferred from the type with \(<\text{type name}\>)
- have attributes as specified by \(<\text{attributes}\>)
- satisfy \(<\text{condition}\>.

Not specifying the clause \(<\text{its}\> <\text{attributes}\>) results in complete instances. Not specifying the clause \(<\text{where}\> <\text{condition}\>) results in all instances.

The simplest retrieval is the selection of data satisfying certain criteria. Let us look at a database with a model only containing the composite types department and employee. Each organizational department is defined by department name and business_town; in addition, we have name, home_town, department and salary for each employee. The type definitions are therefore as follows:

\[
\begin{align*}
\text{type department} & = \text{department name, business_town} \\
\text{type employee} & = \text{name, home_town, department, salary}.
\end{align*}
\]

The complete abstraction hierarchy is shown in figure 5.1.
The model tells us, amongst other things, that every employee works for one department only (remember relatability), and that the role attributes business_town and home_town are comparable; both are related to the same type: town. We will now demonstrate a few selections based on this simple model.

Example 1: Select data of the employee with the identification E3.

Employee data can be found in the type employee, where we limit ourselves to the instances containing the identification E3. Making use of the unique identification we obtain

"get employee 'E3'."

The keyword get indicates that the selection result is required as the output.

Example 2: Select employees living in Guilding.

And again, the subject of this selection is the type employee. If, in addition to the identification, we are interested in name and department, the result of this selection could be

"get employee its name, department."
Because we only want employees living in Guilding, we add following condition:

```
home_town = 'Guilding'.
```

So, the required data can be obtained by

```
get employee its name, department
where home_town = 'Guilding'.
```

Example 3: Select commuters.

We are interested in employee data again. The data we want for every employee is

```
employee its name, home_town, department.
```

According to the intended selection, employee home_town must not be equal to department business_town. Department and employee are linked: each employee works for one and only one department, so we must make use of this relationship between employee and department.

The department corresponding to an employee is defined by the attribute department of type employee. We are only interested in the value of attribute business_town for this department, so

```
department its business_town.
```

The complete condition is therefore

```
home_town \neq department its business_town.
```

And thus the final expression for this problem is

```
get employee its name, home_town, department
where home_town \neq department its business_town.
```

The results of above retrievals consist of (partial) sets of the actual data in the database. We can come across retrievals where selected data collections by themselves are irrelevant, but where only characteristics of these sets are of importance, such as number of elements, sums of the elements or maximum values. The following examples should clarify the idea.

Example 4: How many employees work in Guilding?
We can depart from the outcome of example 2 by applying the set function *count* to the selected set. Because attributes are now no longer required, we can say

```
get count employee
    where department its business_town = 'Guilding'.
```

Following retrieval functions are available:

- **count**: returns the number of elements in a set;
- **max**: returns the maximum value of a set of attribute values;
- **min**: returns the minimum value of a set of attribute values;
- **total**: returns the sum of the values of a set of attribute values;
- **nil**: returns 'true' if the selected set equals the empty set, otherwise 'false';
- **any**: returns 'false' if the selected set equals the empty set, otherwise 'true';
- **some**: returns a random element of the selected set.

Default values may be returned by the functions *max*, *min*, *total* and *some*. The following examples demonstrate the usage.

Example 5: What is the sum of all salaries?

```
get total employee its salary.
```

Example 6: What is the highest salary?

```
get max employee its salary.
```

Example 7: Are there any employees earning more than 50.000?

```
get any employee
    where salary > 50000.
```

Example 8: Select the name of an arbitrary employee in department Purchase.

```
get some employee its name
    where department its department name = 'Purchase'.
```

This should have clarified the way in which selection expressions can be used. A next possible step is to retrieve information by using data contained in a database, for which we always need more than one command. As will be shown in the next section, the extension command, as it is called, plays a predominant role.
5.3 Extensions

The general expression for the extension command is as follows:

\[
\text{extend \ <type name> \ with \ <attribute name> = <extension definition>}
\]

The meaning of the new temporary attribute with \(<attribute name>\) of the type with \(<type name>\) is described by \(<extension definition>\) as demonstrated by following illustrations. The first examples in this section refer to the employees and departments model discussed above:

\[
\begin{align*}
\text{type department} & = \text{department name, business_town} \\
\text{type employee} & = \text{name, home_town, department, salary}.
\end{align*}
\]

Example 9: Provide an overview of the departments, including the number of employees.

In addition to data residing in the database (indicated by a selection expression), inferred data are required here as well. The latter information must be created before we can carry out a selection.

The number of employees per department can be found from the type employee, the instances of which are collected in mutually disjunct groups. We proceed by determining the cardinality of each group by

\[
\begin{align*}
\text{count employee} \\
\text{per department}.
\end{align*}
\]

These numbers must now be allocated to the corresponding department by defining a temporary extension to the type department, which is assigned the value of the obtained number. The type department is extended as follows:

\[
\begin{align*}
\text{extend department with number of employees =} \\
\text{count employee} \\
\text{per department.}
\end{align*}
\]

We can read the definition of the type department after this extension as

\[
\text{type department} = \text{department name, business_town, number of employees},
\]

the last attribute indicating the number of employees for each department. Now all relevant data have been collected, we can select the required data. So, the complete solution is


*extend* department *with* number of employees =

  *count* employee

  *per* department.

*get* department *its* department name, business_town,

  number of employees.

Because all attributes are required in the result, the above attribute list could be omitted.

The extension command represents a powerful tool for solving similar problems in a simple way, as shown in the next examples.

Example 10: Select departments with more than 100 employees.

*extend* department *with* number of employees =

  *count* employee

  *per* department.

*get* department *its* department name, business_town

  *where* number of employees > 100.

Example 11: Find the number of departments with more than 100 employees.

*extend* department *with* number of employees =

  *count* employee

  *per* department.

*get* count department

  *where* number of employees > 100.

Apart from the temporary addition of an attribute to a type, we may also store the result of a set function temporarily for subsequent use in a selection expression. For this purpose we introduce the value command:

*value* <name> = <value definition>

Example 12: Which department has most employees?

We need the number of employees in each department first:

*extend* department *with* number of employees =

  *count* employee

  *per* department.
Next, we determine the maximum value:

\[ \text{value maximum} = \max \text{ department its number of employees.} \]

The final selection of the required department(s) is performed by

\[ \text{get department its department name, business\_town} \]
\[ \text{where number of employees} = \text{maximum.} \]

Three commands were required to formulate the complete solution, whereby we must realize that the result of this retrieval could well consist of more than one instance.

The next examples relate to the model defined by:

- type supplier = company name, business\_town
- type article = make, price
- type user = user name, home\_town
- type customer = supplier, user
- type supply = customer, article, quantity,

and corresponding to the hierarchy shown in figure 5.2.

![Figure 5.2](image)
Example 13: Which users are customers of all suppliers?

We can indicate the desired result in pseudo Xplain as

\[
\text{get user its user name, home\_town} \\
\text{where 'customer of all suppliers'}. 
\]

We can also say for the condition 'customer of all suppliers':

\[
\{\text{customer its supplier per user}\} = \{\text{supplier}\},
\]

having made use of relatability in the conceptual model, as all suppliers are instances of the type supplier.

In order to evaluate this condition we need sets of supplier identifications per user for the comparison. We could also evaluate a simpler condition by making use of data semantics again, because we know that supplier identifications in the type customer only refer to existing supplier identifications in the type supplier:

\[
\{\text{customer its supplier per user}\} \subseteq \{\text{supplier}\}. 
\]

Because every supplier occurs at most once for a user (for convertibility reasons), we can simplify the first condition:

\[
\text{count} \{\text{customer per user}\} = \text{count} \{\text{supplier}\}. 
\]

As a consequence, we have to determine the number of corresponding instances of customer for each user. So we extend the type user temporarily with the attribute number, defined as

\[
\text{extend user with number} = \\
\text{count customer} \\
\text{per user}. 
\]

In addition we need the number of suppliers, which we can do by

\[
\text{value number of suppliers} = \\
\text{count supplier}. 
\]

A summary of the set of commands for this problem is therefore
extend user with number =
      count customer
      per user.

value number of suppliers =
      count supplier.

get user its user name, home_town
     where number = number of suppliers.

We made appropriate use of the semantic concepts of relatability and convertibility in previous example. We might wonder whether it was not a special case which happened to suit the purpose. However, we will demonstrate that this class of problems is huge, by an example which will not initially make use of semantic concepts.

Example 14: Select suppliers with the same users as customer as the supplier identified by S3.

The condition to be evaluated here is

\{customer its user per supplier\} ⊇ \{customer its user where supplier = 'S3'\}.

The sets are not relatable and we must therefore not compare cardinalities at this stage. Conversely, the use of cardinalities is appropriate if we intersect both parts of the condition with the customers of supplier S3. Having done that, we obtain the following equivalent condition:

\{\{customer its user per supplier\} \cap \{customer its user where supplier = 'S3'\}\} = \{customer its user where supplier = 'S3'\}.

We may now compare cardinalities because there is a subset invariant (left hand expression ⊆ right hand expression). The complete solution is therefore:

extend user with S3 customer =
      any customer
      where supplier = 'S3'
      per user.

extend supplier with number of S3 customers =
      count customer
      where user its S3 customer
      per supplier.
value number S3 =
  count user
  where S3 customer.

get supplier its company name, business_town
  where number S3 = number of S3 customers.

The next example is now trivial.

Example 15: Select suppliers with the same users as customer as the supplier with identification S3.

extend supplier with number of customers =
  count customer
  per supplier.

extend user with S3 customer =
  any customer
  where supplier = 'S3'
  per user.

extend supplier with number =
  count customer
  where user its S3 customer
  per supplier.

value number S3 =
  count user
  where S3 customer.

get supplier its company name, business_town
  where number S3 = number and
    number = number of customers.

The last examples in this section will be based on the following model:

type dispatch = driver, motor vehicle, destination, cargo
type driver = name, address, home_town
type destination = company, address, town
type motor vehicle = make, vehicle type, year, chassis number
type vehicle type = manufacturer, engine capacity, fuel.

The abstraction hierarchy is given in figure 5.3.
Example 16: Select drivers operating on all destinations.

We may formulate following condition, in analogy to example 15:

\[
\{\text{dispatch its destination per driver}\} = \{\text{destination}\}
\]

We should not just count the number of dispatches per driver in this example, because this is not necessarily the number of destinations. We come across problems like these when instances of a type with more than two attributes are to be counted (because we cannot apply the convertibility principle).

