Data and Knowledge Engineering, Vol. 9, No. 2,
pp. 157-204, North Holland, Amsterdam (1992)

Conceptual Modelling of Database Applications Using an Extended ER Model

Gregor Engels#, Martin Gogolla*, Uwe Hohenstein+, Klaus Hülsmann*,1,
Perdita Löhr–Richter*, Gunter Saake*, Hans–Dieter Ehrich*

* Technische Universität Braunschweig
Informatik, Abt. Datenbanken,
Postfach 3329
W - 3300 Braunschweig, Germany

\# Dept of Computer Science, Leiden University
P.O Box 9512
2300 RA Leiden The Netherlands

\+ Siemens AG, ZFE BT SE 33
Otto–Hahn–Ring 6
W - 8000 München 83, Germany

ABSTRACT

In this paper, we motivate and present a data model for conceptual design of structural and behavioural aspects of databases. We follow an object centered design paradigm in the spirit of semantic data models. The specification of structural aspects is divided into modelling of object structures and modelling of data types used for describing object properties. The specification of object structures is based on an Extended Entity–Relationship (EER) model. The specification of behavioural aspects is divided into the modelling of admissible database state evolutions by means of temporal integrity constraints and the formulation of database (trans)actions. The central link for integrating these design components is a descriptive logic–based query language for the EER model. The logic part of this language is the basis for static constraints and descriptive action specifications by means of pre- and postconditions. A temporal extension of this logic is the specification language for temporal integrity constraints. We emphasize that the various aspects of a database application are specified using several appropriate, but yet compatible formalisms, which are integrated by a unifying common semantic.

CR Categories and Subject Descriptors:

[1] K. Hülsmann's work is supported by Deutsche Forschungsgemeinschaft under grant (En184/1)

H.2.1 [Database Management] Logical Design – Data models;
Schema and subschema.
H.2.3 [Database Management] Languages – Data description languages (DDL);
Data manipulation languages (DML);
Query languages.
D.2.1 [Software Engineering] Requirements/Specification – Languages.
D.2.10 [Software Engineering] Design – Methodologies.

General terms: Design, Languages, Reliability.

Additional Key Words and Phrases: Conceptual data model, conceptual database design, Entity–Relationship model, database evolutions, integrity constraints, query language, transactions.

Contents

1 Introduction

Since the early times of database systems the scenario of database applications has dramatically changed. New applications like CAD, CASE [1, 2, 3, 4], office information systems [5] or geoscientific databases [6, 7] need more sophisticated database functionalities. These applications typically require the administration of complex structured objects and need a larger variety of data types such as geometric and temporal data types or data types for storing visual or acoustic data.

Another point is that databases are increasingly used for integrating *different* applications. For example a common database for air traffic applications could be accessed by such different applications as flight scheduling, air traffic control, flight booking in travel agencies or the maintenance of airplanes. Since such a database has to reflect more facets of the whole application area than databases for the single applications, integration additionally not only increases the size complexity of the database but also the structural complexity of single entities in the database.

There has been a change from application specific databases to domain specific ones. These changes have also caused a change in database design philosophy from a merely application progam centered way to an *object centered* manner of database modelling. The task of the database designer is not chiefly the modelling of databases specific for dedicated applications but to reflect the structures of some real world part by a database used by a variety of application programs. Consequently, the primary task of the database designer is to model the properties of the real world objects in the domain of discourse and not the functionality of the various application programs.

This attitude to database modelling has been accepted long before so–called object–oriented data models became fashionable in the database research community. The main idea is that the relevant real world objects are represented by corresponding objects in the database. These objects are abstract entities with a fixed identification which can be inserted into or removed from a database and which are often composed of other objects (complex objects). They have properties (or attributes) which contain values of corresponding data types and which can be used for an external, observable representation of objects. Data types themselves may have simple or complex structures. For instance, a geographical object can be described by as simple structured properties as its name over the data type **string** or by as complex structured ones as its border represented by a polygon. Usually, such complex structures can also be expressed as complex objects. Nevertheless, we think the distinction between objects and data is essential, because not every complex structure also meets the intuitive concept of an object.

The object centered way of designing the structure of databases had also consequences for the design of database application programs or, more generally, database dynamics. Many operations on a database are not specific for some particular program rather than for the modelled objects and data structures. Consequently, these operations should be designed together with the object and data types. Another aspect is that implicitly all possible evolutions of a database contents are fixed by the design of the static structure of a database. But, as some of these evolutions do not reflect possible evolutions of the real world, this set has to be restricted to the set of admissible database evolutions. This is done in form of so–called temporal integrity constraints. Indeed, this could also be done by integrating these restrictions directly into database operation specifications. But a separate specification of these two aspects of database behaviour allows operation design to concentrate on the pure functionality of operations and to avoid multiple considerations of the same constraints in different operations.

Summarizing we can say that database design has to deal with increasingly complex structures and must take into account different aspects of databases. Apart from modelling the static structure, database dynamics has to be modelled, too. To reduce the complexity of this modelling task, there have been made a lot of efforts for developing appropriate design methodologies. Generally, several design steps can be distinguished similar to software life cycle models [8, 9].

1) Requirements analysis

2) Conceptual database design

3) Logical database design

4) Physical database design

The most demanding phase in this process is conceptual design whereas the later phases are merely transformation steps. The conceptual schema is the first formalised description of the database application. On the one hand it serves for discussions with the customer about the system functionality. On the other hand it is the basis for further design steps realizing this functionality on an existing database system. Consequently a conceptual schema should be appropriate for both of these roles. For the customer, it is important that the conceptual schema is easy to understand and represented by using suggestive and natural modelling primitives. For subsequent design steps the specification should be highly descriptive for achieving a maximum independence of implementation issues but yet be sufficiently formalised. Therefore, conceptual schemas should have a formal semantics. A formal semantics is also indispensible for obtaining reliable results from schema analysis such as consistency checking but also for the verification of the logical database schema against the conceptual one and, finally, for providing the semantical basis for query languages operating on the data model. Such a query language plays a central role in conceptual design. It provides the necessary means to "talk about" database states which is essential for formulating static and dynamic constraints as well as database manipulation actions.

A variety of approaches have been proposed to develop such a conceptual data model. An overview can be found in [10, 11, 12]. Most of these approaches, however, neglect the dynamic aspects of databases. One of the first attempts was the Entity–Relationship (ER) model introduced by Chen [13]. The basic modelling primitives of this approach are entity types, relationship types and attributes. The ER model has gained a wide acceptance for database modelling but the generality of its relationship concept has often been criticized. This is because relationships subsumes several, semantically different relationship types such as part–of–relationships, is–a–relationships, spezialisation/generalisation as well as association to object sets. For modelling these aspects so–called semantic data models were developed as for example SDM [14], IFO [15], IRIS [16] or TAXIS [17, 18]. But there have also been approaches to enhance the ER model with additional abstraction principles [19, 20, 21, 22].

As already mentioned, a powerful query language is essential to successful database modelling. A brief discussion of query languages for ER models leads us to the first proposals of [23] and

[24]. They used the relationship concept for relieving the user from specifying complicated joins. Later approaches for the ER model range from procedural languages [25, 26] over descriptive ones like GORDAS [27, 19] to graphical languages like HIQUEL [28] or GQL/ER [29].

Analogously, in the area of database (trans-)action specification descriptive and operational approaches can be distinguished. Descriptive action specification languages are usually varieties of pre/post–conditions [30, 31, 32] where preconditions are either evaluated in the previous state or in the whole database history [33, 34]. Sometimes the notion of transition is included explicitly in the specification logic in form of a modal operator [35, 36, 37, 38]. In the research community, descriptive, logic–based languages as the above have long been preferred rather than procedural ones. Nevertheless, most recently the latter ones enjoy a renaissance with the increasing interest in object oriented data models and specification techniques [39]. They are also still indispensible for the final implementation of transactions, although this has the unfavourable effect that a change of paradigm is necessary during the action design. Nevertheless specification languages (which deserve this name) based on a procedural paradigm seem not yet available. One research direction in this area are database programming languages, e.g. Pascal/R [40] or DBPL [41] providing traditional programming languages with easy database access facilities. Current research in this direction addresses persistence, typing and inheritance [42]. The major deficiency of these approaches with respect to conceptual modelling is that they are still in the spirit of modelling *database access* rather than the *behaviour* of database objects. The behaviour aspect, however, is stressed in Petri–net approaches [43, 5, 44] also proposed for action modelling.

Another area are so–called fourth generation languages (4GL) often combining descriptive and operational elements [45]. Also graph grammars have been proposed for database action modelling [46].

Only few approaches allow for a separate specification of dynamic integrity constraints to describe admissible evolutions of databases. In [47, 48, 49] transitional assertions are discussed, i.e. constraints restricting database state transitions. In [5] such constraints are integrated into Petri–net based action specifications by special constructs. There also have been proposed modal logic styles as [38, 36, 50] and temporal logic based languages as [35, 51, 33, 52, 53, 54].

The goal of this paper is to present a uniform framework for specifying all relevant aspects of a conceptual database schema, i.e. data types, object structures, (trans)actions and dynamic integrity constraints and to explain how these heterogeneous structures can be integrated. The presented conceptual data model can be provided with a formal semantics. In this paper, however, we rather aim to show the pragmatics of the approach and to demonstrate its appropriateness for database modelling.

The rest of the paper is organized as follows: In chapter 2 we discuss the requirements for conceptual design of database applications using a structured modelling approach. In chapter 3 the concepts for modelling the static structures are outlined. Additionally, we discuss the inherent integrity constraints and show how elementary database manipulation actions can be automatically derived from these structures. We also present a query language supporting the data model. Chapter 4 deals with modelling database dynamics. We present a language for specifying dynamic integrity constraints and discuss a descriptive and an operational way of specifying database actions.

2 Conceptual Modelling – A Structured Approach

We did already mention that conceptual modelling of database applications consists of adequately describing the relevant features of the application domain. The role of conceptual design in the whole database design process makes high–level, descriptive, powerful, and highly expressive modelling concepts necessary to describe static and dynamic aspects of a database in a uniform way. This chapter is intended to point out which modelling concepts are actually needed. We summarize the discovered requirements for conceptual data models and give a first outline of the proposed modelling approach described in a more elaborate way in the rest of this paper.

Conceptual specifications of database applications can easily reach a degree of complexity which makes it necessary to structure them in an appropriate manner. This complexity has several facets. On the one hand, it is a question of *size complexity*. This means that the size of conceptual database schemas can make it difficult to handle them without a design methodology which supports the development of large design documents. On the other hand, we have the *structural complexity* of the modelled structures and concepts needed to adequately model an application. This structural complexity results from the semantical richness of typical applications. It must be reflected by using adequate modelling concepts in the specification framework and languages.

In the life cycle of a database application, the conceptual database schema is the contract between future users of the application and its developers. To be appropriate for this role, a language for conceptual modelling should have a *formal semantics* in an appropriate mathematical formalism. But because of the different groups of involved people, the language should also be *easy to learn* and *easy to use* and should provide features suitable for the relevant application scenarios. These language features should also be *natural* for the application area. To avoid a complexity explosion of design documents, the language should have a high *descriptive power*.

Apart from these general principles of language design, we have a special situation for database applications concerning the structural complexity of the system to be described. To describe complete database applications, we have to specify such different concepts like

- *attribute values and operations on them*
- *database objects and their attributes*
- *relationships between objects, among them part–of and is–a relationships*
- *allowed database states (static integrity constraints)*
- *allowed evolutions of databases (temporal constraints)*
- *application specific database transactions*

Having these complexity problems in mind, we can identify two contrary approaches to handle the necessary modelling concepts in a conceptual modelling language:

1) The first approach is to use *one broad–spectrum language* for all aspects of database applications, for example an extension of usual first order logic. This choice immediately leads to a conflict between the language design principle of handiness and the complexity of the application area. Either there are only few language concepts with insufficient expressive power or there are too many different modelling concepts in one language. Specification documents tend to become unstructured and unreadable if no adequate specification methodology is additionally provided.

2) The second approach is to use a *family of independent languages* for different basic concepts of database applications (e.g., algebraic specification of data types, a semantic data model for database objects, temporal logic for evolutions, etc.) which can solve the problem of structuring documents as well as the availability of appropriate language features. The first problem with this choice is the existence of a unifying semantics for complete specification documents consisting of parts written in different languages. A maybe even harder problem is that the basic concepts of database applications are not independent from each other. Some concepts are usually needed by other concepts, for example data types for describing database states etc.

We propose a compromise between these two approaches. A database application is structured into some components according to the semantical concepts. For each of these components, we choose a language appropriate for the real world concepts to be described. This combination of languages has the following properties:

- All languages support a logic–based, descriptive style of conceptual modelling. On the other hand, for all components the support of executability is considered in the choice of the description formalisms.
- All languages are based on one uniform, formal semantics framework allowing consistency checks across the components.
- In cases where it makes sense, language primitives are shared by different languages.

The main idea is that *the languages use syntactically and semantically compatible formalisms* to support the desired properties of our specification languages.

A central component for describing a database application is, of course, the description of the database structures itself called *object component*. The object component describes the structure of objects to be stored in the database, their attributes and the relationships between objects together with the restrictions on database states expressed by static integrity constraints. The description of the database structure is done in the framework of a semantic data model derived from the well–known Entity Relationship model. Our *Extended Entity–Relationship model* (for short EER model) adds to the classical ER model new features like complex objects, generalization hierarchies, etc. which are in detail described in section 3.1. A well–defined data model has also to determine a query formalism (section 3.3) as well as the basic update primitives (section 3.5). The query language presented in section 3.3 is also used as a language to formulate static integrity constraints (section 3.4). All these sublanguages together build the language for the object component having as semantical domain the set of possible *database states*.