It will do to just count the number of destinations per driver, as all destinations in dispatch are part of the type destination. The solution is as follows:

\[
\text{extend driver with number = count dispatch its destination per driver.}
\]

\[
\text{value number of destinations = count destination.}
\]

\[
\text{get driver where number = number of destinations.}
\]
Example 17: Which vehicle type is most frequently used in dispatches?

We must first determine the number of dispatches per vehicle type. But, because vehicle type is not an attribute belonging to dispatch, we need two steps:

1. the number of dispatches per motor vehicle;
2. the totals found in step 1, per vehicle type.

We can also do this in one step by counting the number of dispatches per motor vehicle its vehicle type, as in:

\[
\text{extend vehicle type with number = count dispatch per motor vehicle its vehicle type.}
\]

\[
\text{value maximum = max vehicle type its number.}
\]

\[
\text{get vehicle type where number = maximum.}
\]

5.4 Modifications

The following database modifications can be distinguished:

- insertion of instances;
- deletion of instances;
- update of instances.

We shall present simple examples of these categories below.

The insert command has the structure:

\[
\text{insert <type name> <instance identification> its <allocation list>},
\]

where <allocation list> is made up of a series of

\[
<\text{attribute} > = <\text{value}>, \text{separated by commas.}
\]

Suppose we have a type employee, defined by:
type employee = name, home_town, department, salary.

The use of the insert command can be demonstrated by the following.

Example 18: Add an employee with the attributes identification (E20), name (Fletcher), home_town (Guilding), department (D3) and salary (15.000).

The resulting insert command is:

```
insert employee 'E20'
    its name = 'Fletcher',
    home_town = 'Guilding',
    department = 'D3',
    salary = 15000.
```

It should be remembered that an insert command is only executed if it leaves all invariants undisturbed. For example, if the type department exists, the database management system checks whether D3 defines an already existing (i.e. available in the database) instance of the type department. Of course, it will also determine the uniqueness of E20 in the type employee.

Deletion of instances from a database are carried out by:

```
delete <type name>
    where <condition>.
```

An example of the use of this command:

Example 19: Remove all employees whose salary is zero.

```
delete employee
    where salary = 0.
```

The delete command may result in a check on invariants as well.

Database updates are carried out by the command:

```
update <type name> its <allocation list>
    where <condition>.
```

Note that the condition evaluation precedes the allocations.

The following update to a database could well be applied:
Example 20: Increase the salaries of employees earning less than 50.000 by 5%.

\[
\text{update employee its salary = 1.05 \times \text{salary}} \\
\text{where salary < 50000.}
\]

5.5 Evaluation

An important aspect in the development of formal languages is the avoidance of confusing independent concepts; this principle is often called the orthogonality principle [Date 85]. Query languages are provided with a simple, transparent and consistent structure by applying this principle; moreover learning curves are better and the languages are simpler to use and memorize.

Let us examine whether the Xplain language obeys the orthogonality principle. These are the important aspects:

- **the number of concepts should be as small as possible (though not necessarily minimal).**

  Xplain offers but a limited number of commands. The set functions *any* and *nil* are essentially superfluous, because empty sets can also be identified by the *count* function. These functions have only been added because they often fit user solutions well.

- **a given syntactical construct should have the same meaning in various situations.**

  Each syntactical construct in the query language has the same meaning, independent of the context. The only restrictions are due to the nature of the conceptual model (for instance, nonexistent types and attributes cannot be referenced).

- **given semantic constructs should have the same structure under all conditions.**

  If an identification is used within a selection, the latter might be:

  \[\text{get salesman 'E3' its name, address.}\]

  It would, however, seem to be more appropriate to say:

  \[\text{get salesman its name, address} \\
  \text{where salesman = 'E3'.}\]
On the other hand, this would imply that

\[
\text{get salesman its name, address} \\
\text{where salesman < 'E3' or home_town = 'Selling'}. 
\]

is legal as well. But this means that the identification is used as a descriptive property, which is incompatible with the concept of object identity [Khoshafian 88].

In all other cases the same constructs can be used for the solution of similar problems. Restrictions are only caused by the nature of the conceptual model.

- **each command should be atomic.**

  Commands are executed either completely or not at all in the query language. A delete command is ignored completely, for example, if it implies the removal of referenced instances.

- **the use of query languages should not be subject to arbitrary restrictions.**

  Such a restriction appears to have been applied to the extend command: base types (excluding the enumerated ones) cannot be extended.

This might appear as a disadvantage at first sight. Let us, however, briefly consider following model:

\[
\text{type turnover = salesman, date, total.} 
\]

A query that would cause difficulties is, for example,

What is the total turnover per date?

The query can be formulated if date was defined as a composite (or enumerated) type (because all date instances are given). If date was defined as a base type, however, this could result in a total to which some had not contributed (there may have been no turnover on certain dates). If we were to select the date with the minimum turnover from this total, we could be led to the wrong conclusion that this is the date with the lowest turnover. The acceptance of an extension to a base type therefore results in undesirable asymmetry.

- **the execution of commands should not be associated with unexpected side effects.**

  Commands have unmistakable effects because of relatability and convertibility.
nesting expressions should be subject to as few restrictions as possible.

Attribute names in commands can always be replaced by references to attribute names. A deliberate distinction has been made between fixed and variable properties, whereby variable properties are always converted to fixed ones by the extend command.

a language must have been defined carefully.

Definitions should remove any doubts about language syntax and semantics. The appendix contains an exhaustive description of the Xplain language.

a language should enhance reproducibility.

Every problem essentially only has one solution, and consequently different persons should come up with similar solutions.

Reproducibility is favored in the query language by the limited number of commands. Moreover, we only have two possibilities in data retrieval: either 'up' by the extend command, or 'down' by the its construct. The set functions any, nil and count allow some more freedom.

a language should be helpful in the retrieval of useful information.

Unambiguous type relationships are defined by single identifications in the language we are discussing, and 'connection traps' are therefore absolutely impossible.

there should only be few opportunities for the introduction of errors.

The user can be protected simply from making some fatal errors (such as the destruction of a large database), because the language's definition and manipulation sections have been separated. User errors are also minimized by the other points mentioned so far.

a language should have efficient programming, translating (possibly interpreting) and processing capabilities.

The Xplain language supports only few but effective commands and is consequently easy to learn and use. Processing time is at most proportional to the number of instances in use.
• a language should support the model in use.

Data manipulation commands cannot be applied by just any user to modify the conceptual model. The user can retrieve information from the database, and, in addition, is protected from retrieving false information by improper use of the database.

5.6 Query conversion

Several data manipulation languages have been proposed for the relational model. From these, the SQL language has become the standard interface for relational database management systems. The SQL language offers set oriented operations based on blocks like:

\[
\begin{align*}
\text{SELECT} & \quad <\text{attribute list}> \\
\text{FROM} & \quad <\text{relation list}> \\
\text{WHERE} & \quad <\text{condition}>.
\end{align*}
\]

<Condition> may again contain the base block, enabling us to formulate queries in one complex (nested) structure. If all relation attributes are required, the <attribute list> may be substituted by just *.

Let us have a look at some examples based on following type definitions:

\[
\begin{align*}
type \text{ student} & = \text{name, town, faculty} \\
type \text{ subject} & = \text{description, lecturer name} \\
type \text{ result} & = \text{student, subject, date, mark}.
\end{align*}
\]

The SQL solutions depend on following equivalent model:

\[
\begin{align*}
relation \text{ student} & \quad (\text{stud id, name, town, faculty}) \\
relation \text{ subject} & \quad (\text{subject id, description, lecturer name}) \\
relation \text{ result} & \quad (\text{result id, stud id, subject id, date, mark}).
\end{align*}
\]

An SQL model definition was given in chapter 4.

Example C1: Select students from Edinburgh.

This simple query is defined by a single get command:

\[
\text{get student where town = 'Edinburgh'}. \]
The conversion to SQL is similarly straightforward because information from only one relation is required:

```
SELECT *
FROM student
WHERE town = "Edinburgh".
```

Example C2: Select results in the subject Physics.

This problem only requires a single command in semantic queries, because of relatability:

```
get result where subject its description = 'Physics'.
```

Semantic queries with its constructs can be easily translated into SQL:

```
SELECT *
FROM result
WHERE subject id IN
  (SELECT subject id
   FROM subject
   WHERE description = "Physics").
```

We could also have defined this in a flat join expression:

```
SELECT r.result id, r.subject id, r.date, r.mark
FROM result r, subject s
WHERE r.subject id = s.subject id
AND s.description = "Physics".
```

The shallow expression is sometimes useful when handling set functions.

Example C3: Select students who did not pass the subject Physics.

Two commands are required in this example:

```
extend student with fail =
  nil result
where mark ≠ F and subject its description = 'Physics'
per student.
```

```
get student where fail.
```

We recommend defining the query in SQL in two steps as well, the first step
corresponding to the extension. We require following auxiliary table (VIEW):

```sql
CREATE VIEW fail (stud id) AS
SELECT stud id
FROM student
WHERE stud id NOT IN
  (SELECT stud id
   FROM result
   WHERE mark ≠ F
   AND subject id IN
   (SELECT subject id
    FROM subject
    WHERE description = "Physics"));

SELECT *
FROM student
WHERE stud id IN
  (SELECT stud id
   FROM fail).
```

Solutions for the set functions *any* and *nil* (cf. IN and NOT IN) are not always feasible for other set functions, because of the joins which are required for other set functions. The following two comparable examples should explain.

Example C4: Select students with more than eight passes.

```sql
extend student with number =
  count result
where mark ≠ F
per student.

get student where number > 8.
```

Two commands are required for SQL as well, and, in order to obtain the pairs (stud id, number), we use the join:

```sql
CREATE VIEW candidate (stud id, number) AS
SELECT stud id, count (*)
FROM student s, result r
WHERE s.stud id = r.stud id AND r.mark ≠ F
GROUP BY stud id
```

followed by the selection:
SELECT *
FROM student
WHERE stud id IN
    (SELECT stud id
     FROM candidate
     WHERE number > 8).

Example C5: Select students with less than three passes.

The first step can be copied from example C4, but the second step involves a more complicated selection. This is because the view only contains students with at least one result (resulting from the join). In order to select students with no results at all (because they also comply with the query), we add the UNION:

SELECT *
FROM student
WHERE stud id IN
    (SELECT stud id
     FROM candidate
     WHERE number < 3)
UNION
SELECT *
FROM student
WHERE stud id NOT IN
    (SELECT stud id
     FROM result).

Example C6: Select students with the maximum number of passes.

extend student with number =
    count result
    where mark \neq F
    per student.

value maximum =
    max student its number.

get student where number = maximum.

The first step is as in previous example:
CREATE VIEW candidate (stud id, number) AS
SELECT stud id, count (*)
FROM student s, result r
WHERE s.stud id = r.stud id AND r.mark ≠ F
GROUP BY stud id

In general, a set function is evaluated separately by:

CREATE VIEW maximum (value) AS
    SELECT max (number)
    FROM candidate

Also in this example one must be aware of the fact that a NULL can result from this step. Assuming that this is not the case, the final selection consists of:

SELECT *
FROM student
WHERE stud id IN
    (SELECT stud id
    FROM candidate, maximum
    WHERE number = value).

Above guidelines should enable the transformation of any semantic query to an equivalent SQL query.

This section illustrated some pitfalls in the relational language SQL. It is clear that the semantic principles of databases (esp. relatability), as used in the Xplain language, are essential for a correct query formulation.

Evaluation

The conversion is given for all semantic manipulation commands. It is clear that, besides the query structure, also conditions (as < and >) play a role in the SQL query formulation. Illustrations are given in the examples C4 and C5. The fact that conditions play such a role is caused by the join operation which is necessary in SQL formulations with set functions.

5.7 Bibliography

Query language developments have been strongly stimulated by relational model publications by Codd (see also [Date 81a]). Query languages were developed because of
the requirement to make databases accessible to a wider community of users, even to the non-professionals. The relational algebra (with operations such as join, project and divide) developed by Codd, gave rise to many query languages based on relational database management systems (such as SQL for DB2 and ORACLE; QUEL for INGRES). Nine relational query languages have been compared by Lacroix and Pirotte [Lacroix 77]. Date [Date 85] criticizes the language SQL based on accepted programming principles. It is interesting to note that in query languages operations are not geared to the characteristics of data models (relatability and convertibility). Strictly speaking this is not surprising because relational algebra dates back to the time when semantics in data models was unknown (in fact databases had not been developed in the early stages of relational algebra). Even the primary key, a phrase coined by Codd, does not appear in relational query languages.

This lack of data manipulation semantics can lead to serious misunderstanding; the 'connection trap' is a clear example. We will show that the suggested approach is fundamentally infeasible in relational theory. It is being upheld that invariants are not used, but we argue that they should be used in order to arrive at the correct solution.

In our example we have a collection of relations P, Q and R (we chose letters because relation name semantics are irrelevant) with the attributes x, y, ...:

\[
P(x, \ldots)
\]
\[
Q(x, y)
\]
\[
R(x, \ldots).
\]

Note that keys have not been identified because they play no role in data manipulation. The query now is:

Select the y’s corresponding to all x’s (in a semantic environment a relevant query would be: select the suppliers having all users as a customer).

We require the divide operation for this query in relational algebra: the left hand part contains the relation between x and y (in our example Q) and the right hand part contains the relation with all x (say, P):

\[
Q[x \div x]P
\]

This statement will only produce the correct answer if all x are available in P, in other words, if and only if the subset invariant for x is valid. If this is not the case, a relation containing all x should be created first (i.e. generate the subset invariant).

It should be noted that the subset invariant for y in the relational expression is not required for above query, contrary to semantic manipulations. Conversely, it must be
used if the query changes to: which \( y \) do not correspond to all \( x \). This case demonstrates the instability of relational expressions.

These and other observations [ter Bekke 76, 77, 78, 85]) resulted in the Xplain language, which has been designed in the light of relatability and convertibility.

Queries like the one above can only be formulated by making conscious use of invariants. Moreover, fancy constructions leading to the 'connection trap' cannot be set up. The only valid constructions are the ones containing a type identification (and thus correspond to decomposition or generalization), which implies that only sensible operations can be specified.

Ignoring semantics is an extra drawback in modern data processing: almost all operations are of squared order, which means that for e.g. 100 elements the number of required operations is 10 000 \( (=100^2) \). For a detailed explanation see [ter Bekke 85]. By the application of semantics the linear order can be achieved.

Let us finally briefly look at future processing strategies. Parallel processing is expected to gain in importance. The structure of relational algebra operations allows operations never to have more than two operands, which implies that the optimal process structures result in binary trees. The operations discussed in this chapter, conversely, result in n-ary hierarchies, generally allowing n extensions to one type. N-ary hierarchies are normally shallower, which is an advantage for parallel processing (an example is given in [ter Bekke 85]).
6

SEMANTIC INTEGRITY

6.1 Introduction

Data models are useful in managing properties of data processing applications. These applications are generally characterized by

- static properties, such as objects, object properties and object relationships;
- dynamic properties, such as object operations, object property operations and relationships between operations;
- object and relationship integrity rules.

Data models can be applied almost naturally in the presence of the corresponding components: structures, operations and constraints [Codd 81]. This chapter has been fully dedicated to constraints in data languages, with emphasis on the Xplain data language.

6.2 Constraints

Constraints are defined as rules guaranteeing semantic integrity. Data models differ in the way constraints are handled: hierarchical and network model constraints are tightly linked to structuring concepts (record, set and segment definitions), of which the parent-child and owner-member relationships are logical examples. Relational models, on the other hand, have only one constraint recorded structurally in relations; obviously, this structural constraint is the one about relations consisting of a certain number of attributes (in other words, the relation is in the first normal form). Additional constraints in relational models must consequently be specified separately.

Having experienced the varying application of constraints, we generally keep to following classification:
• inherent constraints;
• explicit constraints;
• implicit constraints.

Inherent constraints are defined by concepts providing structure; they constitute the semantic base of data models and must always be adhered to.

Explicit constraints, on the other hand, must be specified explicitly in addition to the given structure.

Implicit constraints, finally, are a logical consequence of existing inherent and explicit constraints, which implies that the constraint category is determined by the nature of a given data model.

Constraint specification languages are given much attention in literature. These languages are obviously only effective if constraint collections can also be verified and if algorithms guaranteeing adherence to constraints are available.

Inherent constraints in database management systems for given data models have been developed most extensively, because certain meta information about the constraint is recorded in data dictionaries. Consistency control and adherence checks are therefore easily managed. As constraints are embedded in the nucleus of database management systems, adherence checks generally cause only minimal loss in performance.

Explicit constraints are rules capable of recording static and dynamic properties which cannot be specified by the structuring means of given data models. By static property we mean a statement which is true for every valid database state with a given data structure. Dynamic properties describe legal transitions from one valid database state to another. As a result, there are two methods for specifying explicit constraints:

• \textit{static specification} where rules indicating valid database states are set up.

• \textit{dynamic specification} where operational rules indicating valid database state transitions are set up.

It would appear logical to specify static properties statically - although we do not really have to. Because every constraint must be enforced on the level of state transitions, specification at that level is always possible. The reverse is not true: real dynamic constraints are not static and can thus not be specified without operations.
This leads us to three levels at which constraints can be specified:

- inherent;
- static;
- dynamic.

This classification is extremely important for the database designer. The effectiveness of a design depends strongly on the constraints specifications; that is why the above level classification is so important.

Relatability and convertibility have been introduced as inherent constraints in order to avoid contradictions in type definitions. As static and dynamic constraint categories have now been introduced as well, the likelihood of conflicts between inherent and other constraints has increased. The new categories will have to be described in a similarly careful way as we did the inherent constraints.

The orthogonality principle should show us the way: every constraint should have essentially only one possible specification. This makes the specification of disjunct constraint categories an objective. If this objective can be realized, constraint verification is substantially simplified: it is superfluous to consider all possible combinations of inherent, static and dynamic constraints during the investigation of the consistency of the collection of constraints. The verification of a constraint can then be limited to the category to which the constraint belongs.

### 6.3 Inherent constraints

**Type specifications**

Every conceptual model is subject to the inherent constraints of relatability and convertibility. The necessary checks are carried out by the database management system without additional user input. For example, if we have the type definitions

\[
\text{type } A_1 = B_1, B_2, \ldots, B_n \\
\text{type } A_2 = B_1, B_2, \ldots, B_m,
\]

these will be rejected for positive \(n\) equaling \(m\).

Definitions where attributes are related to the specified type are valid; the attribute is a role attribute and must be prefixed in the specification, as in
type employee = ..., manager_employee.

On the other hand,

type employee = ..., employee, ...

is meaningless.

Above valid type definition is represented by the abstraction hierarchy in figure 6.1.

Figure 6.1

However, the type definition

\[ type \ A = B, \ ..., \ B, \ ... \]

is meaningless as well.

Role attributes should be used in cases where attributes are specified twice; at least one attribute should be prefixed:

\[ type \ A = B, \ ..., \ X_B, \ ... \] or
\[ type \ A = Y_B, \ ..., \ X_B, \ ... \]

We can also have nonsensical specialization definitions, as in

\[ type \ A = [X_A], \ ... \]
\[ type \ A = [X_B], \ ..., \ [Y_B]. \]

Types must not be defined as their own specializations, as in the first line above, neither must a type contain a double specialization of a given type, as in the second one.

Value ranges

We have not discussed type representation matters so far - and it is irrelevant for
conceptual models, because the definition and the relationship between concepts is emphasized here.

In the design of conceptual models it should be remembered that database management systems must be supplied with information about the values to be expected for the types. This is what value range specifications are for:

- *representation*, e.g.
  - (In) up to n digit integers;
  - (Rn,m) reals with up to n digits before, and m after the decimal point;
  - (B) logic values (boolean), true or false;
  - (An) up to n characters containing alphanumeric strings.

- *enumeration*, e.g.
  the type working day ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday').

- *range*, e.g.
  employee age (16..65), or a positive integer (1..*).

- *pattern*, e.g.
  an article number (XX9999, two letter department code, 4 numbers for article number).

When specifying value ranges we should watch out for redundancy. We must realize that specifications determine possible values for a simple type, independently of the usage in the composite type definition, because attributes in composite types are subject to relatability requirements in conceptual models.

6.4 Static constraints

Static constraints are statements which are true for every valid database state with a defined structure. Static constraints can hardly be represented inherently (i.e. with structural concepts) or dynamically (i.e. with modifying operations).
Query languages should enable us to verify that a database obeys certain static constraints, without making use of the get command. It would be senseless to set up static constraints without being able to find out whether or not they are obeyed.

This means that we can only consider extend and value commands for static constraints. As both commands are generic by nature (i.e. they are defined at type level), only generic, static constraints are valid.

We can find frequent examples in literature where the above three category division is not supported, e.g.

- The total of employee salaries does not exceed a given maximum.
- The number of staff per department is subject to a minimum and a maximum value.
- Invoices at least show the invoice amount.