The object component describes the possible states of a database. For example, the interpretation of queries is changing from state to state. In contrast to this changing interpretation, the basic data structures used as values for attributes have fixed semantics independent from the current state. These values and the operations on them are described by means of *abstract data types* in the *data type component*. An example are the geometric data types **point** and **lines** together with operations on them. A data type itself can have a complex internal structure as, for example, the data type **lines**. A first hint whether a complex structure should be modelled in the object component or as a data type is given by the following observation: objects can be updated (without changing the object identity) which is not possible for data values. From a semantical point of view, the data component is the kernel of the object component because data types are used as attribute domains as well as for query results.

Based on data type and object component, we have two components describing the dynamic evolution of a database. These two components reflect two complementary viewpoints of describing such applications: the first one, the *evolution component*, describes the temporal evolutions without referring to the modifying actions (see section 4.1). This description style can be characterized as descriptive and data–oriented. The specification formalism is a temporal logic extension of the constraint language from the object component; the used semantical domain is the set of possible *database state sequences*.

The second component to describe application dynamics is the *action component*. In contrast to the evolution component, the allowed application evolutions are described in terms of the allowed actions modifying the database contents. The semantical domain for the action component is the same as for the evolution component; in fact, the action component gives an independent description of the same topic, namely the desired database state sequences. In this paper, we present two languages to describe database actions, a descriptive one based on pre- and postconditions in section 4.2 and a more operational one in section 4.3 where complex actions are composed of the basic update primitives presented in section 3.5. The language to describe pre- and postconditions is based on the query language (section 3.3).

From these considerations we can deduce a somehow *natural* structure of a database application's description as shown in figure 2.1. The three concentric circles depict different layers of the semantical structure and also an inclusion hierarchy for the specified components, for example the data type component is necessary for the description of all outer layers.

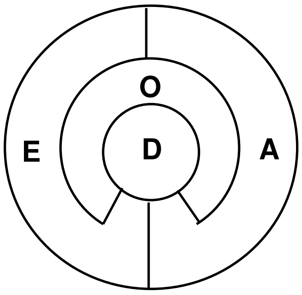

Figure 2.1: Components of a Database Application Description

In more detail, the three concentric circles of figure 2.1 stand for the different semantical domains

- domain of values (printable data) – ***Data Type Component D***
- database states – ***Object Component O***
- sequences of database states (database behaviour) – ***Evolution & Action Component* E & A**

This layered structure describes the syntactical and semantical hierarchy of the different specification parts. Object specifications refer to data type specifications whereas the database evolutions and actions are expressed by referring to object and data specifications. However, this does *not* mean that a specification document has to be built up strictly following these layers. In fact, database design is rather an iterative and incremental process. Following an object centered proceeding, designing (or augmenting) a database schema would begin with modelling (some of the) object structures. During this process the other features as data types and database behaviour are modelled around the object specifications.

Layered approaches to structure conceptual modelling formalisms are proposed among others by [55, 56, 22, 34, 57, 58]. These approaches agree in structuring conceptual models according to the handled semantic domains and specification logics, but differ in the number and the separation of the used several layers. None of them captures completely the full spectrum of the specification formalisms as presented in this paper.

Summarizing we can say that a complete specification of a database application comprises structural as well as behavioural aspects of a database. In the subsequent chapters we call such a complete specification a *database schema*. This is in contrast to most traditional approaches usually denoting by schema what is called the object component in our terminology.

After having pointed out the main aspects of database design we will give a more detailed discussion of the various specification components in the subsequent chapters. In chapter 3 we will present the specification formalisms for the structural part of a database specification. Afterwards we will discuss the languages for specifying database behaviour in chapter 4.

3 Modelling the Database – Its Structure and Access Operations

This chapter is devoted to the modelling of the structural part of a database application. This includes besides the description of the database structure, i.e., the object component of a database schema, also the presentation of languages for querying and modifying database states which are induced by the object and data type specifications.

The first section 3.1 describes the data model used for specifying the structure of database states. We use an extended Entity–Relationship (EER) model basically offering as modelling concepts *objects* ('entities') having *attributes* and participating in *relationships*. In contrast to other data models, we clearly distinguish between persistent (abstract) objects on the one side and (printable) values used as object properties on the other side. The description of value domains using abstract data types is handled in section 3.2. A given schema in the EER model induces languages for database queries, integrity constraints and basic database modifications as being presented in the following sections 3.3 to 3.5.

3.1 EER Model

The chosen example deals with world–wide air traffic, a typical database application area and often modelled to demonstrate certain modelling concepts. The ER diagram in the appendix gives an overview of our air traffic world (ATW). For modelling the air traffic world, we start with the identification and classification of the involved objects such as airplanes, passengers, etc.. After having identified the relevant object types, we continue by investigating the object properties, their attributes and the relationships between objects.

In the literature, many semantic data models have been proposed for the conceptual modelling of complex databases. These data models provide rich concepts for expressing the various structures in the different applications to be modelled. Among them, Chen's Entity–Relationship (ER) model [13] has been successfully used for describing the requirements of later database users because of its ease of understanding and its convenience in representation. It supports a style of conceptual modelling that does not depend on later implementation but can easily be mapped into implemented data models like the relational one. However, the generality of its relationship concept is often criticized. There are important relationships in the real world that have a special and fixed meaning. For example, subset relationships like specializations [59] cannot directly be modelled in the ER model. Since these concepts are very important in the real world, they should directly be supported by corresponding modelling concepts.

On the other hand, data models with a semantic hierarchy (cf. [10]) provide a lot of modelling features, and thus possess a high expressiveness. TAXIS [17], SDM [14] , SHM+ [60], IFO [15], and IRIS [16] are the most known ones. By summarizing their concepts, the following main concepts can be worked out [61] :

- *Aggregation*, for formulating part–of or property-of relationships. For example, the staff is a part of an airline company, while the name of the company or the head office are properties of it.
- *Association*, which is sometimes also called grouping or cover aggregation, is used to build sets of objects of an existing type.
- *Specialization/generalization*, to express subset or ISA relationships, e.g., each passenger is a special person.

A more detailed description of these concepts as well as an overview over semantic data models can be found in [10, 11, 12] .

We follow [19, 20, 21] and combine both approaches, i.e., we extend the ER model by additional concepts. Our version of an ER extension, called EER model in the following, was originally defined in [62] and slightly modified in [22]. Its main characteristics are:

- The EER model uses arbitrarily user–defined data types for attribute domains (see section 3.3).
- It supports all the concepts mentioned above. Thus, it possesses a high expressiveness. On the other hand, the extensions are conform to the ER model.

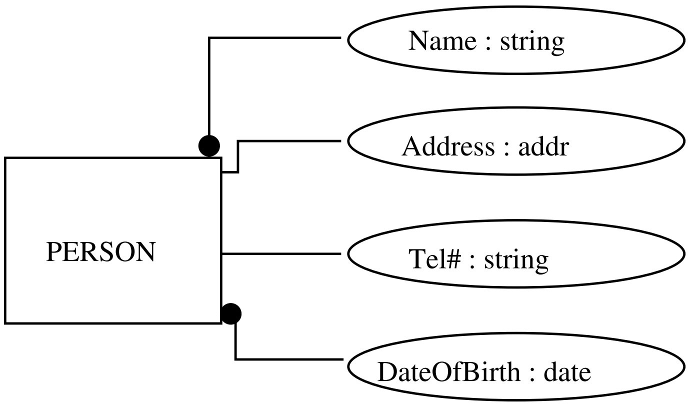

Figure 3.1: Entity type **PERSON**

- In contrast to many so–called "semantic" data models, the EER model possesses a formal semantics [63, 64, 65]. This is very important, because data models without formal semantics imply query languages and languages for specifying integrity constraints and transactions which do not possess precisely defined semantics.

Our approach starts from Chen's ER model [13]. Thus, the basic concepts are entity types, relationship types, and (data–valued) attributes. We extend this model by

- *type constructions* in order to support specialization/generalization [59],
- *object–valued* attributes, which allow a general form of aggregation,
- *multivalued* attributes for describing association types, and
- several *structural restrictions* like cardinality numbers, key specifications, etc.

In the following, we want to explain these concepts in more detail by modelling an ATW (air traffic world).

Entity types E:

Similar objects with common properties are summarized to entity types. As in the ER model, entity types are graphically represented by rectangles. Figure 3.1 shows the entity type **PERSON**.

We formally express the semantics of entity types by a *database state* σ yielding for each entity type **E** a set of current instances. Thus, σ**(PERSON)** represents the set of currently existing instances of type **PERSON**. σ**(PERSON)** can change in time, persons can be inserted into or removed from the database.

Entities are abstract items and thus not printable. Only the *properties* of entities are printable, and these are represented by the values of their attributes.

Attributes a:

Attributes describe the properties of an entity. Every attribute belongs to an entity type **E** and possesses a domain **d** of values. We denote this fact by **a:E** $\rightarrow$ **d**. Each attribute **a** is graphically represented by an oval that is connected with the type (rectangle) **E** and contains **a:d**. Due to the nature of the domain **d**, we can distinguish several cases:

- *data–valued* attributes, if the domain of an attribute is a data type. This is the usual case in the ER model. We can use as domains standard types like **int** or **real** as well as non–standard data types like **point** or **addr**. The latter ones must be specified at the data type component (see section 3.2). In figure 3.1, the entity type **PERSON** has the attributes **Name** (with the data type **string**), **Tel#** (data type **string**), **Address** (data type **addr**), and **DateOfBirth** (data type **date**).
- *object–valued* attributes, i.e., the attribute domain is an entity type. Consequently, the attribute value is an instance of this type. In the EER diagram, object–valued attributes are denoted by a ‘ □—— ’ symbol together with a connecting arc to the domain type.
- Both data– or object–valued attributes can be *multivalued*: An attribute value may be a set or a list of values of the corresponding domain. Considering figure 3.2, **Nationality** of **PASSENGER** is list– and data–valued, while **PlaneCrew** of **NON-STOP-FLIGHT** is set– and object–valued. In contrast to sets, an element may occur more than once in a list. Moreover, lists have their elements enumerated so that we can reference them by their position number. Multivalued, data–valued attributes can be modelled in other ER approaches like [19, 21], too.

By default, every attribute is *optional*, i.e., there need not exist values for the attribute. Optional attributes may take the value “unknown”. For example, the telephone number (**Tel#**) of a person may be unknown.

In comparison with other semantic data models, our notion of attribute supports the general form of aggregation (like in SHM+, TAXIS, or SDM) completely since we can use data– and object–valued attributes together. Furthermore, the combination of object- and multivalued attributes allows us to model complex structured entity types [66, 6, 7, 67]. Many other ER approaches do not directly support associations. However, we have resolved this problem by the use of multi– and object–valued attributes. For example, we could have modelled an association **TOURIST-PARTY** over **PASSENGER** by an entity type **TOURIST-PARTY** with a set– and object–valued attribute **Participants**.

Let us have a brief look at the semantics of attributes. Each attribute **a:E** $\rightarrow$ **d** is interpreted by a function σ(a) that yields for each instance (entity) of σ**(E)** a value of the attribute domain, i.e., a data value, an entity, or a set resp. list of them.

Relationship types r($\mathbf{E}_1$, ..., $\mathbf{E}_n$):

Relationship types are the usual form of aggregation in ER approaches. Relationship types are aggregations of several entity types $\mathbf{E}_1,\ldots,\mathbf{E}_n$ ($n \geq 2$). Thus, concrete entities of these types form relationships in the world to be modelled. Figure 3.2 presents two relationship types

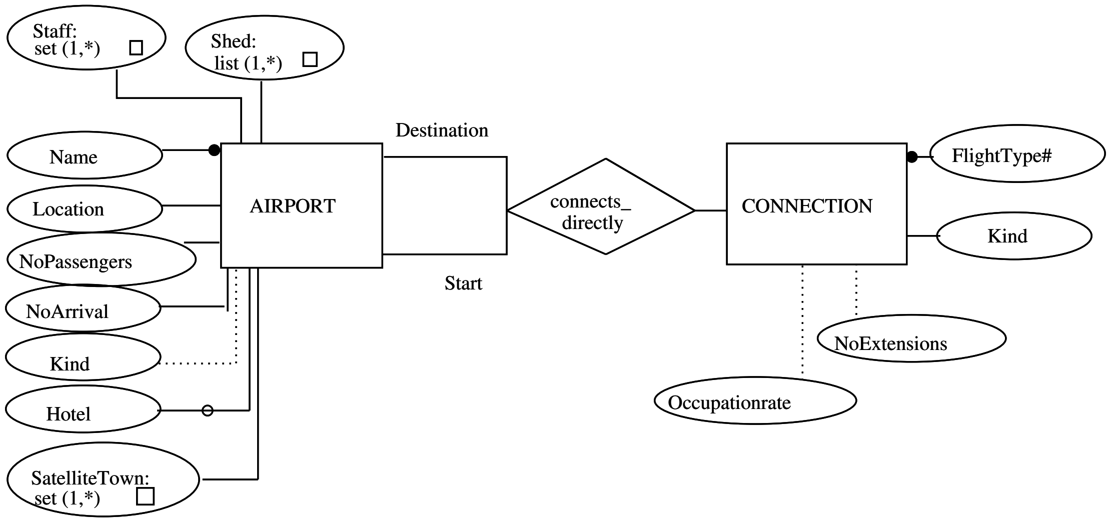

Figure 3.2: Relationships **booked-for** and **connects-directly**

shown as diamonds: a binary type ($n = 2$) **booked-for** between **PASSENGER** and **NON-STOP-FLIGHT** and a ternary type ($n = 3$) **connects-directly** between **AIRPORT** (twice) and **CONNECTION**. Please notice that the entity type **AIRPORT** participates in this relationship twice. We can distinguish the different roles an airport plays in **connects-directly** by means of *rolenames*. Thus, one airport is the **Start** point of a connection, and the other one is the **Destination**.

The semantics of a relationship type **r** can be understood as a relation $\sigma(\mathbf{r}) \subseteq \sigma(\mathbf{E}_1) \times \ldots \times \sigma(\mathbf{E}_n)$. Each relationship (instance) is a tuple of entities of corresponding types.

Relationship types can also have attributes, like **BookingDay** of **booked-for**. However, we only allow data–valued attributes because object–valued attributes should be modelled by additionally participating entity types. The semantics $\sigma(\mathbf{a})$ of relationship attributes is directly carried on from the one of entity type attributes.

Up to now we have extended the basic concepts of the ER model. Using these concepts, aggregation and association can be modelled, but no subset or ISA relationships known as specialization/generalization [59]. We introduce the concept of type construction for this purpose.

Type construction:

A type construction can be regarded as a new classification of the entities from certain types. Starting with already defined entity types $\mathbf{I}_k$, $(k = 1, \ldots n, n \geq 1)$, called *input* types, the new *output* types $\mathbf{O}_j (j = 1, \ldots, m, m \geq 1)$ are constructed by classifying the entities of the input types newly in output types. Consider figure 3.3 where we present the general form of type construction.

At the base line of the triangle, there are the already defined input types $\mathbf{I}_1, \ldots, \mathbf{I}_n$ $(n \geq 1)$. The types $\mathbf{O}_1, \ldots, \mathbf{O}_m$ $(m \geq 1)$, connected with the opposite point of the triangle, are the constructed output types. The following conditions must hold between the entities of the input and the output types of one type construction and for nonconstructed entity types :

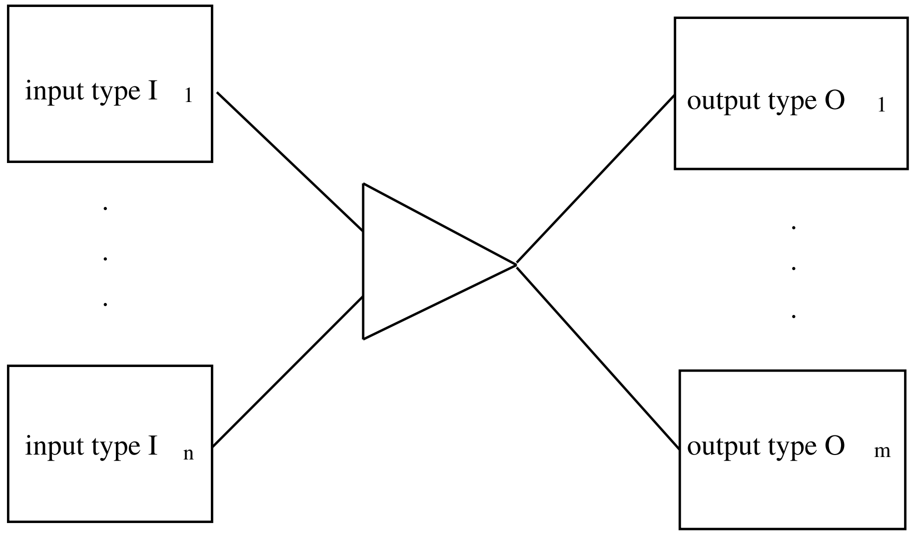

Figure 3.3: General form of type construction

(i) $\bigcup_{k=1}^{n} \sigma(I_k) \supseteq \bigcup_{j=1}^{m} \sigma(O_j)$

(ii) $\sigma(O_p) \cap \sigma(O_q) = \emptyset$ for $p \neq q$

(iii) $\sigma(E_p) \cap \sigma(E_q) = \emptyset$ for $p \neq q$, E_p, E_q non–constructed.

Condition (i) means that all entities of output types $\mathbf{O}_j$ are also instances of input types $\mathbf{I}_k$, but not all entities of input types have to be instances of output types due to the inclusion. Consequently, the entities from the output types are not new entities, they already exist (in the input types), but will now be seen in a new context (given by one of the output types). Secondly, (ii) requires that the output types are disjoint; no entity (from any input type) may occur in several output types. Finally, we demand by (iii) all non–constructed entity types, i.e. entity types that are not output type of any type construction, to be disjoint. In other words, type construction provides the only possibility to model subset relationships.

Given the above description of semantics, we now show the modelling of specializations and generalizations. Let us therefore consider figure 3.4.

At first, a simple specialization **PASSENGER** of **PERSON** is defined. This is the case of one input type **PERSON** (n=1) and one output type **PASSENGER** (m=1). Considering (i) of above, we obtain the following semantic condition:

$\sigma(\mathbf{PERSON}) \supseteq \sigma(\mathbf{PASSENGER})$,

i.e. each passenger is a person, too. But the inverted direction does not hold.

In the same way, we have specified another specialization **STAFF-MEMBER** of **PERSON** that formally satisfies $\sigma(\mathbf{PERSON}) \supseteq \sigma(\mathbf{STAFF\text{-}MEMBER})$. Both constructed types **PASSENGER** and **STAFF-MEMBER** are independent of each other. Especially, a person can be a passenger as well as a staff member, i.e., $\sigma(\mathbf{PASSENGER}) \cap \sigma(\mathbf{STAFF\text{-}MEMBER}) = \emptyset$ need not hold in general.

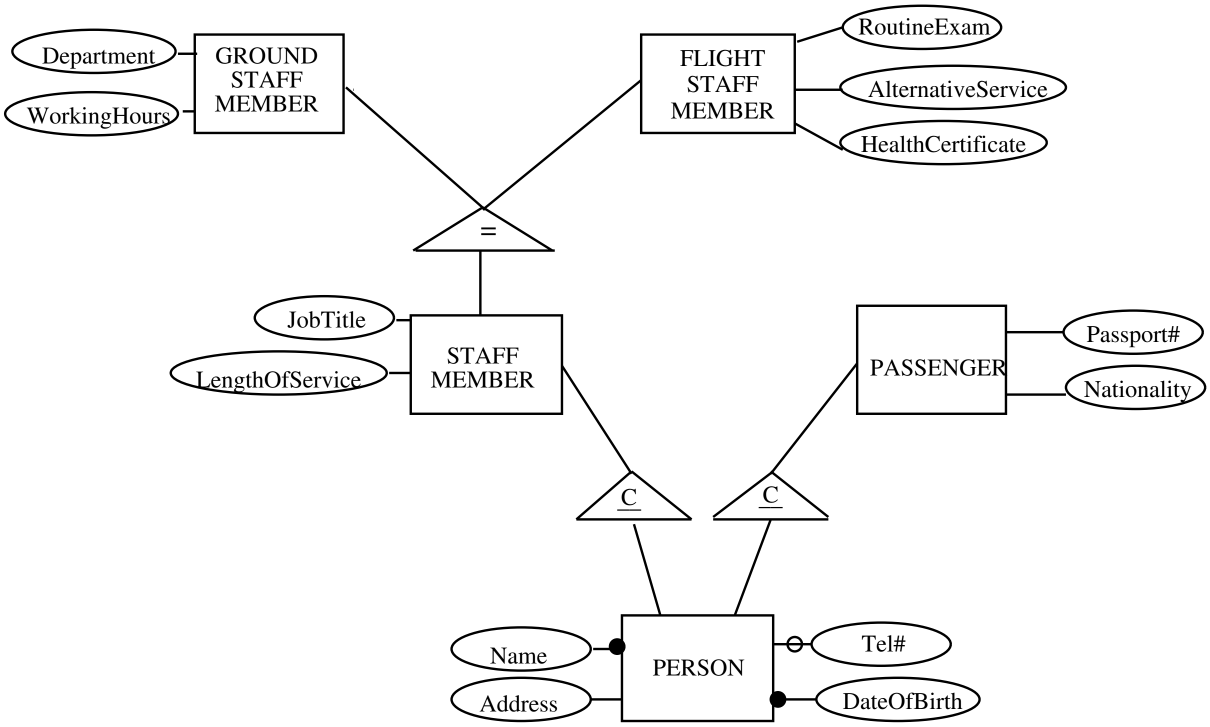

Figure 3.4: Type constructions

To achieve this disjointness, we use a type construction with several output types. In this way, the disjoint specializations **GROUND-STAFF-MEMBER** and **FLIGHT-STAFF-MEMBER** of **STAFF-MEMBER** are modelled. Due to (i) and (ii), we obtain the following semantics:

$$\begin{aligned}\sigma(\textbf{STAFF-MEMBER}) \quad &\supseteq \quad \sigma\ (\textbf{FLIGHT-STAFF-MEMBER}) \\ &\quad\ \cup\ \sigma\ (\textbf{GROUND-STAFF-MEMBER}),\end{aligned}$$

$$\sigma\ (\textbf{FLIGHT-STAFF-MEMBER})\ \cap\ \sigma\ (\textbf{GROUND-STAFF-MEMBER}) = \emptyset$$

This corresponds to the disjointness condition of [11, 15].

Finally, we show how to express generalizations using type constructions with several input types ($n > 1, m = 1$). For example, we would obtain a different effect if we altered the two specializations (**PERSON** to **PASSENGER** and **PERSON** to **STAFF-MEMBER**) to a generalization with the input types **STAFF-MEMBER** and **PASSENGER** and an output type **PERSON**:

$$\sigma(\textbf{STAFF-MEMBER}) \cup \sigma(\textbf{PASSENGER}) \supseteq \sigma(\textbf{PERSON})$$

Now, we would only store persons that are either staff members or passengers. There are no other persons of interest.

We see the general concept of type construction covers the known data abstractions generalization or superclasses ($n > 1, m = 1$) and (possibly disjoint) specialization or subclasses ($n = 1, m \geq 1$).

The view of seeing specialization/generalization as a type construction suggests two syntactic restrictions:

(i) Every constructed entity type has to be the result of exactly one type construction.

(ii) Every constructed type must not, directly or indirectly, be an input type of its own construction, i.e., the directed graph consisting of all entity types as nodes and type constructions as edges must be acyclic.

While the second condition is generally requested by all semantic data models, the first one is sometimes omitted, as for instance in IFO and SDM.

Strictly related to specialization/generalization (or type constructions in our terms) is the notion of inheritance. In the literature, an agreement is reached in the case of specializations. Subtypes then inherit all the attributes from the supertype. Related to our EER model, each staff member inherits the attributes of **PERSON**, and each flight staff member the attributes of **STAFF-MEMBER**. Thus, we can refer to **Name** of **PASSENGER** or to **DateOfBirth** of **FLIGHT-STAFF-MEMBER**. This is the usual way of inheritance in semantic data models like IFO or SDM. Of course, naming conflicts can occur. In this case, we have to use the corresponding (input) type names as prefix. This understanding of inheritance can also be used for several output types.

But how about the general form of type construction, especially type constructions with several input types, which attributes are now inherited? SDM proposes to inherit the attributes common to all input types. However, attributes could have the same name, although they have different meanings (homonyms). Otherwise, if all the attributes of all input types are inherited, naming conflicts appear quite often. Particularly, synonyms in different input types lead to several attributes in output types having the same meaning.

Therefore, we do not offer inheritance in the case of several input types. However, we can use *derived attributes* (see later in this section) in order to carry on the attributes explicitly. In our opinion it is more advantageous to leave the specification of inheritance to the user.

There are some further concepts that do not provide any modelling primitives but rather *structural restrictions* on the possible database contents. Thus, they form a special case of general integrity constraints (see section 3.4). Their frequent use just as their close relation to modelling primitives advise us to offer explicit concepts with an explicit notation.

- We can give object-valued attributes more semantics than rather yielding an entity (or a set resp. list of entities) of another type. Object–valued attributes establish references which may be rather loose or very close. An example for the second case is the entity type **MAINTENANCE-SHED**. A **MAINTENANCE-SHED** may only exist as one of the **Shed**s of one unique **AIRPORT** and remains with this airport during the whole time it exists. We say every entity of type **MAINTENANCE-SHED** is *dependent* on the entity of type **AIRPORT** it is associated with. This kind of entity is also known as the concept of weak entity already presented in [13]. Our graphical notation for dependency is a broad connecting arrow (' $\Box\!\!\longrightarrow$ ') from the dominant entity type to the dependent one. Dependency has also consequences on the way dependent objects are identified. In general, object identity is modelled by key attributes as discussed later on in this section. Since dependent objects only exist in the context of their parent object it is sufficient to have a local key for identifying the dependent objects in their context. For example the attribute **Name** of **MAINTENANCE-SHED** identifies each maintenance shed within the set of all sheds belonging to one airport. This means

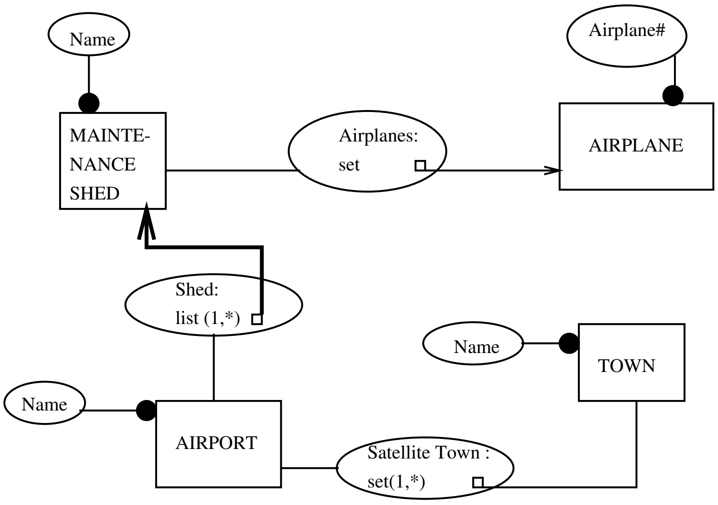

Figure 3.5: Different kinds of object–valued attributes

different maintenance sheds may have the same name unless they do not belong to the same airport. For a global identification of a dependent object, the dominating object, i.e. the values of its key attributes, must be specified additionally to the local key.

Another consequence of dependency is that sets of dependent objects dominated by different entities are always disjoint. *Disjointness* may also be a desired property of non–dependent object–valued attributes. For example, the sets of airplanes of different maintenance sheds are disjoint, i.e. no airplane can be maintained at several sheds. This is expressed by an arrow (' □⟶ ') between the object–valued attribute **Airplanes** of **MAINTENANCE-SHED** and the entity type **AIRPLANE**.