It is surprising that we often have cases of aggregate information which can be obtained by a set function. We can set up these constraints by extend or value commands, in which both the extension and the variable must be considered as virtual items. They consequently obtain their value(s) exclusively from the database management system. Consider the conceptual model

\[
\begin{align*}
\text{type} & \quad \text{invoice} & = & \text{customer}, \text{date}, \ldots \\
\text{type} & \quad \text{invoice line} & = & \text{invoice}, \text{article}, \text{quantity}, \text{unit}_\text{price}, \ldots
\end{align*}
\]

to which we add some extensions, such as

\[
\begin{align*}
\text{extend} & \quad \text{invoice line with amount} = \\
& \quad \text{quantity} \times \text{unit}_\text{price}. \\
\text{extend} & \quad \text{invoice with invoice_amount} = \\
& \quad \text{total} \text{ invoice line its amount per invoice}. \\
\text{extend} & \quad \text{invoice with number of lines} = \\
& \quad \text{count} \text{ invoice line per invoice}.
\end{align*}
\]

We have defined the virtual attribute values ourselves. However, the requirement that the number of invoice lines be \( \geq 1 \) has not been enforced, and the value range \((1\ldots\ast)\) is assumed.
We have explained in the above that conditions are specified by value ranges, which implies that controlled redundancy is the only remaining item at the static constraint level. Controlled redundancy may be determined by the extend and value commands, which implies that the static constraint syntax can be defined by the syntax of the extend and value commands, as in

\[
\text{<assertion> ::= assert <type name> its <virtual attribute> <representation> = }
\]
\[
\text{<extension definition> | assert <virtual value> <representation> = <value definition>}
\]

According to the above example

\[
type \text{ invoice} \quad = \text{ customer, date, ...}
\]
\[
type \text{ invoice line} \quad = \text{ invoice, article, quantity, unit_price, ...}
\]

we obtain the constraints

\[
\text{assert invoice line its amount (R8,2) = quantity * unit_price.}
\]
\[
\text{assert invoice its invoice_amount (R8,2) = total invoice line its amount per invoice.}
\]
\[
\text{assert invoice its number of lines (1..*) = count invoice line per invoice.}
\]

6.5 Dynamic constraints

Dynamic constraints depend on the base operations insert, delete and update. Dependence on two or more base operations is excluded, because the constraint would have to be defined statically, resulting in undesirable overlap between categories. Let us therefore consider the base operations separately:

- insert constraints

Dynamic constraints exclusively depending on insert commands point to the existence of conditions active at the time of insert command execution. The conditions apply to the value of attributes involved in the insert command. The following situations can be distinguished:
initialization

This is generally done when initial attribute values are always the same, or when initial values of a large number of instances are the same. Initialization in the latter case is default in nature: the system only initializes if no value was specified. The initial value can be made dependent on the database state by the use of conditions (if then else, case). The attribute in question is always related to a base type.

Consider the definition:

type invoice = customer, date, ...

If the input date is generally required, we could initialize by:

default invoice its date = 'system date'

The syntax is simple:

default <type> its <attribute> = <property expression>

instantaneous relationship

Attribute value and database state at the time of insert command execution are related. Redundancy can be observed while the insert command executes, but not immediately after.

We can again let the initialization depend on the database state, and the attribute in question need not be related to a simple type.

An insert constraint example can be found in invoicing; consider a model with the type definitions:

type invoice = customer, date, ...
type invoice line = invoice, article, quantity, unit_price, ...
type article = price, ...

and initialized by:

init invoice line its unit_price = article its price.

The corresponding syntax is:
\textit{init <type> its <attribute> = <property expression>}.  

The article price is recorded at input time by above initialization. Should the article price change at a later stage, the prices in the invoice lines can be left unaltered, or if a different price is negotiated with the customer, the modification can be applied easily.

- \textit{delete constraints}

There are no dynamic constraints exclusively related to delete commands, because these constraints would refer to non-existent data - and it would obviously be senseless to apply a condition to "nothing". Apart from that, we can find a delete command blocked by an inherent or static constraint.

- \textit{update constraints}

Dynamic constraints exclusively dependent on update commands point to a relationship between the old and new attribute values involved in the update command.

This constraint is important if the number of state transitions is limited. New values can be made dependent on the existing ones in the database by conditions \textit{(case of)}. A simplified vehicle registration system serves as an example; assume a model defined by following types:

\begin{verbatim}
type vehicle = model, serial number, registration number, ...  
type transfer = vehicle, date, make, ...  
\end{verbatim}

Transfers are subject to limitations in practice. The first transfer is the supply of the vehicle to the garage and the second is the purchase of the vehicle by a customer. The vehicle can be exchanged at the garage after a few purchases, possibly followed either by a purchase by an other customer or scrapping of the vehicle. Valid state transitions are:

\begin{verbatim}
check transfer its make =  
  case of  
  supply    : purchase,  
  purchase   : purchase,  
  purchase   : exchange,  
  exchange   : purchase,  
  exchange   : scrap,  
  default    : supply.  
\end{verbatim}
The update constraint syntax is:

\[
\text{check } <\text{type}> \text{ } it\text{ } <\text{attribute}> = \\
\text{case of} \\
<\text{case list}>, \\
<\text{default element}>.
\]

We shall revert to these matters in connection with update constraints.

6.6 Transactions

Transactions are defined as logical work units, consisting of a series of one or more coherent simple operations transferring the database from one consistent state to another. Databases may temporarily be in a non-consistent state while transactions are being executed.

Transactions originate from database integrity requirements. The series of simple operations may be necessary because of non-adherence to e.g. inherent, static or dynamic constraints. Transactions start off by simple operations (so called trigger operations), followed by one or more additional simple operations (so called triggered operations). The trigger and triggered operations do not have to operate upon the same types and may consist of any type of simple operation (insert, delete and update).

Assume a trigger operation O1 on type T1, resulting in a triggered operation O2 on type T2. If we apply simple operation O1 to instance i of type T1, we will find that simple operation O2 is applied to instance j of type T2.

The question is: How is instance j selected? A relationship, important to the model, appears to exist between types T1 and T2, because of the trigger-triggered relationship. It seems appropriate to suppose that the two instances have been related by aggregation or generalization connections. If this is not the case, we have an undefined relationship in the structure, which should be viewed with caution.

We will now discuss transactions in the light of the trigger-triggered distinction, and we will differentiate between triggered insert, triggered delete and triggered update. We will limit ourselves to those operations requiring user actions.

Triggered inserts

This case is found when obligatory existential constraints have been applied to aggregation connections between two types. An example is the following model:
Various authors per article and various articles per author can be identified according to above definitions. It is clear that an article should at least have one author.

When articles are filed, at least the authorship should be recorded, which may imply that the author should be recorded as well. The trigger operation insert article consequently involves an insert authorship and possibly an insert author. The example demonstrates that trigger operations are not linked to fixed numbers of triggered operations.

There are cases in which one type is a specialization of another given type, with the constraint that every generalization instance be defined in one of its specializations. An example:

\[
\begin{align*}
type & \text{ vehicle} & = & \ldots, \text{ case} \\
type & \text{ car} & = & \{\text{vehicle}\}, \ldots \\
type & \text{ truck} & = & \{\text{vehicle}\}, \ldots
\end{align*}
\]

Car and truck are possible values for attribute case, because they are the types included in the type vehicle block.

Whenever a vehicle is added, a value should be allocated to the attribute case, which is required as a pointer to the condition controlling the triggered operation: for example, if the attribute value is car, a car insert must follow.

A third possibility is found with following type definitions:

\[
\begin{align*}
type & \text{ department} & = & \ldots, \text{ manager}\_\text{employee} \\
type & \text{ employee} & = & \text{name}, \ldots, \text{department}.
\end{align*}
\]

Insertion of a department must be followed by the insertion of an employee. This structure does not require triggered operations for other simple operations: for example, if a department manager is replaced, an employee insert is not necessary.

*Triggered deletes*

Triggered deletes are associated with trigger deletes and strongly depend on the above combination. In this case no data are required from the user after a trigger delete - which
is a potentially dangerous situation: the user can remove large parts of the database by a single command. The retention of relatability has far-reaching consequences in this case. It therefore appears as more practical to interrupt the chain reaction of triggered deletes and carry out the deletions separately. A simple example should clarify the situation.

Assume the conceptual model:

\[
\text{type department} = \ldots \\
\text{type employee} = \ldots, \text{department}.
\]

The removal of a department can start a chain reaction, for relatability reasons: all employees of the given department have to be removed as well.

The more practical solution is to conduct the transaction in two parts: removal of the appropriate employees followed by the removal of the department.

\textit{Triggered updates}

Inserts, deletes and updates can act as trigger operations for triggered updates. An important application of triggered updates is controlled redundancy, where the triggered update operates on a virtual value. Because the derivations are handled by the system, the user cannot intervene.

The same effect can be noted in our departments and employees example: it is unnecessary to adjust a virtual attribute like 'number of employees' when a new employee is added, because this is the responsibility of the database management system.

\subsection*{6.7 Design aspects}

Integrity rules should be set up in such a way that they can be executed at any time, although this leads to conditions for the constraints.

An investigation into 'deadlock' situations must be carried out first of all; these arise when for example init attributes are defined circularly, and virtual attributes depend on attributes of the same type. Examples are:

\begin{verbatim}
1 type a = x, y, z.
    init a its x = y.
    init a its y = x.
\end{verbatim}
2 \hspace{1em} \text{type} \ a = x, y, z. \\
\hspace{2em} \text{assert a its} \ v = x * y. \\
\hspace{2em} \text{init a its} \ x = v + z.

It should be obvious that above statements lock the initialization process completely. A similar circularity with only virtual attributes as in

3 \hspace{1em} \text{type} \ a = x, y, z. \\
\hspace{2em} \text{assert a its} \ v = w. \\
\hspace{2em} \text{assert a its} \ w = v.

is infeasible because a virtual attribute is defined simultaneously with its assertion - an undefined virtual attribute cannot be used in this way. However, when defining initializations, attributes already exist because of the type definitions. Of course, circular constructs like these must be avoided.

Virtual attributes and variables are not part of the conceptual model because they mean redundancy. As a consequence, conceptual models which are freed from assertions should be proper models, satisfying the two structural integrity rules of relatability and convertibility. Assertions play no role.

Designers must first build a proper model before proceeding to add redundancy by defining assertions. The addition of assertions may apparently disturb convertibility, which happens especially when existing type names are used, as with extensions.

6.8 Conclusion

Constraints are introduced for the improvement of data quality in the database. The level at which a constraint is specified is of the utmost importance. Let us illustrate this by two examples.

We will consider the specification of relatability first. Relatability is specified structurally in our approach: the inclusion of attributes in type definitions indicates their relatability to other types.

If we had, for example, specified the relatability between the types

\begin{verbatim}
    type invoice = customer, ...
    type customer = ...
\end{verbatim}

with static constraints, this would imply:
specification of the number of customer instances;

specification of the number of customer identifications;

equality of above numbers;

specification of the invariability of the customer identifications;

specification 'not null' for the customer identifications;

specification of the existence of references from the type invoice to the identification collection of the customer (subset invariant).

Inherent concepts are split up into various static level concepts here.

Assertions at the static constraint level specify controlled redundancy. Specifications of these constraints of course have consequences for the modification operations insert, delete and update. This can occur when controlled redundancy is specified by dynamic constraints. The assertion specification:

\[
\text{assert invoice } its \text{ invoice_amount (R8,2) =}
\]

\[
\text{total invoice line } its \text{ amount } per \text{ invoice.}
\]

has following consequences for the modifications:

- \text{insert invoice line:} \newline
  \text{invoice_amount: } = \text{invoice_amount } + \text{amount}.

- \text{delete invoice line:} \newline
  \text{invoice_amount: } = \text{invoice_amount } - \text{amount}.

- \text{update invoice line:} \newline
  \text{invoice_amount: } = \text{invoice_amount } - \text{amount(old)} + \text{amount(new)}.

It is apparent from this example that a specification is split up into various lower level ones, similar to the other example. There are no redundancies that cannot be identified by the database management system if concepts are specified at too low a level. It is consequently impossible for such a system to check specifications which originate from the same constraint for completeness, and therefore the designer and user are kept responsible. This is an undesirable situation as far as data management is concerned.

The above problems are non-existent if the approach described in this chapter is applied. The classification is such that categories do not overlap. In other words the constraint
specification language is orthogonal. This approach has many advantages for designers and users: constraints can only be defined in one manner by the constraint facilities. It is also preferable because constraint addition and deletion have no adverse side-effects.

Overlaps in constraint categories result in implicit constraints. These constraints can be handled by a database management system, which should have facilities capable of discovering these implicit constraints. Because every constraint can be regarded as (part of) a program, we run into insurmountable problems here - it is generally accepted that equivalence proofs between different programs cannot be given.

The approach described in this chapter avoids problems in consistency checking. Because the categories do not overlap, the approach contains no implicit constraints which can be input into the database management system. The consistency of the total system is guaranteed by the consistency of the individual categories. Consistency checking of the static and dynamic constraints has been vastly simplified by the introduction of virtual values and attributes. Static and dynamic constraints also influence the inherent constraints at the meta level (i.e. the data dictionary level).

6.9 Bibliography

Date [Date 81a] gives a detailed description of the consequences of primary and foreign keys in relational databases; primary keys extend the database administrator's responsibilities by four tasks and foreign keys by six. However, these activities do not guarantee database consistency, even if performed correctly.

Ample attention to a database specification language was paid by Brodie in his ACM/PCM approach, the language supporting facilities for explicit constraint specification. The IFIP case in [Brodie 82b] shows that these explicit constraints are almost exclusively used to support the concepts of relatibility, key and association. The size of the article (about 50 pages) is indicative for the consequences of explicit specification of inherent constraints.

Tsichritzis and Lochovsky also discuss constraint specifications in [Tsichritzis 77], pointing to the importance of inherent constraints. Enforcement algorithms are feasible if inherent constraints are coupled to carefully selected data manipulation operations, although the enforcement of explicit (static and dynamic) constraints is generally regarded as an extremely complex implementation problem.
7

EPILOGUE

7.1 Introduction

Having introduced a new semantic model in the previous chapters, we will now compare it to the current state of the computing discipline, and also to other developments. Section 7.2 shows that the closure property is relevant to mathematics as well, which emphasizes the essential role of object relativity in semantic models.

Semantic concepts have been well validated in the development of the Xplain DBMS, and in the production of data models for business applications. Section 7.3 contains three concise cases to illustrate the point, the third one demonstrating new potentialities of semantic concepts.

Section 7.4 summarizes the research into the area, and lists the achievements. In view of the results, a closer look at some of the research subjects is required; the relevant ones are indicated.

7.2 The closure property

Processes and data are fundamental objects in computer science, where programming (i.e. process description) has always attracted considerable attention. Data modeling, however, was established as a research subject only at a much later stage. This was due to two reasons: firstly, the available hardware was not suited to the handling of large amounts of data, and secondly, technical and scientific methods were not customary in this application area.

Programming developments have long been hampered by the absence of fundamental problem solving - just remember the times that flowcharts were used. Progress was only achieved after the introduction of basic concepts. A well-known example here is the invariance theorem for repetitive commands [Hoare 69].
Basic concepts are also vital to data modeling. Although the basic relational model has greatly contributed to rendering the approaches more fundamental, it lacks at least one fundamental characteristic: the closure property - so essential to many scientific disciplines, as we will demonstrate.

The algebraic property guaranteeing that the results of operations and its operands are of the same type is generally called the closure property [Date 86a]. The property applies to set algebra operations, because set operations (e.g. intersection, union or difference) bring about other sets.

We find the closure property in programming as well, where the relevant objects are the statements. For example: the union of statements may again be regarded as a statement. It has taken a long time before the closure property was positively defined within this young discipline - the 'go to statement' proved to be an obstacle for a long time.

The closure property appears to be part of the relational model because of the base which it shares with set theory [Date 86a]. The comparison with set theory, however, is not justified.

The (potentially repeated) execution of commands is not applied in programming when formulating the closure property. The only thing that matters is the process model. The use of instances (the individual tuples, for example) as relevant constructs for closures in data models is not allowed for similar reasons.

Relational models (i.e. basic and extended, see chapter 2) are based on the concepts of relation, attribute and domain. These concepts are relevant in a relational closure property. Because these interpretations cannot be generalized (attributes, for example, are not relations, and vice versa), closure is not a property of relational models.

Object relativity is the main principle in the Xplain model: the result of every aggregation, generalization or differentiation is also a type. So the closure property for the modeling algebra is inherent to this model. This implies that Xplain models are semantic, contrary to relational models.

7.3 Practical experience

The study, divided into a number of sub-phases, has been conducted as part of the Xplain project. The diversity in the nature of the practical applications is indicated by the following summary:

- Complex semantic models of housing grant registration are developed in [van der
Veer 83], the complexity being inherent to the nature of legislation. The master’s thesis suggests that the 1983 legislation is too intricate for adequate support by computerized systems.

- Semantic modeling is applied successfully to marine surveys in [te Kulve 85], where generalizations and specializations have worked in an extremely revealing way. An example will be given later.

- A classical database is described in [de Jong 85]: a personnel information system - no novelty in computing. The salient point of this study is the conversion of semantic models to classical ones.

- Two quality measurement and control situations are presented in [Giesberts 86] and [de Swart 86], including the use of the Xplain DBMS application generator.

- [Schravesande 87] studies data dictionaries. Semantic model design for a data dictionary system is explained, as well as practical problems of data description. Although one would expect the number of data elements to increase significantly by defining specializations, the contrary is true. Semantic models are capable of reducing the number of data elements (i.e. records and attributes) by 80%, which is confirmed by [Bes 88] in a semantic design for a municipal housing agency.

- [Mulder 87] applies semantic modeling to banking, especially inland funds transfer, and concludes that defining the relevant objects for outward processes (i.e. bank transactions) is a complicated matter.

- [van der Borg 88] designs a semantic model for the production of (language) dictionaries. The design is distinctly hampered by the incomplete semantics which linguists apply in defining linguistic concepts.

- Semantic models, applications and transactions for estate agent applications are defined in [Verveld 88].

- A simple semantic material description model, for an implementation with classical resources (i.e. programming languages and function libraries), is developed in [de Wolff 89]. The study shows that semantic models constitute a sound base for meaningful dialogue processing. The subject will be studied in more detail in the future, emphasizing the relationships between semantic models and organizational structures in general.

- Semantic principles are applied to a technical CAD/CAM environment in [van Rijswijk 90]. Although the familiar concepts are still valid, specific actions are required by the unusual nature of these applications. Similar problems appear in
many technical applications, and these will therefore be the subject of more detailed future studies. An example of this type of problems will be discussed later.

- Static constraints and calculated attributes are evaluated in [Zwaagstra 91]. These constraints are specified by an entity-relation-attribute model.
- Finally, [Mersel 91] discusses semantic models in the chemical process industry. Production processes are taken as starting points here, rather than information requirements.

Other researchers have also studied the practical application of semantic modeling: conceptual models for censuses [Barreto Vieira 85], international mailing tariffs [van de Craats 85], user interface management systems [Dur 89] and VLSI design [van den Hamer 90, van Leuken 88, van der Wolf 88].

The advantages of semantic concepts have proved especially useful in the application of specializations. Semantic concept boundaries therefore show up when specializations lead to practical problems, especially in extremely complex technical applications. The following cases show that, in addition to the advantages, certain drawbacks eventually emerge. We will also discuss a semantic remedy for the constraints.

**Case 1. Twined generalization hierarchies**

This example describes a cartography database covering North Sea data modeling as used in a survey office [ter Bekke 86].

Cartographic institutes are responsible for information concerning coasts and waterways, required by navigation. The information is, for example, made available by means of nautical maps indicating visible objects (drilling plants, lights, etc.).

The main objective of database design is the production of a database containing co-ordinate data for objects in specific sections of the continental shelf. The application of generalizations and specializations in data models will now be explained by making use of the obstructions and marks section of the database.

Navigation marks serve a variety of purposes; they indicate safe waterways and point to dangers, for example. Specializations are therefore unsuitable, as they result in identical type definitions. They are only functional when mark properties are included, and the following categories can consequently be distinguished:
beacon: anchored navigation mark;
buoy: floating navigation mark equipped with a light;
can: floating navigation mark.

These categories result in a generalization hierarchy with the following type definitions:

\[
\begin{align*}
\text{type} & \quad \text{mark} \quad = \text{name, color, form, purpose, position} \\
\text{type} & \quad \text{can} \quad = [\text{mark}], \text{sign, cardinality} \\
\text{type} & \quad \text{beacon} \quad = [\text{mark}], \text{type} \\
\text{type} & \quad \text{buoy} \quad = [\text{mark}], \text{sign, cardinality, light.}
\end{align*}
\]

The types can, and buoy have the common properties: mark, sign and cardinality. Buoy is therefore a specialization of can. This results in the abstraction hierarchy of figure 7.1.