Possibly non–disjoint and non–dependent object–valued attributes are finally described by a line instead of an arrow, as can be seen for the attribute **SatelliteTown** of **AIRPORT**. Here, the same town may be one of the satellite towns of different, neighbouring airports.

With respect to dependency we require pure hierarchies. Thus, cycles are not allowed just as sharing of dependent types is not possible. These requirements are not necessary for "normal" object–valued attributes, which specify non–dependent references.

Please notice the effect of dependent attributes on update propagation. If we want to insert an entity of a dependent type, we also have to update the corresponding attribute of an entity of the "parent" type. Similarly, the deletion of an entity from the "parent" type requires the deletion of all dependent entities from the "child" types (cf. section 3.5).

- Non–multivalued attributes can be specified as keys. Key attributes are mandatory. *Key attribute* values identify entities in their types. For example, **Name** and **DateOfBirth** are the key attributes of **PERSON**, graphically marked by a broad dot (**'•').** This means that different people must not have the same name and the same date of birth. Similar to dependency, object–valued key attributes must not form a cycle. It is important that only non–constructed entity types (that are not output of any type construction) can have keys. The entities in constructed types inherit their identity from the corresponding entity in the input types.

We have a special effect for dependent types: As mentioned before, dependent entities are identified by their own key attributes and the ones of their parent entity. For example, several maintenance sheds may have the same name, provided they belong to different airports. Consequently, the key attributes of **MAINTENANCE-SHED** are composed of those of **AIRPORT** and its own ones. However, dependent entity types need not have any own key attributes, as it is in case of the entity type **TIME-TABLE**.

By summarizing, non–constructed and non–dependent entity types must have key attributes, whereas dependent entity types can have some, but constructed types must not have them.

- Type constructions can be constrained to the semantics

  $$\bigcup_{k=1}^{n} \sigma(I_k) = \bigcup_{j=1}^{m} \sigma(O_j),$$

  graphically denoted by a '=' symbol inside the triangle. This means that every entity of an input type must occur in one of the output types. This allows us the specification of *covered specializations* known from IFO, ECR [19], or SDM. For example, **STAFF-MEMBER** is partitioned into **GROUND-STAFF-MEMBER** and **FLIGHT-STAFF-MEMBER**.

- Finally, we can specify *derived information* that is not explicitly stored, but can be computed from other stored information [19].

  Example 3.6: We can compute the pilot of a non stop flight **nsf**, assuming (s)he is the first one in the plane crew list, by

  nsf.Pilot ← **nsf.PlaneCrew[1]**

  Please note that the attribute **Pilot** would not be stored at all; if we refer to it, it will automatically be computed corresponding to the above rule. We can use the full power of our query language, presented in section 3.3, for the specification of rules.

  Example 3.7: A more complex example computes the occupation of a non stop flight **nsf** by counting bookings:

  nsf.Occupation ← ***cnt*(*select* bf**
  ***from* bf *in* booked-for**
  ***where* bf.NON–STOP–FLIGHT = nsf)**

  Derived attributes like **Occupation** are graphically denoted by a dotted line.

  The concept of derivation generally provides an easier access to information. Instead of explicitly computing the desired information by a query, we can simply refer to **Occupation** of **NSF**. However, the computation is still done but now implicitly during the execution of the rule. Thus, the effect of derivation is very similar to the views in relational database systems. Indeed, derived attributes provide a more powerful tool for specifying views. Entity types, relationship types, and type constructions can be specified as derived as well. There are a lot of useful applications.

A quite often but not explicitly defined relationship is **flies** between **AIRLINE-COMPANY** and **NON-STOP-FLIGHT**: A concrete instance holds iff an airline company has chartered an airplane (via **chartered**) and this airplane is assigned to the non stop flight. This **flies** could have been defined derived.

Automatic or predicate–defined partitions, known from SDM, can be modelled by derived type constructions. In our example, we could model the specializations of **STAFF-MEMBER** into **GROUND-STAFF-MEMBER** and **FLIGHT-STAFF-MEMBER** as derived using the **JobTitle** attribute of **STAFF-MEMBER**. Thus, the job title 'Pilot' implies the membership to **FLIGHT-STAFF-MEMBER**, while the title 'Mechanic' indicates a ground staff member, and so on.

Note that rules do not generate *new* objects but only further properties of existing objects, e.g. the property of being a **FLIGHT–STAFF–MEMBER**. Finally, derived attributes play an important role for inheritance in case of type constructions with several input types. As mentioned above, inheritance must here be specified explicitly. The use of rules enables us to rename attributes, to unify attributes from different input types, etc.

3.2 Data types

Data types constitute the domain of printable data in a database application. For instance, data types are used for the definition of attribute domains of entity or relationship types. But, in our approach a data type is not only a collection of values. A data type is characterized also by the way these values are used. Therefore, a data type consists of a set of *values* together with *functions* and *relations* defined on them. Values are instances of a certain data sort and the functions and relations are named by data operations and data predicates.

The most simple data types are data sorts like **nat**, **int**, **real** (for natural, integer and decimal numbers, resp.) together with data operations like $+$, $*$, etc. and data predicates like $=$, $<$, $\leq$, etc. These numerical types are examples of *standard data types* which are frequently used and therefore supported by our framework. We also include boolean values (data sort **bool**), single characters (data sort **char**), strings of characters (data sort **string**) together with appropriate operations and predicates. The names of these data types and their operations are determined by a data type *signature*:

sorts **nat, int, real,** ...
operations $+$ **: int** $\times$ **int** $\rightarrow$ **int** ; ...
predicates $<$ **: int** $\times$ **int** ; ...

The semantics σ associates sets, functions and relations with the given names. But unlike the interpretation of (e.g.) entity and relationship types the semantics of data types does not change in time, but is fixed once for all database states, for example $\sigma(\mathbf{int}) := \mathbb{Z}$.

In addition to these predefined standard data types, also application–dependent data types can be defined. In our example, data types like **date**, **point**, or **line** occur as attribute domains for the attributes **DepartureDay** of **NON-STOP-FLIGHT**s, **Location** of **AIRPORT**, resp. **Route** of **NON-STOP-FLIGHT**.

In the literature, various styles for the description of data types are proposed. Among them are

(i) descriptive approaches, for instance, by means of algebraic equations [68, 69] which are quite implementation independent,

(ii) constructive approaches, for instance by use of predefined type constructors [70, 71] or even

(iii) programming language–like type definitions with procedures being very close to implementation.

As an example for (i), we sketch the definition of the data types **point** and **line**, which consist of a data type signature together with a set of equations to define the meaning of operations and predicates.

Example 3.8:

sorts **point, line**
operations **make-point : real** $\times$ **real** $\rightarrow$ **point;**
x-coord, y-coord: point $\rightarrow$ **real;**
make-line : point $\rightarrow$ **line;**
add-point : point $\times$ **line** $\rightarrow$ **line;**
point-dist : point $\times$ **point** $\rightarrow$ **real;**
length : line $\rightarrow$ **real;** ...
variables **x, y : real ; p1, p2 : point ; l1, l2: line**
equations **x-coord(make-point(x,y)) = x**
y-coord(make-point(x,y)) = y
point-dist(p1,p2) = sqrt(exp (x-coord(p1) - x-coord(p2), 2) +
exp (y-coord(p1) - y-coord(p2), 2))
length(make-line(p1)) = 0
length(add-point(p2, makeline(p1))) = point-dist(p1, p2)
length(add-point(p2, add-point(p1,l1))) = point-dist(p2,p1)+
length(add-point(p1,l1))
...

The operation **sqrt** computes the square root, **exp** stands for exponentiation, and + and < for the usual addition and comparison of decimal numbers. All are imported from the data type **real**.

Our constructive approach to data type definitions is based on four *data type constructors*. These are the data type constructor ***set***(**<type>**) to define sets of a certain **<type>**, ***list***(**<type>**) for lists, ***record***(**<type**$_1$ **>**,...,**<type**$_n$ **>**) for cartesian products, and ***bag***(**<type>**) for multisets (or bags for short). The difference between sets and multisets is, that multisets can contain multiple occurrences of the same value. There are numerous operations and predicates associated with these constructed types.

For example, for every data type constructed by ***set***(**<type>**) we have the following operations and predicates with the usual semantics:

Operations for ***set***(**<type>**) :
cnt: ***set***(<type>) $\rightarrow$ ***int***; **/* counts the number of elements */**

insert: **<type>** × ***set***(**<type>**) → ***set***(**<type>**);
delete: **<type>** × ***set***(**<type>**) → ***set***(**<type>**);
union: ***set***(**<type>**) × ***set***(**<type>**) → ***set***(**<type>**);
...
Predicates for ***set***(**<type>**) :
is-empty: ***set***(**<type>**);
in: **<type>** × ***set***(**<type>**);
= : ***set***(**<type>**) × ***set***(**<type>**);
...

Other operations and predicates for other data type constructors will appear later in this section and will be explained there.

All type constructors can be used in a totally orthogonal way, e.g. we allow ***record***(***set***(***int***),***list***(...)), ***set***(***record***(...)), etc.

As an example for the use of these data type constructors, we sketch a very simplified definition of the data type **date**. As usual, this data type is constructed as cartesian product of day, month, and year. This is described by use of the ***record***–constructor. Implicitly associated projection operations **_.i** to select the i–th component of a value of this type can then be used to define the meaning of projection operations **day**, **month**, and **year**. Based on these, further operations can be defined like **monthdiff** to compute the number of months between two dates.

Example 3.9:

sorts **date = *record*(int, int, int)**
operations **day, month, year : date → int;**
daydiff : date × date → int;
monthdiff : date × date → int;
predicates **before : date × date;**
variables **d, d1, d2 : date;**
equations **day(d) = d.1**
month(d) = d.2
year (d) = d.3
monthdiff(d1, d2) = abs(12 ∗ (year(d1) - year(d2)) +
abs(month(d1) - month(d2))
...

Another class of operations not mentioned above are so–called aggregation functions as for example ***sum***, ***max***, ***min***, ***avg***. These aggregates are defined for types which are constructed by ***list***, ***bag***, or ***set*** and yield as result a single value of the argument type of the construction. In contrast to the operations mentioned earlier, aggregation functions require appropriate functions and predicates defined for the argument type, e.g. ***sum*** and ***avg*** need a **+**–operator, ***max*** and ***min*** need comparison predicates $<$ or $>$.

In principle, data types could be specified in a pure algebraic way without using explicit data type constructors. But, data type constructors are indispensible for determining the type of a query result, because, in general the result type is not specified explicitly in the conceptual schema. For example, the result type of a query may be something like ***bag***(***record***(**string,**

int)). The type of a query result determines which operations are allowed on it. These are the predefined operations associated with the corresponding type constructors.

To support *optional attributes* and *incomplete information* via null values, there is a special value in every (interpretation of) a single data sort representing "undefined"[63]. The predicate ***is null*** defined for every data type allows to test on definedness.

Finally, we shortly discuss the *role of data types* in the conceptual modelling process: We allow suitable data types for every application; thus we are close to "real world" in the names for data sorts and their operations and predicates. Our approach supports abstraction: single items stand for complex structures (e.g. variables or terms of sort **circle**, **lines**). Queries use data types as result items; thus entities can be "observed" only by means of data types and (later we will see in more detail that) the structure of query results depends on the data type constructors.

3.3 Queries

The EER model which is presented in section 3.1, provides powerful constructs for modelling real world information structures. This ensures that the relevant information about real world states can be mapped to database states in a natural way. Besides this also a powerful query language is indispensible for database modelling. It provides the necessary means to "talk" about database states when specifying (static) integrity constraints (cf. section 3.4), derived information (cf. section 3.1) and allowed dynamic database behaviour (cf. chapter 4).

Up to now a lot of ER query languages have already been developed. Nevertheless, we defined a new language because almost all of them are based upon different, specific variants of the ER model. Furthermore, only few of these languages possess a formal mathematical semantics.

First proposals are CABLE [23] and [24]. They showed how the relationship concept can be used for avoiding the formulation of joins, which seems to be imperative but tiresome in relational query languages. On the other hand, both languages are of restricted expressiveness. In the meantime, a great variety of more powerful languages exist. Starting with procedural languages [25, 26], more and more descriptive ones like GORDAS [27, 19] have been developed. In order to support easy query formulation, some approaches like ERROL [72] try to give their language the flavour of natural language sentences. For the same purpose, some graphical approaches like HIQUEL [28], GQL/ER [29], or a graphical variant of GORDAS [73] have been proposed. Certain database browsers [74] can also be seen as query languages for ER models. Being less powerful than other approaches they are rather aimed at the class of casual users than at sophisticated ones. Finally, more recent approaches take into account further concepts extending the ER model. LAMBDA [75] allows the retrieval of structured documents, whereas DESPATH [76] supports subtypes and hierarchy relationships. CERMo-QL [77] is based upon an object–oriented extension of the ER model, and [66, 6, 78] present a language for geoscientific applications.

Among the various database languages, SQL is probably the best known approach. It is the standard language in relational database systems and has successfully been adapted for NF^2 data models (cf. HDBL [79, 80, 81] or SQL/NF [82]). Thereby the language has gained more orthogonality by taking into account Date's critique [83].

We follow these approaches and propose an SQL–like query language for our EER model. Due to the rich expressiveness, we obtain a powerful, high–level, and completely orthogonal language. However, by describing the language, we have attached great importance to rigorous formal semantics [63, 64, 65, 84].

Our language supports all concepts of the EER model, e.g., relationships, object–valued attributes, attributes of relationships, and type construction. Furthermore, we have incorporated concepts that are established in contemporary languages like arithmetic and aggregate functions or explicit control over null values and duplicates. Especially the handling of set–valued terms allows nesting and unnesting, known from the NF^2 model [85, 86], and a clean and sound use of aggregate functions.

Analogous to relational SQL [87], our language uses a ***select–from–where*** block:

> ***select*** <term$_1$>,...,<term$_n$>
> ***from*** <variable$_1$> ***in*** <range$_1$>,...,<variable$_k$> ***in*** <range$_k$>
> ***where*** <formula>

We recognize the basic concepts of SQL, namely:

- a list of target terms <term$_1$>,...,<term$_n$>, which compute the desired information,
- a qualifying <formula>, and
- a list of declarations, each one of the form <variable> ***in*** <range>, declaring variables for their use in the terms or in the formula.

But in contrast to SQL, we do without a ***group–by*** or ***having*** clause. This does not cause any restriction w.r.t. the functionality of the language, however, it is of benefit to the orthogonality of the language.

The formal semantics and nearly all language features are based on an EER calculus [63, 64, 65]. The calculus will not be explained here. Nevertheless, we emphasize all constructs of the language are defined on mathematically precise foundations.

[84] presents a formal mapping of query language to the calculus. Let us illustrate the syntax and informally the semantics by means of a simple query:

Example 3.10: *"Airplanes that are not airworthy"*

> ***distinct select*** **a.PlaneModel**
> ***from*** **a** ***in*** **AIRPLANE**
> ***where not*** **a.Airworthy**

The target list contains one term **a.PlaneModel** applying the attribute **PlaneModel** to the variable **a**. The variable **a** itself is declared in the ***from***–clause: **a** ***in*** **AIRPLANE** means that **a** is bound to the set of currently (in the state σ) existing airplanes; we say the range of **a** is **AIRPLANE**. The built–in function ***distinct*** is used to suppress duplicates. If we omitted ***distinct***, the result would contain duplicate plane models. The formula **a.Airworthy** is a **bool**–valued term which only selects the airworthy airplanes.

Since the attributes **PlaneModel** and **Airworthy** can clearly be related to the range **AIRPLANE**, we can omit the variable **a** in both terms, thus resulting in simply **PlaneModel** and **Airworthy**. Furthermore, there is no need for explicitly introducing the variable '**a**' by a declaration so that we can omit the part '**a** ***in***', too.

We now summarize the rules for building terms:

- Variables like **a** (of type **AIRPLANE**) or **bf** (of type **booked-for**) are the elementary form of terms. Every variable must be declared in an appropriate declaration, denoting a finite range (for the details see [65]) e.g., **a *in* AIRPLANE** or **bf *in* booked-for**, in the ***from***–part.

- By using variables **x**, we can build terms **x.a** if **a** is an attribute of an entity or relationship type and **x** has exactly this type. For example, **a.PlaneModel** is a correct term, but **a.Name** is not a syntactically correct term because **Name** is not an attribute of **AIRPLANE**. Please note that we are able to refer to relationship attributes. For example, **bf.BookingDay** yields the booking day of an instance of type **booked-for**, i.e., the day a passenger books a flight. Any object- or multivalued attribute can be used. Thus, **a.DateOfMaintenance** is a term yielding the list of dates of the airplane **a**. Concatenations defining access paths are also possible. Assuming **Pilot** is an object–valued attribute **Pilot: NON-STOP-FLIGHT→PERSON**, **nsf.Pilot.Name** computes the name of the pilot of the given flight **nsf**.

- Having data–valued terms, data operations can be applied. Since **nsf.Route** is a term of the sort **line**, the length of such a route can be computed by **length(nsf.Route)**. Similarly, arithmetical operations like '+' or '**exp**' ('power of') are applicable.

- There are a lot of built–in functions. ***distinct*** eliminates duplicates as shown above in example 3.10. Furthermore, special functions are offered to handle lists: [**i**] selects the i–th element of a list, where the variable '**i**' can be bound to the set of currently used indices computed by ***ind***. For example, **a.DateOfMaintenance**[**1**] computes the first date of maintenance of the airplane **a**, while ***ind*(a.DateOfMaintenance**) yields the set $\{1, 2, 3, \ldots, k\}$ of list indices (where k is equal to ***cnt*(a.DateOfMaintenance)**). Finally, ***cnt***, ***sum***, ***min***, ***max***, and ***avg*** are the usual aggregate functions.

- Any subquery of the form ***select–from–where*** is a special kind of multivalued term.

- Special concepts are related to the EER model, especially type construction and relationships. Having a variable **x** related to a relationship type, we can select the participant $\mathbf{E}_i$ of a concrete relationship by $\mathbf{x.E}_i$. In this way, **bf.NON-STOP-FLIGHT** computes the flight of an instance of type **booked-for**. If $\mathbf{E}_i$ participates more than once in a relationship rolenames must be used to select the participant. Furthermore, we can trail subset hierarchies given by type constructions. For example, **sm.PERSON** converts the staff member **sm** into the person she/he really is (in the sense of type construction). Now, we can formulate **sm.PERSON.Name** in order to compute the name of this person. Similarly, **sm.GROUND-STAFF-MEMBER** converts the staff member **sm** into a ground staff member. However, if the staff member is rather a flight staff member than rendering service on the ground, the result will be the null value 'undefined' (we have null values for object types, too). Please note that the first form, i.e., **sm.PERSON**, makes an implicit inheritance (explained later) explicit.

We now present some applications of terms. The next example uses a subquery as a target term as well as a range. Thus, we obtain a nested query as known from the NF^2 model [86]:

Example 3.11: *"For each plane model the set of planes"*

select **model, (*select* a.Plane#**
from a _in_ AIRPLANE
where a.PlaneModel = model)
from **model *in distinct* (*selec*t a1.PlaneModel**
from a1 in AIRPLANE)

Let us recall the intuitive semantics given above. Thus, this query binds the variable **model** to the finite set of current airplane models, and computes for each of the models the bag of plane numbers belonging to it. The missing ***where*** formulas are assumed to be true. The result then looks like

$\{\!\{$ (model$_1$, $\{\!\{$ no$_{1,1}$,no$_{1,2}$,. . ., no$_{1,k1}$ $\}\!\}$),(model$_2$, $\{\!\{$ no$_{2,1}$,no$_{2,2}$,. . ., no$_{2,k2}$ $\}\!\}$), . . . $\}\!\}$

As mentioned in section 3.2, the result type of a query is constructed from defined data types using data type constructor instantiations. The result type of query 3.11 is given by

bag **(*record* (string, *bag* (*record* (int))))**

Please note that the variable **model** declared in the outer part is still valid in the inner ***select–from–where***–block. However, the variable **a1** cannot be used outside its block.

In this query, we have used a ***select–from–where*** term as the range for the variable **model**. Indeed, besides entity and relationship types, we can use any multivalued term as a range, especially subqueries. Furthermore, ranges can be united by using the form <range$_1$> ***union*** . . . ***union*** <range$_r$>. In any case, every variable is bound to a finite value set.

Example 3.12: *"Airline companies together with the names of their staff members."*

select **ac.Name, sm.Name**
from **sm *in* ac.Staff, ac *in* AIRLINE-COMPANY**

Here, we use the set–valued term **ac.Staff**, yielding the set of staff members of an airline company, as a range for the variable **sm**. However, the evaluation of this term depends on the variable **ac**. Thus, both declarations must be seen together: We first bind **ac** to an airline company, and then **sm** to a staff member of this company.

In this query, we are confronted with 'inheritance' for the first time. We see that we can refer to a staff member's **Name** although **Name** is not an attribute of **STAFF-MEMBER**. This is possible because **STAFF-MEMBER** inherits the **Name** attribute from **PERSON**. In order to make inheritance explicit, we can also use the long form **sm.PERSON.Name** applying the conversion **PERSON:STAFF-MEMBER→PERSON**. Inheritance happens even in case of several type constructions. Thus, **GROUND-STAFF-MEMBER** inherits the attributes of **PERSON** and **STAFF-MEMBER**. Please note that no inheritance occurs in the other direction: We have to explicitly state **p.STAFF-MEMBER.JobTitle** to refer to the job title of a person. In a similar way, the participation in relationships is inherited in the direction of the type construction.

Since the use of aggregate functions is different from that in SQL [87], we present an example for aggregates:

Example 3.13: *"For each plane model the number of planes"*

select **a1.PlaneModel, *cnt* (*select* a2.Plane#**
from a2 _in_ AIRPLANE
where a2.PlaneModel = a1.PlaneModel)
from **a1 *in* AIRPLANE**

In principle, this query is very similar to example 3.11. However, here we compute for each airplane **a1** the plane model and the number of planes of the same model. Consequently, the result contains duplicates. The argument of an aggregate function can be any multivalued term. Beside subqueries, we can therefore use multivalued attributes as well. For example, ***cnt*** **(a.DateOfMaintenance)** computes the number of maintenances existing for the airplane **a**, as **DateOfMaintenance** is a list–valued attribute.

The next example demonstrates the use of formulas:

Example 3.14: *"Names of airline companies which have chartered only 'new' airplanes."*

select **ac.Name**
from **ac** ***in*** **AIRLINE-COMPANY**
where forall **a** ***in*** **AIRPLANE :**
    ac chartered a ***implies*** **a.YearOfConstruction>1985**

We can see the use of the quantifier ***forall***, of the standard predicate '>', and of the relationship predicate **chartered**, the last one requiring that the airline company has chartered the airplane **a**. There are further possibilities for the construction of formulas:

- Any boolean–valued term is also a formula. In example 3.10 we have already used the term **a.Airworthy** in this way.
- Besides the standard data predicates like '=' or '>', we can also apply predicates that are defined for corresponding user–defined data types. For example, **circle-cut(c1,c2)** expresses that two circles have points in common.
- Relationship types can be used as predicates to "join" the participating entity types. However, this is done without join equations known from the relational model.
- There are some predefined predicates like ***is null*** and ***in***. The first one tests for the null value 'undefined' whereas the second one represents the element–of relation for sets, bags and lists.
- By using the logical connectives ***and***, ***or***, ***not***, ***implies***, new formulas can be built.
- Similarly, there are the well–known quantifiers ***exists*** (∃) and ***forall*** (∀), however in the form { ***exists*** | ***forall*** } <variable> ***in*** <range> : <formula>.

We allow an infix notation for binary predicates, no matter whether they are data predicates, relationships or predefined. This increases the readability of formulas. However, their usual, long prefix form can also be used, e.g., **chartered(ac,a**) or >**(YearOfConstruction,1980**).

Let us now study some more complex examples describing queries that occur quite often in the air traffic's world.

Example 3.15: *"Every direct connection from Hamburg to London"*

select **c.FlightType#, c.OccupationRate**
from **c** ***in*** **CONNECTION**
where exists **ap1** ***in*** **AIRPORT, ap2** ***in*** **AIRPORT, t1** ***in*** **TOWN, t2** ***in*** **TOWN :**
    (t1.Name='Hamburg' ***and*** **t2.Name='London'** ***and***
    t1 ***in*** **ap1.SatelliteTown** ***and*** **t2** ***in*** **ap2.SatelliteTown** ***and***
    connects-directly(c, Start:ap1, Destination:ap2))

connects-directly is again a relationship predicate. Since **AIRPORT** participates twice, we have to use the rolenames **Start** and **Destination** to make the corresponding role of an airport in the query explicit.

We now present some examples which make full use of aggregate functions to compute the number of occurences, maxima, percentages, etc.

Example 3.16: *"All non stop flights that are booked up"*

> ***select*** **nsf.Flight#**
> ***from*** **nsf *in* NON-STOP-FLIGHT**
> ***where*** ***sum*****(*select* a.NoOfSeats**
>   ***from*** **a *in* AIRPLANE**
>   ***where*** **a assigned-to nsf) = *cnt*(*select* pa**
>   ***from*** **pa *in* PASSENGER**
>   ***where*** **pa booked-for nsf)**

Example 3.17: *"The airline companies with the maximal number of agencies"*

> ***select*** **ac.Name**
> ***from*** **ac *in* AIRLINE-COMPANY**
> ***where*** ***cnt*** **(*select* ap *from* ap *in* AIRPORT *where* ac has-agency ap) =**
>   ***max*****(*cnt* (*select* ap *from* ap *in* AIRPORT**
>   ***where exists*** **ac *in* AIRLINE-COMPANY : ac has-agency ap))**

In contrast to relational SQL our query language allows functional compositions of aggregate functions. The possibility to use subqueries in the output list of a query allows a structured output of query results. Subqueries are allowed on every level of nesting. This enables an easy way of grouping data in an orthogonal way and to avoid the less powerful and complicated ***group-by*** construct known from SQL.

The purpose of this language is not only limited to the formulation of queries. We can also make use of it for the formulation of static and dynamic integrity constraints. These constraints will be discussed in the following sections.

3.4 Constraints on Database States

One important requirement for a database schema is to be capable of describing the relevant information about the real world by some database state. Our EER model provides powerful modelling constructs for capturing real world information structures so that relevant information about real world states can be mapped to database states in a natural way. But since an EER schema merely describes the structure, it does not say everything about

a) which database states correspond to possible real world states.

   So, for instance, our sample schema admits values for the **LengthOfService** attribute of **STAFF-MEMBER** exceeding the age of that **PERSON** (which is derivable from the **DateOfBirth**).

b) which possible information is actually relevant for the application.

For example, it is not clear from the sample schema which **NON-STOP-FLIGHT**s are actually stored. So it is possible to require every flight ever taking place to be stored or only those ones remaining to be scheduled in the future.

c) the way data concerning real world is mapped to a database state and vice versa. For example, the **BookingDay** of the **booked-for** relationship may refer to the day the passenger made her/his request for some flight or to the day the request was acknowledged by the airline company.