![Diagram](image_url)

*Figure 7.1*

Figure 7.1 corresponds to the following type definitions:

\[
\begin{align*}
\text{type} & \quad \text{mark} \quad = \text{name, color, form, purpose, position} \\
\text{type} & \quad \text{can} \quad = [\text{mark}], \text{sign, cardinality} \\
\text{type} & \quad \text{beacon} \quad = [\text{mark}], \text{type} \\
\text{type} & \quad \text{buoy} \quad = [\text{can}], \text{light.}
\end{align*}
\]

While navigation mark data are prominent in charting databases, obstruction data are required as well. Obstructions generally imply dangers to navigation, such as rocks and wrecks. The latter are registered accurately; following categories are distinguished:

- charted wrecks, occurring on at least one sea chart;
• sleeping wrecks, having been omitted from sea charts because they do no longer constitute a danger to navigation;
• raised wrecks, having been removed physically from the waters.

Some charted wrecks are marked by buoys, and therefore an additional category is required. The generalization hierarchy in figure 7.2 now includes obstructions.

![Diagram](image-url)

**Figure 7.2**

The corresponding type definitions are:

- \textit{type} obstruction = registration\_date, position, source, type, depth
- \textit{type} wreck = [obstruction], name, nationality, lost\_date, type, length, width, height, capacity, cargo, material
- \textit{type} raised wreck = [wreck], raise\_date
- \textit{type} charted wreck = [wreck], charting\_date, symbol, chart\_scale
- \textit{type} buoyed wreck = [charted wreck], installation\_date, wreck\_buoy.

Marks and obstructions have now been modeled. A special feature of the property buoy should be noted: while it occurs as a buoyed wreck attribute in the above definitions, it is a type as far as marks are concerned (see figure 7.1). The two parts are therefore related by an aggregation as shown in the abstraction hierarchy in figure 7.3.
Case 2. Specialization versus aggregation

Databases in production environments are generally of an extremely simple structure. The complexity often resides in the applications because that is where integrity is controlled. The magnitude of the problem area can be imagined by just considering the subject of bill of materials. Only two types are relevant here: the first (product type) describes all products from part to end product, and the second defines the relationships between products (type relationship). For example (see [Codd 70]), only one aggregation level (see figure 7.4) can be defined for

\[
\begin{align*}
\text{type product} & = \text{description, make, type, price, stock, assembly time} \\
\text{type relationship} & = \text{compound_product, component_product, number.}
\end{align*}
\]
All products, i.e. parts, intermediary and end products, are described in product, and the product structure can therefore only be defined with difficulty. In addition, only common attributes of these categories are included, leading to incomplete product descriptions. Furthermore, the combination of products is cumbersome with these models.

Product databases gain in transparency by structuring. The conceptual model should correspond to modular product definitions, as amplified by the following example about bicycle product structure.

There are many types of bicycles: standard and 'de luxe' ones, sports bicycles and the ones for children. But they all have properties in common: saddle, frame, handlebar, wheel, propulsion and brake, for which a generalization can be defined. The base bicycle is an aggregation with following type definitions:

- \textit{type} base bicycle \quad = \text{saddle, frame, handlebar, front\_wheel, rear\_wheel, propulsion}
- \textit{type} frame \quad = \text{frame\_tube, saddle\_tube, steering\_tube, front\_fork, size}
- \textit{type} propulsion \quad = \text{chain, chain\_cover, pedals.}

The other attributes may be considered as specializations of the type part, and we consequently also find a set of type definitions, of which the following are just a few (see figure 7.5):

- \textit{type} part \quad = \text{description, make, type, price, stock, assembly\_time}
- \textit{type} tube \quad = \text{[part], length, diameter, material}
- \textit{type} front\_fork \quad = \text{[part], height, diameter, thickness, material.}

Base bicycle specializations might be:

- \textit{type} childs\_bicycle \quad = \text{[base\_bicycle], side\_wheels, lock, bell}
- \textit{type} de\_luxe\_bicycle \quad = \text{[base\_bicycle], mudguard, stand, lock, pump, gear, lighting, bell}
- \textit{type} standard\_bicycle \quad = \text{[base\_bicycle], mudguard, luggage\_carrier, lighting, lock, bell}
- \textit{type} sports\_bicycle \quad = \text{[base\_bicycle], gear}
- \textit{type} racing\_bicycle \quad = \text{[sports\_bicycle], sprocket\_gear}
- \textit{type} touring\_bicycle \quad = \text{[sports\_bicycle], pump, lighting, bell.}

Structural restrictions have been included in the above model, but we also need static and dynamic restrictions, such as:
Figure 7.5

- Base bicycle wheel sizes should correspond to frame sizes:
  
  ```plaintext
  assert base bicycle its acceptable (true) =
  frame its size = front_wheel its size and
  frame its size = rear_wheel its size.
  ```

- Racing bicycles have either racing handlebars or triathlon handlebars:
assert racing bicycle its type handlebar ('racing', 'triathlon') =
sports bicycle its base bicycle its handlebar its type.

- De luxe bicycles are equipped with ASSA locks:
  default de luxe bicycle its lock = 'ASSA'.

Products are described in detail by this structure; the main characteristic of semantic models is the clear resemblance to modular product structure. It is therefore much easier to introduce new bicycle models based on one of the existing ones.

**Case 3. The meta model solution**

So far we have seen cases where the introduction of specializations resulted in better understanding of the problems. And, because only a few dozen specializations were required, the models run well in practice. If, on the other hand, many specializations are required, say, a couple of thousands, we have a problem, because databases cannot be controlled adequately under these conditions. The definition of models and applications is extremely cumbersome, and alternatives must be found. An indication was given in case 2, by limiting the number of data structures to two. Because data and meta data have been integrated, we can now take a more effective route than the one described in figure 7.4. Type descriptions remain intact, and data integrity is consequently guaranteed. As these problems only occur with specializations in practice, the following example is based on these.

The relationship between an instance and the type to which the instance belongs, can be represented in an abstraction hierarchy (figure 7.6). Type is considered as a dictionary containing definitions of all product types within a given range.

```
instance

  type
```

*Figure 7.6*

Suppose we wished to design a conceptual model of a product database containing many (say, thousands) product type data. In addition to certain fixed attributes we have product dependent ones, requiring the introduction of thousands of specializations. We cannot
apply the bicycle database method here, because the definition of the model and its applications would be extremely laborious.

We would rather apply a two level product description: attribute values (data) are allocated to product instances at the first level, and the interpretation of these values (meta data) is given at the second. The relationship between attributes and types is defined by aggregates, as shown in figure 7.7.

![Diagram](figure_7.7)

The relationship between attributes and instances can be characterized by the following statements:

- each instance corresponds to various attribute values, and each attribute value belongs to only one instance;
- each attribute corresponds to various attribute values, and each attribute value belongs to only one attribute.

The relationship between attributes and instances is shown in figure 7.8.

![Diagram](figure_7.8)
Combining the abstraction hierarchies in figures 7.6, 7.7 and 7.8, we end up with a meta model containing data integrated with meta data (figure 7.9).

![Diagram](image)

Figure 7.9

The corresponding type definitions are:

- `type type` = product name, domain
- `type instance` = composite_type, serial number
- `type attribute` = composite_type, type, role, kind
- `type attribute value` = attribute, instance, value.

Type and attribute value are linked in several ways, so that following static restriction is essential:

```
assert attribute value its validity (true) =
  attribute its composite_type = instance its composite_type.
```

7.4 Supporting software

The availability of supporting computerized facilities is essential for the development and testing of new methods in computer science. Because of the nature of our research, it was clear from the outset that we required a system including the capability of handling the semantic issues. The following research is related to the Xplain DBMS:
• [Goossens 83] summarizes the Xplain DBMS architecture by describing three modules: definition, editing and manipulation. These modules process the commands supplied to the DBMS by referring to the filing system and (meta) data in the active data dictionary.

• [Pasma 83] describes the Xplain DBMS filing system in more detail. Highly efficient basic functions required by the particular filing system based on transposed files are defined, providing a certain degree of independence of the operating system.

• [Cheung 84] discusses the Xplain DBMS data manipulation section. Realistic performance measurements confirm expectations that semantic DBMSs are considerably more efficient than relational ones (as also indicated in [ter Bekke 76]). In addition, users are now in a position to gain ample experience in the data manipulation language.

• The integrity control inherent to semantic models requires additional support from special B+-tree functions, described and measured extensively in [van Vuren 84].

• Query languages enjoyed much attention in the early phases of the Xplain project, but the interest shifted to application generators subsequently. [Bron 85] introduces an application generator for the Xplain DBMS, completely geared to the nature of semantic models. This is evident because no software is generated for the individual applications, but only semantic data structures which are provided with structural application data. Testing of applications is essentially no longer required.

• Publications on the subject of the optimistic concurrency control method have evaluated the applicability of the Xplain DBMS [Huijgens 85]. Although the method can be used, serious problems occur with B+-trees.

• [van Bruchem 87] extensively discusses static and dynamic restrictions. The study has decisively influenced the nature of these restrictions in the Xplain DBMS (see chapter 6). The definition of restrictions is tightly linked to enforcement and consistence control. Its complexity is often underrated in practice, which often follows from the careless adherence to integrity rules (for example, referential integrity in relational systems).

• Users should have controlled access authority to data in practice, because databases are generally accessed by a multitude of users. [Barbier 87] defines the Xplain DBMS mechanism which decides whether an individual user is authorized to operate (define, insert, delete, update, get) on data (database, type, attribute).
Experience shows that the required functionality of current DBMS architectures leads to extremely voluminous software systems. Various new architectures have been proposed to comply with future functional requirements, which are expected to grow in complexity. [Janson 88] investigates whether self-descriptive DBMS architectures can be applied to semantic databases, and finds that it is as yet impossible due to the complexity of the matter.

DBMSs generally provide facilities for easy report generation. [Stolk 88] indicates the definition and implementation of an Xplain DBMS report generator.

Not only inherent restrictions are essential to integrity control - static and dynamic ones are required as well. [Kruijf 88] designs and implements the domain specification section (patterns, trajectories and enumerations).

Graphics workstations, of increasing importance lately, have greatly influenced the handling of databases. Abstraction hierarchies are essential to the semantic Xplain model. [Goossens 90] enables us to define semantic models graphically, and textual definitions are thus superfluous. Semantic integrity control is indispensable here.

Client/server architectures are attractive in database environments, because new applications such as distributed operation and graphics are now available. [Lasschuyt 90] presents a client/server architecture, the server of which provides graphics data definition and manipulation capabilities in Xplain environments.

Abstraction hierarchies provide excellent starting points for data manipulation, and [Polderman 90] designs and implements the required software. Semantic databases can thus be accessed without the requirement for textual manipulation languages.

The problem of dynamic restrictions in the Xplain DBMS has been solved in [Wijntjes 90] after reconsidering the complete data definition implementation. The modified data definition enables simple implementations of static restrictions.

Client/servers should be available in concurrent and distributed environments; [van Reeuwijk 91] provides the facilities.