Particularly b) and c) are a great source of faults and misunderstandings because different persons might have different interpretations of a database schema. For this reason, it is particularly important for conceptual design that the specification of database structures not only consists of the definition of possible database states but also of assertions to emphasize desired states. Usually, these assertions are called *(static) integrity constraints.*

With the appeareance of database systems with more powerful, descriptive and logic oriented data models and query languages, like relational database systems, it became apparent that descriptive specification of the desired database states was necessary. It was recognised that integrity constraints can be specified within the same notational framework. Consequently, it is commonly aggreed upon that database integrity should be managed and ensured by the database management system itself [88, 89]. However, up to now there seems to be no database system providing full support of arbitrary integrity constraints. So the burden of ensuring database integrity is still left to the application programmer or the accidental database user. Nevertheless, it is well accepted that specification of integrity constraints is an important task of conceptual database design [90, 91, 92, 93, 94].

Most results on integrity constraints concern relational [95, 96, 97] and, as an offspring thereof, deductive databases [98, 99, 100, 101, 102]. Some key words in this area are *key constraints, referential integrity, cardinality requirements, type integrity,* and *redundancy.* In our framework, most of these constraint classes can directly be expressed as schema inherent constraints. Our data model directly supports primary keys. Referential integrity is captured by the concept of relationship and object–valued attributes. Type integrity is covered by the possibility of using arbitrary user–defined data types as attribute domains. Redundancy can be avoided by the concept of derived information.

But although a large class of constraints can be expressed by means of our data model we still need some mechanism for specifying further constraints. We therefore propose a language for specifying constraints is based on the query language presented in the previous section. A static constraint may be any closed formula allowed in the query language, i.e. any formula whose variables are bound by one of the variable quantifiers (**exists** or **forall**). The above requirement that the **LengthOfService** attribute of **STAFF-MEMBER** must not exceed the age of the corresponding **PERSON** is formulated as follows:

Example 3.18: *"A staff member's **LengthOfService** cannot exceed his/her age"*

forall **sm** ***in*** **STAFF-MEMBER :**
    sm.LengthOfService < (year(today) - year(sm.PERSON.DateOfBirth))

Let us remind you that there are built–in operations like **today : → date** yielding the current date and data operations like **year : date → int**. Please note, the above is a purely descriptive characterization of the desired database states. In this case it restricts the value of two attributes, namely **LengthOfService** of entity type **STAFF-MEMBER** and **DateOfBirth** of **PERSON**. Nothing is said about the way to achieve this formula being true in all database states.

The set of all integrity constraints determines the *admissible database states.* A database state is admissible if and only if all (static) integrity constraints evaluate to true in the database state. The *set of all admissible states* is denoted by Σ_c which is a subset of the *set of all possible states* Σ allowed by the pure EER schema.

In our model every attribute is optional by default, i.e., there need not exist values for the attribute. If one wants to exclude this optionality, this is done by an explicit constraint like ***forall*** **p** ***in*** **PERSON : p.Name** ***is not null*** which requires that a person's name is not allowed to be undefined. This kind of constraints also has a *graphical representation* in the diagram: solid lines stand for non–optional attributes and lines broken by a circle represent optional ones. For instance, the attribute **Tel#** is an optional attribute of **PERSON**.

Another class of explicit integrity constraints denoted in the EER diagram are *cardinalities.* Each entity type **E** participating in a relationship type **r** can be restricted by cardinality numbers (min, max), $\min \in \mathbb{N}_\circ$, $\max \in \mathbb{N} \cup \{*\}$. A concrete entity of type **E** can participate in at least 'min' and at most 'max' relationships of type **r**. The asterisk $(*)$ denotes infinitely many times, i.e., no upper bound. Some important cases quite often occur:

- $(0, *)$ is the default case meaning no restriction. A passenger, i.e. an instance of type **PASSENGER**, **has-booked-for** none, one or several non stop flights.
- $(1, *)$ is used to express mandatory memberships in relationship types. In the example every airline company must offer at least one connection, i.e. it must participate in at least one entry in the relationship type **offers**.
- $(0, 1)$ requires that an entity participates in at most one concrete relationship. For instance, every non stop flight has at most one airplane **assigned-to** it.
- Finally, a connection **belongs-to** exactly one non stop flight due to the specification $(1, 1)$.

Multivalued attributes can also have cardinalities restricting the number of elements in the set resp. list. For example, the **PlaneCrew** of a **NON-STOP-FLIGHT** must consist of at least one person. Please notice the special effect resulting from the optionality of this attribute: there may be currently no crew, i.e., the crew is "unknown"; however, if there exists a crew, it must possess at least one member.

As mentioned above, our data model involves schema inherent constraints. For instance, specifying **Name** and **DateOfBirth** as key attributes for **PERSON** is equivalent to demanding:

forall **p1, p2** ***in*** **PERSON :**
  (p1.Name=p2.Name ***and*** **p1.DateOfBirth=p2.DateOfBirth)**
  implies **p1 = p2**

The following example demonstrates that not only simple attributes can be used in constraints but also complex expressions involving data operations as well as attributes.

Example 3.19: *"A flight staff member is at most 50 years old"*

forall **fsm** ***in*** **FLIGHT-STAFF-MEMBER :**
(year(today) - year(fsm.STAFF-MEMBER.PERSON.DateOfBirth))<=50

3.5 Elementary Operations

Up to now, we have described how the static structure of a database can be modelled. Using the concepts of our EER model, an EER diagram fixes the possible structure of objects and their interrelations in a database. But, the specification of the static structure of a database by an EER diagram is only a first step in modelling a database. A database is not a dead, unchangeable read–only memory of objects, but an alive, often changing storage, where objects can be inserted, deleted or updated. It is clear that all these modifications have to regard the restrictions of the object structure specified by the EER diagram. This means more formally that each modification has to be a transition of a possible database state σ of Σ into another possible database state σ'. Such transitions of $\Sigma \times \Sigma$ can further be subdivided into *elementary transitions*, which only modify one database object and its consistent embedding in the database structure, and into *complex transitions*, which may be considered as sequences of elementary transitions.

All elementary transitions are implicitly fixed by the specification of the static structure of a database. They can automatically be derived from an EER diagram and described by so–called *elementary operations*.

These elementary operations are in turn composed of *basic operations*. Such basic operations describe the modification of exactly one database object. In contrast to elementary operations, this local modification may cause a (temporary) violation of the database structure demanded by the EER diagram. All basic operations are also implicitly given for each object or relationship type.

To summarize the different classes of update operations, we show the hierarchy of operations in figure 3.20. Example 3.21 shows the signature of some of the most basic insert resp. delete operations for our running example.

complex operations	*application specific (see subsection 4.3)*
	scope: database-state
elementary operations	*schema–dependent generated*
	scope: database-object and its consistent embedding
basic operations	*schema–dependent generated* scope: database-object

Figure 3.20: Hierarchy of database modifications

Example 3.21: For our running example, the signature of some of these basic insert resp. delete operations have the following form:

(b1) **basic-insert-entity-PERSON (Name : string, DateOfBirth : date) : PERSON**

basic-delete-entity-PERSON (p : PERSON)

(b2) **basic-insert-relship-booked-for (pa : PASSENGER,**
nsf : NON-STOP-FLIGHT) : booked-for

basic-delete-relship-booked-for (bf : booked-for)

All insert operations are functions which yield as result a modified database state, and, additionally, a reference to the inserted instance (or object) of an entity or relationship type. All key attributes of an entity type are mandatory, and, therefore, occur as formal parameters. The other attributes are optional and could be set by subsequent update operations (see below).

All delete operations as well as the insertion of a relationship instance only require references to objects as actual parameters to denote the database objects which are relevant for the execution of this operation.

In case of type constructions, already existing objects are inserted into resp. removed from the set of instances of an output entity type.

Example 3.22: Examples of such basic operations are:

(b3) **basic-insert-construct-PERSON-STAFF-MEMBER (p : PERSON) :**
STAFF-MEMBER

basic-delete-construct-STAFF-MEMBER (sm : STAFF-MEMBER)

At last, update operations are needed for the modification of attributes resp. object–valued attributes:

(b4) **basic-update-attr-PERSON.Address (p : PERSON, Address : addr)**

basic-update-attr-booked-for.PriceReduction (bf : booked-for,
pricereduction : real)

basic-update-objattr-NON-STOP-FLIGHT.Schedule
(nsf : NON-STOP-FLIGHT, tt : TIME-TABLE)

After the manipulation of a single database object by a basic operation, the new database state may not be a possible one, i.e. not a member of Σ. In this case, additional manipulations, known as *update propagations* in the literature [103], are necessary to result in a possible database state. Minimal sequences of basic operations leading to a possible database state are the *elementary operations*. It depends on the structure of the database, specified by the EER diagram, which basic operations have to be contained in an elementary operation. Let us illustrate this by some examples:

In our running example, each instance of type **PERSON** is identified by the two key attributes **Name** and **DateOfBirth**. Therefore, insertion of a person means to check whether the key attribute values are unique, and to create an object of type **PERSON** with these key values.

This insert operation becomes more complex, if an instance of a constructed entity type or of a dependent entity type has to be inserted. For example, in case of a staff member, it has to be ensured that this staff member already exists as person in the database. Otherwise, this person has to be inserted in a previous step. As **STAFF-MEMBER** is input type of a partition, it also has to be decided whether (s)he is a **GROUND-STAFF-MEMBER** or **FLIGHT-STAFF-MEMBER**.

All entity types affected by an insert operation form a subgraph of the given EER diagram. This subgraph exactly describes the scope of interest for this operation. Therefore, we call it the *propagation subgraph*. In the first example, it only consists of the entity type **PERSON**. For the insertion of a **STAFF-MEMBER** it has the shape depicted in figure 3.23.

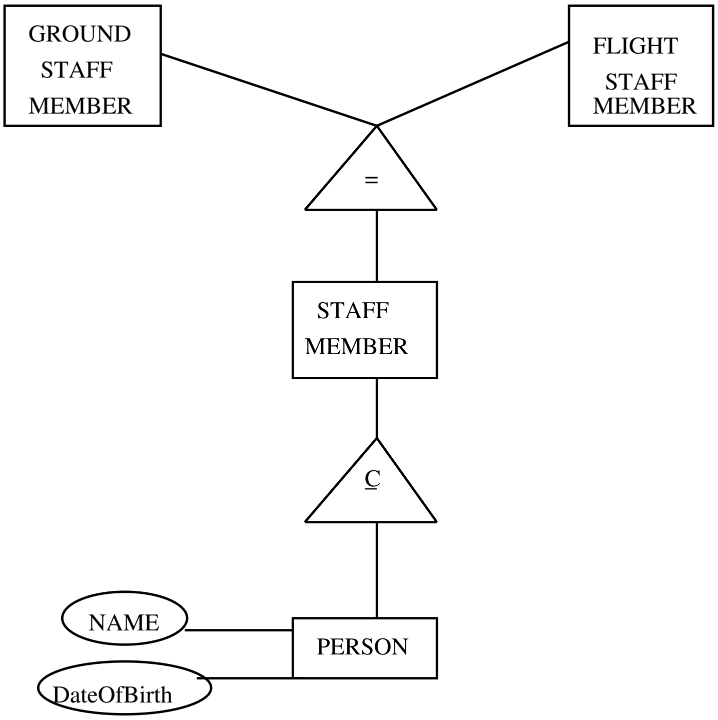

Figure 3.23: Propagation subgraph for **STAFF-MEMBER**

The parameter list of the corresponding elementary operation contains the reference to the person, to be specialized to a staff member, and the decision how to partition:

elem-insert-construct-PERSON-STAFF-MEMBER (p : PERSON,
part : (IS-GSM, IS-FSM))

Besides elementary operations, read operations are provided to access single instances of an entity or relationship type. These read operations have as parameters key attributes for the identification of an instance of an entity type and yield as result a reference to an object:

fetch-entity-PERSON (name : string, DateOfBirth : date) : PERSON

In case of dependent entity types, the insertion of an instance of a dependent type implies an update of the corresponding attribute of the "parent" type. For instance, each object of type **TIME-TABLE** belongs to a certain non stop flight. Therefore, elementary insert operations for object–valued attributes need the reference to a corresponding "parent" object as parameter:

elem-insert-objattr-TIME-TABLE (nsf : NON-STOP-FLIGHT)

The corresponding propagation subgraph, describing the scope of interest for this insertion, is depicted in figure 3.24.

The execution of this insert operations consists of the creation of a new object of type **TIME-TABLE**, and an update of the attribute **Schedule** of the corresponding **NON-STOP-**

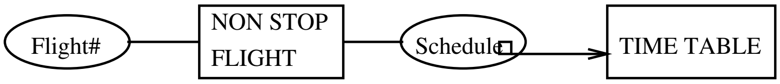

Figure 3.24: Propagation subgraph for component insertion.

FLIGHT (cf. (b4)).

The effort for the deletion of a database object is similar to the effort in case of the insertion of a database object, because the context of the instance has to be updated, too, to yield a possible database state. This means, for example, that, if this instance participates in a relationship or is the value of an object–valued attribute of another database object, these memberships or references also have to be removed. Let us illustrate elementary delete operations by two examples:

Deletion of a **NON-STOP-FLIGHT** means the deletion of an instance of this entity type together with its dependent object of type **TIME-TABLE**, and the deletion of participations of this non stop flight in the relationships **belongs-to**, **assigned-to**, and **booked-for**. The propagation subgraph for this operation can be found in figure 3.25.

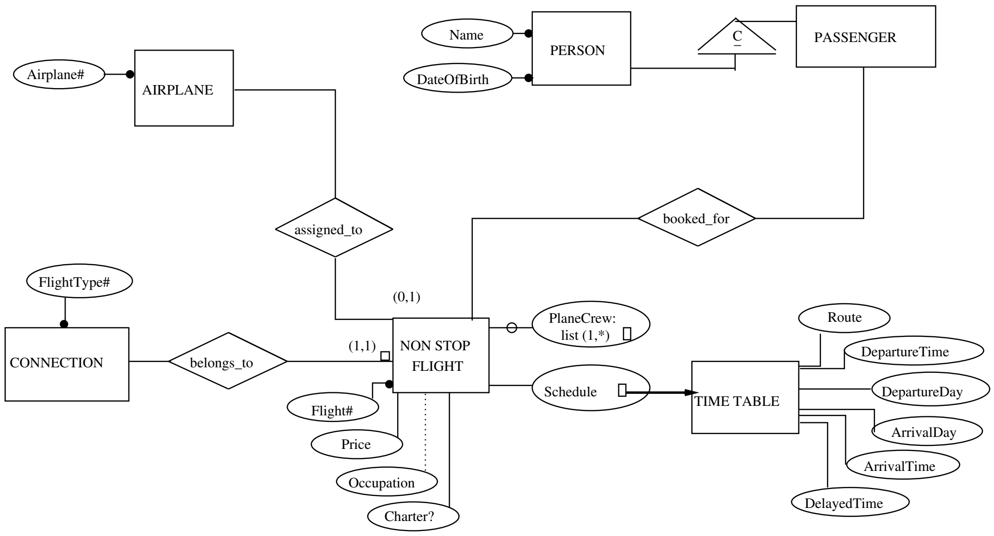

Figure 3.25: Propagation subgraph for deletion of a **NON-STOP-FLIGHT**

While the deletion of a non stop flight doesn't affect a lot of other instances in the database, the deletion of a person may have great influence on the current database state. As in real life, it depends on how active this person was which parts of the current database state have to be updated. For instance, if this person also was a ground staff member, (s)he has to be removed from the staff list of an airport, its maintenance sheds, and possibly of airline companies. The corresponding propagation subgraph in figure 3.26 contains all entity and relationship types which possibly are affected.

As mentioned in section 3.1, all data–valued non–key attributes as well as non–dependent object–valued attributes are optional. They are not part of the update propagation operations of an elementary insert operation. They have to be set by separate elementary update operations. In case of atomic entity types, these update operations have the same signature

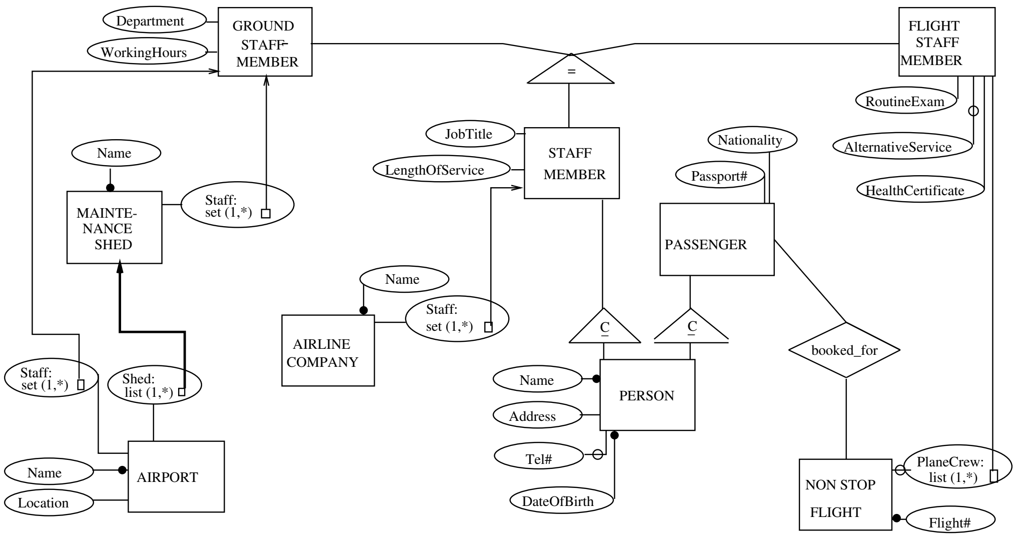

Figure 3.26: Propagation subgraph for deletion of a **PERSON**

as basic update operations (cf. (b4)). All dependent objects are only accessible by their parent objects. For example, operations to update the attribute values of a time table need the reference to the corresponding non stop flight:

elem-update-objattr-TIME-TABLE.ArrivalDay
(nsf : NON-STOP-FLIGHT,arrivday : date)

None of the elementary operations regard any cardinality constraint, which might be contained in an EER diagram. Cardinality constraints as well as other explicit integrity constraints have to be enforced during the execution of complex operations (cf. section 4.3). This procedure yields the advantage that elementary operations do not contain cycles of sequences of basic operations, which usually is the main reason for the update propagation problem discussed in the literature (cf. [103]). In our case, it is possible to determine all relevant information for the execution of an operation in advance and to deliver it as actual parameters to an elementary operation.

4 Modelling the Database – Its Dynamic Behaviour And Applications

In conceptual database design not only the information structures but also the desired database behaviour must be specified. The way a database evolves is determined by the sequence of database states $\sigma_0\sigma_1\sigma_2\sigma_3\ldots$ the database runs through, also called *evolution*. Specifying the desired database behaviour can be done in different ways. One way is to restrict the possible state sequences to admissible ones by means of so–called *temporal integrity constraints*. Temporal constraints allow to decide whether a given database sequence is admissible or not but do not say anything about how to obtain such a state sequence. Another way to specify database behaviour is to model the relevant activities changing the real world by database actions changing the database.

For conceptual database design we propose a combination of these two approaches. The distinction between temporal integrity constraints and actions allows action design to concentrate on the pure functionality of actions without taking into account whether the resulting state sequences are admissible. On the other hand admissibility can then be achieved by providing the database management system with a central integrity monitoring facility or by integrating constraints in database actions during further design steps after conceptual design.

This chapter deals with the specification of temporal integrity constraints and database manipulations. For specifying temporal constraints we follow a descriptive, logic oriented approach based on temporal logic (section 4.1). Sections 4.2 and 4.3 deal with the specification of database actions. In a similar way an EER diagram should reflect the relevant information structures of the application area, the modelled database modifications should be a reflection of the relevant *activity structures.* This requires appropriate modelling primitives for the design of database actions. On the one hand, it should be possible to model atomic real world activities in a descriptive way merely concentrating on the results of the activities than on low level database manipulations. On the other hand complex real world activities are composed of several atomic ones and hence require operational concepts for action modelling. We believe both approaches are justified in conceptual database design and present both a language for descriptive action specification based on pre/post conditions (section 4.2) and a language for operational specification (section 4.3) based on elementary operations (compare section 3.5).

4.1 Constraints on Database Evolutions

The most simple class of constraints on database evolutions are static constraints. Static constraints allow any sequence of database states which do not violate the constraints. However, in general admissibility of some state within some sequence also depends on earlier states in database evolution. We can subdivide the constraints resulting from such dependencies into transitional constraints expressing dependencies between successive states and the more general class of arbitrary temporal constraints to express dependencies on the whole history. This section mainly addresses the more general kind of constraints and discusses some relations to the other ones.

Similar to static constraints, temporal constraints have different roles in database specifications. So, we have constraints reflecting restrictions in the application area itself such as "a **FLIGHT-STAFF-MEMBER** must work for some period as a copilot before she/he can become a pilot". Other constraints arise from the intended meaning of stored data. So, for example, it is not clear from our sample EER schema whether the **DateOfMaintenance** attribute of an **AIRPLANE** refers to the previous maintenance dates or to the dates of the following maintenances. As already pointed out for static constraints, it is very important to resolve such ambiguities. In this case, we can do this, for example, by adding the temporal constraint "after a maintenance, **DateOfMaintenance** contains the date of that maintenance". Another role of temporal constraints is to assure that historical information stored in the database corresponds to the actual evolution of the database. So for example in our air traffic database the **BookingDay** of a **booked-for** relationship has to correspond to the date the booking actually took place, i.e. the date the relationship was inserted.

Some typical temporal constraints are listed in the following example.

Example 4.1:

1) The **LengthOfService** of a **GROUND-STAFF-MEMBER** may not decrease.
2) The **Miles** of an **AIRPLANE** may not decrease.
3) An **AIRPLANE** must sometime be assigned to a **NON–STOP–FLIGHT**.
4) A **NON–STOP–FLIGHT** must have an **AIRPLANE** assigned to it before it can take place.
5) An **AIRPLANE** must be maintained at least every 6 months, i.e. before the maximum date in the set **DateOfMaintenance** is more than 6 months ago.
6) An **AIRPLANE** must be maintained at least every 500000 miles, i.e. before its current **Miles** differ more than 500000 from the **Miles** value at the last maintenance.
7) During the last two weeks before the departure of a flight, a deposit of 100 DM must exist for every booking.
8) During the last week before the departure of a flight, the full price must be paid for every booking.
9) A new **PASSENGER**, i.e., a passenger booking his/her first flight, must pay the full price at the booking day.
10) An **AIRPLANE**'s **DateOfMaintenance** contains the dates of its *past* maintenances.
11) A **NON-STOP-FLIGHT** must not be deleted from the database.

There have been proposed a variety of styles for the specification of admissible database behaviour. Important directions in this area are transitional assertions [47, 48, 49] and Petri net approaches [43, 104, 5, 105]. Another important direction is to specify long term behaviour in a descriptive, logic oriented way. The work in this direction was much inspired by the approaches in [106, 107, 108] proposing modal and temporal logic for program construction and verification. To this area belong action logic and modal logic proposed in [38, 36, 37, 50], obligations/permissions [109] and our approach which is along the lines of [35, 51, 33, 52, 110, 111, 53, 54, 32, 58] using a temporal logic framework for specifying and monitoring temporal constraints.

Our language for specifying temporal constraints is a temporal extension of the language for static constraints presented in section 3.4. In addition to usual constructs to build up formulas, we have so–called temporal connectives like the nexttime operator ***next***, the unbounded temporal quantifications ***always*** and ***sometime*** as well as the bounded quantifications ***always*** ... ***before***... and ***sometime***... ***before***.... If such a formula contains free variables it must be preceded by a variable declaration part ***var*** $(\langle\text{varid}\rangle : \langle\text{type}\rangle), \ldots, (\langle\text{varid}\rangle : \langle\text{type}\rangle)$ for the free variables. Free variables are assumed to be implicitly universally quantified. We do not allow any explicit variable quantification in front of a formula containing temporal connectives.

Informally speaking, an ***always*** formula is valid for a state sequence if the subformula preceded by ***always*** is valid in every state. To be exact, the subformula must be valid in any suffix of the state sequence, i.e. in any subsequence beginning at any state. This is because the subformula in turn may be a temporal formula and hence must be evaluated on state sequences. Analogously, we can say a ***sometime*** formula is valid if the subformula preceded by ***sometime*** is valid in *some* suffix of the sequence. Similarly, validity of the other temporal connectives can be informally defined.

We should waste some words about the meaning of (implicit) universal variable quantification in the context of temporal logic. Universal quantification means the formula must become valid for all objects of the corresponding type *during their lifetime*, i.e. between their insertion and deletion. In this context, a formulation like "... the first state..." reads as "... the first state *the referenced objects exist* ..." and not "... the first state the database exists ...".

Our language for temporal constraints contains the language for static constraints as a sublanguage. But note that formulations of static constraints in this sublanguage (static formulas) have another meaning when interpreted as constraints on database evolutions. Such formulas are already valid for a state sequence iff they are valid for the first state. Formulas without any temporal operator are valid if they are valid in the first state (i.e. σ_0) of the corresponding state sequence. Static constraints however must be valid in *every* state of a state sequence. In formulations of temporal constraints, this property must explicitly be expressed in the formula. Thus a temporal logic formulation of a static constraint has the form ***always*** $\langle$ static formula $\rangle$. As an example, we formulate constraint 7) of example 4.1 in our language for temporal constraints:

Example 4.2: *"During the last two weeks before the departure of the flight a deposit of 100 DM must exist for every booking."*

var **(bf : booked–for)**
always **(daydiff(bf.NON-STOP-FLIGHT.Schedule.DepartureDay, today)< 14**
implies **bf.Account >= 100.0)**

To illustrate further temporal connectives we continue with an alternative formulation of this constraint.

Example 4.3:

var **(bf : booked-for)**
(*sometime* bf.Account $\geq$ 100.0
before **daydiff(bf.NON-STOP-FLIGHT.Schedule.DepatureDay, today)<14)**
or **(bf.Account $\geq$ 100.0)**

Note, that ***sometime*** ... ***before*** ... is *one* dyadic temporal connective. The constraint is satisfied if a deposit is paid before the deadline (14 days before the departure) or if it is paid when booking the flight (which is also possible after the deadline).

If it is sure that no passenger will ever get her/his money back, which certainly is, this formula is equivalent to the formula in example 4.2.

Example 4.3 is a nice illustration of the fact that temporal logic formulations of static constraints can occasionally be monitored more efficiently than the formulations as static constraints. For the second formulation of the booking constraint, only those entries in **booked-for** must be monitored which still don't have fulfilled the ***sometime*** condition whereas the formulation as static constraint requires all entries to be monitored.

Also transitional constraints can be expressed in this temporal framework. This can be done using the temporal operator ***next*** which relates successive database states. The temporal logic formulation of transitional constraints has the following form:

always (⟨ static formula 1 ⟩ ***implies next*** ⟨ static formula 2 ⟩)

We now illustrate the meaning of nested temporal formulas. We will do this by stepwise elaborating the temporal logic formulation of constraint 6) in example 4.1 ("An **AIRPLANE** must be maintained at least every 500000 miles"). We can reformulate this constraint as:

"whenever an airplane is released from maintenance in the next state, it must sometime have another maintenance before it will have flown another 500000 miles".

In "temporal pseudo code" this can be formulated as follows:

Example 4.4:

var **(a : AIRPLANE), (m :** ***integer*****)**
always (Airplane **a** released from maintenance in the next state with **a.Miles = m**
implies
after release another maintenance of **a** before **a.Miles > m +500000)**

One might wonder why a constraint only imposing restrictions on the behaviour of airplanes needs a second variable **m** of type ***integer***. The reason is that the current value of the attribute **a.Miles** must be compared with the value at the last maintenance. The only way to do this in temporal logic is to "bind" the old value to an additional variable and to compare in the subsequent the current value with that variable.