The current Xplain software system [ter Bekke 89] consists of the following components:

- **Xplain nucleus** comprising software handling the physical storage of database contents. The modern file management techniques which have been applied (e.g. transposed files), contribute to achieving outstanding performance. Special functions guarantee the independence from operating systems, terminals and printers.
Main program
responsible for the dialogues, while facilities are available for the efficient adaptation of system messages and user commands.

DDL processor
handling commands for the definition and modification of the conceptual model.

DML processor
handling commands for data retrieval, update, addition and deletion.

B-tree management
consisting of special software for efficient data retrieval, even from large databases. B+-tree implementation in addition provides the maintenance of data integrity.

Application generator
enabling the user to define customized screens (views), whereby one is not confined to simple tables, but where even hierarchies can be defined easily. Defined screens can be used for all types of data manipulation and retrieval, allowing users unfamiliar with semantic manipulation languages to turn the power of the system to advantage.

Report generator
offering facilities for the production of user defined reports, in a manner comparable to the application generator, but differing by the fact that the report generator acts on previously defined retrievals.

Authorization manager
recognizing database users with individual profiles for operations (insert, delete, update, retrieve) on types and attributes of certain databases.

Xplain server applications
Windowing and graphics developments have recently been implemented. Data definition (and limited data manipulation) can currently be carried out using a graphics screen displaying abstraction hierarchies. DDL and DML communication with the original system is based on a client/server architecture.

7.5 Conclusion

Initially the author's research into databases centered around non-procedural data manipulation languages [ter Bekke 76], prompted by the unavailability of adequate query
languages for relational databases. Another reason was the keen international interest in the subject (see [Chamberlin 76]). The study identified quite some shortcomings in the basic relational model, and, also prompted by international developments ([Smith 77a, Smith 77b]), undertook to research the entire area of data modeling. As additional shortcomings came to light, the study objectives had to be widened. The main question was:

Can semantic data models be defined consistently, and can their appropriateness be determined?

The ensuing study initially proposed a consistent data manipulation language ([Iter Bekke 76]), but a complete data language emerged after reviews and extensions (for reports see [Iter Bekke 83, Iter Bekke 88b]).

Experiences with the new concepts were gained in two ways:

First of all, the views were applied extensively to organizations, where the inadequacies inherent to the relational models did not show up. The concepts provide clear insight, even allowing adequate descriptions of complex situations.

Secondly, the ideas were implemented in the Xplain DBMS. Semantic modeling concepts were also used in the design of the software system. Semantic principles have greatly influenced the creation of the Xplain DBMS, involving the extensive application of semantic concepts as found in practice. The advantage was that the DBMS could be used to test the practical results, and it also allowed performance measurements which were, as expected, favourable ([Iter Bekke 76, Iter Bekke 85]).

Above applications and implementations essentially confirmed that the study objectives had been met. In order to widen the applicability of semantic concepts, we investigated whether the results of semantic designs can also be used when semantic implementations like the Xplain DBMS are unavailable.

The answer to the question was simple, as it turned out. Semantic solutions are easily converted to relational ones by simple procedures. These relational implementations have been compared in tests to others, their performances not being significantly less favourable. Relational DBMSs do not support all structural components of semantic solutions as effectively as semantic DBMSs, and additional procedures are therefore required. This is, however, not due to the semantic solutions, but inherent to relational DBMSs.

The study has suggested a number of subjects worth additional research:

Firstly, it would be useful to investigate the role of semantic views in distributed
processes. Horizontal and vertical fragmentation play a major part in relational environments (see also [Date 90]). Aggregations and generalizations, on the other hand, are applied under semantic conditions. It would be useful to investigate whether other semantic concepts are required in distributed environments, in addition to the above ones.

Secondly, the role of abstraction hierarchies in the development and application of semantic databases should be studied. The current Xplain DBMS tests whether the required relationships between application data can be deduced from abstraction hierarchies. Similar tests are additionally carried out during data manipulation, and abstraction hierarchies therefore already play a minor part in database application development with the current DBMS. It is worth investigating whether the role of abstraction hierarchies can be extended, whereby graphics user interfaces would be indispensable.

Finally, our study has shown that abstraction hierarchies are essential to structuring and processing of activities involved in the use of databases - and the subject is just waiting for additional developments, such as the interaction between abstraction hierarchies and the arrangement of activities within organizations in general.
REFERENCES


REFERENCES


[Chen 86] P.P.-S. Chen, Presentation for the NGI Database Club (may 1986).


[Codd 79] E.F. Codd, Extending the database relational model to capture more meaning, ACM TODS 4 (1979), 397-434.


[Date 81b] C.J. Date, Referential integrity, Proc. 7th Int. Conf. Very Large Data Bases (1981), 2-12.

[Date 85] C.J. Date, A critique of the SQL database language, ACM SIGMOD Record (1985), 8-52.

[Date 86b] C.J. Date, Relational Database: Selected Writings, Addison-Wesley (1986).


[Goossens 89] M.C.S. Goossens, A graphical DDL interface for the Xplain DBMS, Master’s Thesis Delft University of Technology, Faculty of Technical Mathematics and Informatics (1990), (in Dutch).


REFERENCES


[Lasschuyt 90] E. Lasschuyt, A server for the semantic DBMS Xplain, Master’s Thesis Delft University of Technology, Faculty of Technical Mathematics and Informatics (1990), (in Dutch).


[Polderman 90] H.E. Polderman, A graphical user interface for the Xplain DBMS, Master’s Thesis Delft University of Technology, Faculty of Technical Mathematics and Informatics (1990), (in Dutch).


REFERENCES


REFERENCES


[van Bruchem 87] M. van Bruchem, Controlled redundancy in the Xplain DBMS, Master’s Thesis Delft University of Technology, Faculty of Technical Mathematics and Informatics (1987), (in Dutch).


INDEX

abstraction ........................................... 12, 28
abstraction hierarchy .................................. 52
aggregation ............................................... 12
algebra ..................................................... 3
ANSI ....................................................... 4
assert command .......................................... 93
assertion ................................................... 29, 43
associative entity ......................................... 19
attribute ................................................... 8
axiom of extensionality .................................. 28, 38
block ........................................................ 52
cartesian product ......................................... 4
characteristic entity ....................................... 19
class ........................................................ 38
closure property .......................................... 103
conceptual level .......................................... 5
conceptual model ......................................... 5, 41
connection trap .......................................... 80
consistent database ....................................... 6
constraint .................................................... 87
convertibility ............................................ 44, 48
data dictionary ........................................... 116
data model ................................................... 6
data independence ........................................ 3, 5
database ...................................................... 6
database management system ......................... 6
datum ........................................................ 42
DBMS ....................................................... 6
definition language ....................................... 3
delete command ......................................... 76
denotation ................................................... 35
difference .................................................. 4, 33
differentiation ............................................ 35
discriminated union ...................................... 14
dynamic constraint ....................................... 88, 93
E-R diagram ............................................... 8
E relation .................................................... 19
empty set ................................................... 37
enterprise schema ........................................ 7
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>8</td>
</tr>
<tr>
<td>entity-relationship approach</td>
<td>7</td>
</tr>
<tr>
<td>explicit constraint</td>
<td>88</td>
</tr>
<tr>
<td>extend command</td>
<td>70</td>
</tr>
<tr>
<td>extended relational model</td>
<td>18</td>
</tr>
<tr>
<td>extension</td>
<td>35</td>
</tr>
<tr>
<td>external level</td>
<td>5</td>
</tr>
<tr>
<td>external model</td>
<td>5</td>
</tr>
<tr>
<td>file organization</td>
<td>2</td>
</tr>
<tr>
<td>generalization</td>
<td>12, 35</td>
</tr>
<tr>
<td>get command</td>
<td>67</td>
</tr>
<tr>
<td>hereditary set</td>
<td>28</td>
</tr>
<tr>
<td>implicit constraint</td>
<td>88</td>
</tr>
<tr>
<td>information</td>
<td>42</td>
</tr>
<tr>
<td>inherent constraint</td>
<td>88, 89</td>
</tr>
<tr>
<td>inheritance link</td>
<td>23</td>
</tr>
<tr>
<td>insert command</td>
<td>75</td>
</tr>
<tr>
<td>insert constraint</td>
<td>93</td>
</tr>
<tr>
<td>instantiation</td>
<td>24</td>
</tr>
<tr>
<td>intension</td>
<td>28</td>
</tr>
<tr>
<td>internal level</td>
<td>4, 5</td>
</tr>
<tr>
<td>intersection</td>
<td>9, 33</td>
</tr>
<tr>
<td>invariant</td>
<td>27</td>
</tr>
<tr>
<td>invariant set property</td>
<td>32</td>
</tr>
<tr>
<td>invariant type property</td>
<td>34</td>
</tr>
<tr>
<td>is a abstraction</td>
<td>53</td>
</tr>
<tr>
<td>is a link</td>
<td>23</td>
</tr>
<tr>
<td>is part of abstraction</td>
<td>53</td>
</tr>
<tr>
<td>its</td>
<td>49</td>
</tr>
<tr>
<td>kernel entity</td>
<td>19</td>
</tr>
<tr>
<td>logical level</td>
<td>4</td>
</tr>
<tr>
<td>manipulation language</td>
<td>3</td>
</tr>
<tr>
<td>meaning</td>
<td>41</td>
</tr>
<tr>
<td>meta data</td>
<td>116</td>
</tr>
<tr>
<td>modification</td>
<td>75</td>
</tr>
<tr>
<td>naive set theory</td>
<td>38</td>
</tr>
<tr>
<td>object identity</td>
<td>43</td>
</tr>
<tr>
<td>object relativity</td>
<td>50, 54</td>
</tr>
<tr>
<td>orthogonality</td>
<td>77</td>
</tr>
<tr>
<td>P relation</td>
<td>19</td>
</tr>
<tr>
<td>per</td>
<td>70</td>
</tr>
<tr>
<td>predicate</td>
<td>29</td>
</tr>
<tr>
<td>primary key</td>
<td>13, 19</td>
</tr>
<tr>
<td>Term</td>
<td>Page(s)</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>projection</td>
<td>4</td>
</tr>
<tr>
<td>proposition</td>
<td>44</td>
</tr>
<tr>
<td>query conversion</td>
<td>83</td>
</tr>
<tr>
<td>record</td>
<td>3</td>
</tr>
<tr>
<td>referential integrity</td>
<td>48, 49</td>
</tr>
<tr>
<td>relatability</td>
<td>48</td>
</tr>
<tr>
<td>relation</td>
<td>3</td>
</tr>
<tr>
<td>relational algebra</td>
<td>4</td>
</tr>
<tr>
<td>relationship</td>
<td>8</td>
</tr>
<tr>
<td>role attribute</td>
<td>67</td>
</tr>
<tr>
<td>RM/T</td>
<td>18</td>
</tr>
<tr>
<td>selection</td>
<td>4</td>
</tr>
<tr>
<td>selection command</td>
<td>66</td>
</tr>
<tr>
<td>semantic hierarchy model</td>
<td>12</td>
</tr>
<tr>
<td>semantic modeling</td>
<td>6</td>
</tr>
<tr>
<td>semantic networks</td>
<td>23</td>
</tr>
<tr>
<td>semantics</td>
<td>4, 41, 42</td>
</tr>
<tr>
<td>set</td>
<td>28, 32, 38</td>
</tr>
<tr>
<td>set algebra</td>
<td>33</td>
</tr>
<tr>
<td>set function</td>
<td>69</td>
</tr>
<tr>
<td>specialization</td>
<td>35</td>
</tr>
<tr>
<td>SQL</td>
<td>63, 82</td>
</tr>
<tr>
<td>static constraint</td>
<td>88, 91</td>
</tr>
<tr>
<td>subject</td>
<td>29</td>
</tr>
<tr>
<td>subtype</td>
<td>22</td>
</tr>
<tr>
<td>supertype</td>
<td>22</td>
</tr>
<tr>
<td>surrogate</td>
<td>19</td>
</tr>
<tr>
<td>transaction</td>
<td>96</td>
</tr>
<tr>
<td>trigger operation</td>
<td>96</td>
</tr>
<tr>
<td>triggered operation</td>
<td>96</td>
</tr>
<tr>
<td>type</td>
<td>33, 34</td>
</tr>
<tr>
<td>type algebra</td>
<td>35</td>
</tr>
<tr>
<td>union</td>
<td>4, 9, 33</td>
</tr>
<tr>
<td>universal set</td>
<td>37</td>
</tr>
<tr>
<td>update command</td>
<td>76</td>
</tr>
<tr>
<td>update constraint</td>
<td>97</td>
</tr>
<tr>
<td>value command</td>
<td>71</td>
</tr>
<tr>
<td>value range</td>
<td>90</td>
</tr>
<tr>
<td>Venn diagram</td>
<td>33</td>
</tr>
<tr>
<td>view independence</td>
<td>5</td>
</tr>
<tr>
<td>Xplain</td>
<td>66</td>
</tr>
</tbody>
</table>


SUMMARY

The application of computers during the past decades has strongly dominated the way we think about data processing. Data and their meanings were quite separate items originally. Only raw data were submitted for processing, while the interpretation was left to the user. The software was completely ignorant of the real meaning of the data - think of mathematical analysis software.

As the usage of computers intensified, a need was felt to record (part of) the meaning together with the data. In the initial period, this was done by the programs used for the processing of these data. It led to redundant data descriptions and inconsistencies and as a consequence it became clear that data and their interpretations needed to be regarded as an integrated unit - the database concept was born. Data were no longer regarded as meaningless bits but as significant facts about the real world.

Data correspond to a varying real world and are therefore generally not static. Models for the description of data (i.e. data models) must consequently allow dynamics, in addition to the capability of recording meanings. In this context, we often interpret dynamics as the capability of modifying existing data collections, e.g. by allowing extendable populations.

In addition, data models must be able to handle data interpretations in a flexible manner. This indicates a much more significant dynamical data aspect. Meta concepts such as entity, relationship, generalization, specialization and attribute should be exchangeable, depending on the required application.

All known data models are capable of handling variable data collections. While this flexibility is self-evident, matters are different for interpretation flexibility. The relational model would appear to be the first model with this property, because it is based on only one data structure: the (mathematical) relation. A closer look, however, reveals that the model is associated with various meta concepts incapable of exchange, because the model is familiar with the non-exchangeable concepts of attribute and domain, in addition to relation.

Data models satisfying the requirement for flexible interpretations are called semantic. Due attention has been paid to theory, but in addition ample thought has been given to data modeling aspects, while the applicability has been extensively verified in practice. The emphasis in this dissertation lies with the objectives and principles of semantic data modeling. Examples show that the conceptual (semantic) approach may also be applied successfully in other (e.g. relational) environments.
Chapter 1 describes the developments in the profession during the past decades, including relational models, because they influenced matters decisively. New data models evolved from the use of the three schema architecture, reviewed in chapter 2. Although these data models were occasionally improvements to the original relational data models, none of the proposals comply with the three schema architecture.

Chapter 3 sets the scene for the proposed semantic data model. After explaining the meaning of abstractions, two base principles are introduced: set and type. Whereas sets correspond to the familiar mathematical notion, types are at the base of semantic data models. The analogy with the set concept is amazing: while sets are applied to extensions (i.e. the objects proper), types are used for intensions (i.e. the object properties) in a similar way.

Chapter 4 contains a practical overview of the components of data modeling, confirming that flexibility requirements are met by the approach. This is especially true for the object relativity principle.

Data models consist of structures, operations and restrictions. Chapters 5 and 6 introduce operations and restrictions respectively, which have been tailored to structuring concepts and result in unambiguous solutions, as shown by a number of examples in the text. Semantic solutions therefore provide a sound basis for relational environment solutions. Examples illustrate the point.

In chapter 7, the model is compared to developments within the profession and other areas. In addition to a broad overview of applications used to test the semantic concepts, it contains a summary of the developed software. Reference is also made to subjects meriting further research.

This dissertation introduces a semantic data model complying with flexibility requirements, and the advantages of this data model have been repeatedly demonstrated in practice. The data model has also been implemented, which implies that supporting software can be developed, allowing effective control over the additional knowledge we have about the real world. The application of semantic data modeling methods results in unambiguous solutions, which form a sound base for solutions following other approaches. Relational environment applications are shown, the familiar anomalies normally found in relational models being absent.
SAMENVATTING

Het gebruik van computers heeft het denken over gegevensverwerking de afgelopen decennia sterk beïnvloed. In eerste instantie leidde het gebruik van computers tot een scheiding tussen gegevens en hun betekenis. Alleen ruwe gegevens werden ter verwerking aangeboden, de interpretatie werd overgelaten aan de gebruiker. De programmatuur had geen enkele notie van de betekenis van de gegevens. Voorbeelden hiervan zijn te vinden in programmatuur voor wiskundige analyses.

Naarmate het gebruik van computers toename, nam ook de noodzaak toe om (een deel van) de betekenis samen met de gegevens vast te leggen. In het begin gebeurde dit in de programma's die voor de verwerking van de gegevens nodig waren. Deze aanpak leidde tot redundante gegevensbeschrijvingen met als gevolg inconsistenties. Het werd duidelijk dat gegevens samen met hun interpretatie als één geheel moesten worden beschouwd; het database concept was ontstaan. Gegevens werden niet langer beschouwd worden als betekenisloze bits, maar als betekenisvolle feiten over de werkelijkheid.

Gegevens corresponderen met een werkelijkheid die aan veranderingen onderhevig is; ze zijn in het algemeen niet statisch. Hieruit volgt dat een model voor de beschrijving van gegevens (d.w.z. een datamodel) naast de mogelijkheid om betekenen vast te leggen, dynamiek moet toestaan. Bij dynamiek wordt in deze context vaak gedacht aan de mogelijkheid om de bestaande gegevensverzamelingen te veranderen (door bijvoorbeeld uitbreidbare populaties toe te staan).

Naast het bovenstaande moet een datamodel flexibel omgaan met de interpretaties van gegevens. Hiermee wordt een veel belangrijker dynamisch aspect van de gegevens aangegeven. Metabegrippen als entiteit, relatie, generalisatie, specialisatie, attribuut etc. zijn dan, afhankelijk van de gewenste toepassing, uitwisselbaar.

Alle bekende datamodellen bezitten de mogelijkheid om veranderlijke gegevensverzamelingen te beschouwen. Deze flexibiliteit is dus vanzelfsprekend. Met de flexibiliteit van interpretatie is het anders gesteld. Omdat het relationele model gebaseerd is op slechts één gegevensstructuur, nl. de (wiskundige) relatie, lijkt dit model deze eigenschap te bezitten. Een nadere beschouwing leert echter dat dit model diverse niet-uitwisselbare metabegrippen kent, immers naast relatie kent dit model de niet-uitwisselbare begrippen attribuut en domein.

Een datamodel dat flexibele interpretatie ondersteunt, wordt semantisch genoemd. In dit proefschrift wordt de nadruk gelegd op de doelstellingen en fundamente van semantische datamodellering. Dat men ook in een andere (bijv. relationele) omgeving voordeel kan hebben van een semantische benadering wordt door middel van voorbeelden aangegeven.

Hoofdstuk 3 legt de basis voor het ontwikkelde semantisch datamodel. Na een uiteenzetting over het begrip abstractie worden twee basisvormen geïntroduceerd: verzameling en type. Het begrip verzameling komt overeen met het begrip uit de wiskunde, het begrip type legt de basis voor het semantisch datamodel. De analogie met het begrip verzameling is frappant. Terwijl verzamelingen worden toegepast op extensies (d.w.z. de objecten zelf), worden typen begrip op soortgelijke manier toegepast op intensies (d.w.z. de objecteigenschappen).

Hoofdstuk 4 bevat een praktische uiteenzetting van de begrippen voor datamodellering. Hieruit blijkt dat de voorgestelde benadering voldoet aan de flexibiliteitseisen. Dit komt vooral tot uitdrukking in het objectrelativiteitsprincipe.


In hoofdstuk 7 wordt het model gerelateerd aan ontwikkelingen binnen en buiten het vakgebied. Er wordt een globaal overzicht gegeven van de toepassingen waarmee de semantische concepten zijn getoetst. Het hoofdstuk bevat ook een overzicht van de ontwikkelde software. Naast de bereikte resultaten worden enkele onderwerpen voor nader onderzoek aangegeven.

In het proefschrift is een semantisch datamodel geïntroduceerd dat voldoet aan de flexibiliteitseisen. Daarbij is de praktische bruikbaarheid van dit datamodel in diverse praktijk-situaties aangetoond. Het datamodel is tevens geïmplementeerd: het is dus mogelijk om ondersteunende software te ontwikkelen die de extra kennis van de werkelijkheid effectief kan behoren. Toepassing van de semantische modelleringsbegrippen resulteert in eenduidige oplossingen, die een goede basis vormen voor oplossingen volgens andere benaderingen. Toepassingen in relationele omgevingen zijn gegeven. De anomalieën die verbonden zijn aan het relationele model blijken hierbij niet op te treden.
CURRICULUM VITAE