Filling in the unspecified parts of our constraint leads to the complete temporal logic formulation in example 4.5. We assume to have a predefined predicate **in-maintenance** for extracting from a database state whether an airplane is currently maintained.

Example 4.5:

var **(a : AIRPLANE) (m :** ***integer*****)**
always **((in-maintenance(a)** ***and*** **m = a.Miles** ***and not next*** **in-maintenance(a))**
implies
next **(*****sometime*** **in-maintenance(a)** ***before*** **a.Miles > m+500000))**

If we had designed **AIRPLANE** to contain an Attribute for **string** the mileage at the last maintenance this restriction could also be templated as a static constraint saying that the current value of **Miles** must not exceed this historical value by more than 500000 miles. In this case, however, another dynamic constraint was neccessary to ensure that the historical mileage corresponds to the mileage at the last maintenance.

Another interesting example is constraint 11) in example 4.1. It says **NON-STOP-FLIGHT**s must not be deleted. A first idea to formulate this constraint could be the formula:

var **(nsf : NON-STOP-FLIGHT)**
always **(*****exists*** **nsf'** ***in*** **NON-STOP-FLIGHT: nsf'=nsf)**

But remember that formulas are evaluated only during the lifetime of the referenced objects. Thus the above formula states, that an **nsf'** must exist as long as **nsf** exists. This is of course always true; we only need to define **nsf'** to be identical to **nsf**. However, for detecting violations of the constraint we have to check whether **nsf** is deleted. This is not possible if the **NON-STOP-FLIGHT**s are referred to in the formula itself. For giving a correct formulation of the constraint we exploit the property that object keys provide a unique, state independent object identification. The key of the object class **NON-STOP-FLIGHT** consists of the single attribute **Flight#** . State independent object identification by keys has the effect that **NON-STOP-FLIGHT**s with the same **Flight#** in different states are all the same object. For ensuring that no flight is ever deleted we only have to assure that after its insertion always a flight with the same **Flight#** exists.

Example 4.6:

"**NON-STOP-FLIGHT***s must not be deleted.*"
var (**key** : ***string***)
always ((***exists*** **nsf** ***in*** **NON-STOP-FLIGHT:**
    nsf.Flight# = key)
    implies
    always (***exists*** **nsf** ***in*** **NON-STOP-FLIGHT:**
        nsf.Flight# = key)
)

Temporal constraints can also be represented graphically by some kind of finite state automata. Such automata are called transition graphs and can be automatically derived from the temporal logic formulations of constraints. Let us illustrate this by the constraint "An airplane must sometime be maintained" which has the following temporal logic formulation: ***sometime*** **in-maintenance(a)**. During database evolution we can distinguish two different situations for an **AIRPLANE**. Either the maintenance has already taken place or it has not. An **AIRPLANE** a switches in some database state from the latter situation to the former if the non–temporal formula **in-maintenance(a)** is true. This can be generalized for arbitrary temporal logic formulas. For every temporal formula there can be distinguished a finite set of different situations each of which characterized by a temporal formula to become valid in the subsequent future and transitions between these situations characterized by non–temporal formulas. So, temporal constraints can be represented equivalently by some kind of finite state machines, so–called transition graphs, with non–temporal formulas as transition conditions [54, 58]. In fact transition graphs are the basic tool for monitoring temporal constraints at database runtime.

4.2 Descriptive Specification of Database Transactions

The dynamic evolution of a database is induced by a sequence of actions modifying the database contents. In the database area, such actions are called *transactions*. Database transactions are *integrity preserving transitions* between database states. This means, that for a given transaction **t** the semantics $[\![\mathbf{t}]\!]$ is a relation

$$[\![\mathbf{t}]\!] \subset \Sigma_c \times \Sigma_c$$

There are different paradigms to describe database state transitions in a data model. For early design steps, one can use a logic–oriented, descriptive style which describes the effects of a transaction in an abstract manner. Typically, descriptive specifications have as formal semantics the set of all functions satisfying the given abstract description. A descriptive formalism using pre- and postconditions will be presented in this section. If the database designer wants to fix one determined state transition function, s/he can use an operational style of describing transactions as presented in the next section 4.3.

Several frameworks for descriptive transaction specification are proposed in the related literature. If we look at a database state as a value of a complex data type, we can use algebraic specification of abstract data types to specify database transactions, too. In this description framework a transaction is handled as a function on complex values [112, 113, 114, 115, 116]. Besides the problem of correctly specifying functions on complex structured domains as database states typically are, this approach neglects the logic–oriented view on database states evolved for the discussion of queries and constraints.

The framework of pre- and postconditions [30, 31] states for each transaction a precondition evaluated in the current state and a postcondition which must be valid in the state resulting from the transaction execution. As proposed by [30, 31], we allow several pre- and postconditions for one transaction to structure the specification while distinguishing several situations by different preconditions. Additionally, we can give an *enabling condition* to restrict the states where the execution of the transaction is allowed.

A transaction specification has the general form:

transaction ⟨ name ⟩ (⟨parameters⟩):
[***var*** ⟨variables⟩ ;]
[***on*** ⟨enabling condition⟩ ;]
{ ***pre*** ⟨precondition⟩ ;
post ⟨postcondition⟩ ; }$^+$

where the last two syntactic categories can be repeated to have a list of ⟨precondition⟩s and ⟨postcondition⟩s for one transaction. All three conditions are formulas of the EER query language already used to describe static integrity constraints in section 3.4. The introduced ⟨variables⟩ and formal ⟨parameters⟩ can occur free in these conditions. The newly introduced variables are implicitly universally quantified for the pre- and postconditions. The ***var*** and ***on*** clauses are optional.

To illustrate the use of pre- and postconditions, we look at a few examples for descriptive transaction specifications. The first modelled transaction is the transaction **DeleteAirplane** which removes an airplane from the database.

Example 4.7:

transaction **DeleteAirplane (plane# : string):**
pre ***true***;
post ***not exists*** **a** ***in*** **AIRPLANE: a.Airplane# = plane#** ;

For the transaction **DeleteAirplane** we simply state that after the execution of the transaction there does not exist an airplane with the key value **plane#** anymore. We have no enabling condition; and as a precondition the formula ***true***. The parameter **plane#** is used inside the postcondition like a free variable of type **string**.

Please note two features (or problems) of using pre- and postconditions for transaction specification: Firstly, additional changes necessary for maintaining database integrity are not specified explicitly. For example, we may have to update the object–valued attributes and relationships involving the type **AIRPLANE** to guarantee integrity. Secondly, the transaction specification describes the desired changes only. From a logical point of view, we do not specify what happens to database elements not mentioned explicitly in the conditions.

Both effects arise due to the fact that we want to describe the desired database changes only without worrying on (desired or undesired) side effects. We have two implicit rules to handle this situation:

> The *consistency rule* is the implicit rule that each transaction has to obey the integrity constraints [31].
>
> The *frame rule* is the implicit rule that a transaction effect should be somehow minimal in terms of changed information[31, 117].

In the following, we assume that our descriptive specifications are used as verification conditions for implementing transactions only. A transaction is correct w.r.t. such a verification condition if it satisfies the given conditions – even if it additionally produces a lot of junk. Therefore, we do not worry about frame rules. For using such specifications for example also for rapid prototyping, we would have to find a mechanism to support the generation of suitable frame rules.

As an example for the use of the enabling condition, we give a refinement of the transaction specification **DeleteAirplane** additionally forbidding the deletion of an airplaine while there are still bookings for future flights with this airplane.

Example 4.8:

transaction **DeleteAirplane (plane# : string):**
on ***not exists*** **nsf** ***in*** **NON-STOP-FLIGHT:**
    (*exists* a *in* AIRPLANE: a.Airplane# = plane# ***and***
        assigned-to(a,nsf)
    and **nsf.Occupation > 0**
    and **daydiff(nsf.Schedule.Arrivalday, today) > 0)**
pre ***true***;
post ***not exists*** **a** ***in*** **AIRPLANE: a.Airplane# = plane# ;**

The introduction of explicit enabling conditions allows to distinguish two different situations appearing in transaction design. The first situation is that a transaction should have no effect in several cases, i.e. the database state remains unchanged after the transaction. This is modelled by the cases where no precondition is valid, or by a postcondition ***true*** (assuming a suitable frame rule!). The other case is that a transaction is undefined under certain circumstances which is expressed by the use of enabling conditions. Of course, that can be modelled by a postcondition ***false***, too, but we prefer to make this important case explicitly modelled by special language features.

The semantics of a transaction specification with fixed parameter values is a relation between admissible database states, i.e., it describes for a given current database state the set of transitions not violating the transaction specification. A comprehensive discussion of the semantics of pre- and postconditions as transaction specification can be found in [32, 118]. For a given state σ in Σ_c and a given substitution of the formal parameters with values, the semantics of a transaction specification can be sketched as follows:

> A transaction is *applicable* in a state σ iff there exists a substitution of the free variables with current values or objects in σ such that the enabling condition becomes valid in σ. For a given transaction $\mathbf{t}$, the subset $\Sigma_{\text{enabled}}(\mathbf{t}) \subseteq \Sigma_C$ denotes the set of states where $\mathbf{t}$ is enabled.
>
> A transition from state σ to σ' is *correct w.r.t. a transaction specification* iff for all substitutions of the free variables with current values or objects in $\sigma \in \Sigma_{\text{enabled}}(\mathbf{t})$ the validity of the precondition in σ implies the validity of the postcondition in σ'.

As mentioned before, the action specifications and the behavior specifications by temporal integrity constraints are complementary specifications of the same structures, namely the desired database state sequences. To bring both specification techniques together, we can interpret action specifications using pre- and postconditions as temporal logic specifications replacing the ***next*** operator by a set of *action–specific next operators*. This is similar to introducing action modalities like in [37, 38]. The semantics of the ***next*** operator is then equivalent to the disjunction of the action–specific next operators. These relations between the specification formalisms can be used to prove consistency and completeness of combined specifications (see [58] for first ideas in this direction).

4.3 Operational Description of Database Transactions

The preceding section presented a language following a descriptive, logic–oriented style for the specification of database transactions. The main characteristics of descriptive specifications is that they only determine the intended effect of a transaction without prescribing how this effect is achieved. Therefore, descriptive specifications are a suitable means for early design steps to describe the dynamic behaviour of a database in an abstract, application–oriented manner. But, at the end of the design process, an executable, system–oriented description of database transactions is required to yield an efficient realization of a database application system. In addition, an executable specification of database transactions facilitates a rapid prototyping of the specification and supports a database designer to compare the specified dynamic behaviour with her/his intention.

In this section, we present a language for an operational, executable style of describing database transactions $\mathbf{t}$. This enables a database designer to fix a determined state transition function $[\![\mathbf{t}]\!] : \Sigma_c \to \Sigma_c$ for each desired database transaction. This language forms a subset of a usual database programming language (cf. [40, 45]), as it focusses on database aspects, but neglects the support of the description of a sophisticated user dialogue.

It is a procedural language, containing assignment statements, procedure calls and the usual control structures like ***if-then-else***, ***while***, ***for***, ***repeat*** as main language constructs to describe the control flow. Data–valued variables can be declared to keep values of attributes or as auxiliary variables. Object–valued (single– or set–valued) variables can be introduced to store references to database objects. These (sets of) references can be retrieved from the current database by appropriate queries.

The language is based on the framework of elementary operations which automatically guarantees all model inherent integrity constraints to be fulfilled. (cf. section 3.5). They are the basic constituents of *complex actions*, i.e., procedures describing the effect of database transactions in a deterministic, executable style. As complex actions change the database only by means of elementary operations, it is guaranteed that they describe transitions from Σ into Σ. But preservation of explicit integrity constraints is not ensured.

In order to ensure that the execution of a complex action is also a transition of Σ_c into Σ_c, i.e., that all explicit integrity constraints are preserved, additional operations are needed. One possibility is to integrate appropriate code into the action specification during action design. The other possibility is to ensure integrity by means of an additional *integrity monitor*. Its task is to check after action execution whether explicit static or temporal integrity constraints have been violated. If the monitor detects a violation, the complex action is rejected and the database is rolled back to the previous state. In this case no further design activity is necessary for guaranteeing integrity.

Let us illustrate now the language by an example. It is an operational description of the transaction specified by pre–/post–condition in the previous section (example 4.8).

The example describes the deletion of an airplane, which should be forbidden if there are still bookings for future flights with this airplane:

Example 4.9:

```
transaction DeleteAirplane ( plane# :string )
  objects a : AIRPLANE;
          flights : set of NON-STOP-FLIGHT;
    begin
      a := fetch-entity-AIRPLANE ( plane# );
      if defined ( a ) then
        /* an airplane a with the number 'plane#' exists */
        flights := (select  nsf
                    from    NON-STOP-FLIGHT
                    where   assigned-to ( a, nsf ) and
                            nsf.Occupation > 0 and
                            daydiff ( nsf.Schedule.ArrivalDay, today ) > 0);
        if flights = { } then
          elem-delete-entity-AIRPLANE ( a )
        else
          error-message (" still bookings for flights with this airplane ")
        end
      else
        error-message (" airplane does not exist ")
      end
end
```

The transaction contains two object variables. The variable **a** is used to keep a reference to the airplane. It is retrieved by the read operation **fetch-entity-AIRPLANE** with the values of all key attributes as parameters. The variable **flights** is an object– and set–valued variable to store the result of an object–valued query which yields all non stop flights which are intended to use the current airplane. The elementary operation **elem-delete-entity-AIRPLANE** deletes the airplane and removes this instance from all participations within corresponding relationships or object–valued attributes. At the end of the execution of the whole complex action, it is checked whether any integrity constraint has been violated. For instance, if the airplane to be deleted is the last airplane in the **FleetOfAircraf**t of an airline company, the whole complex action is rejected because of the cardinality constraint $\mathbf{set}(1, *)$ at the attribute **FleetOfAircraft** at **AIRLINE-COMPANY**.

5 Conclusions and Future Work

The goal of this paper was to present a uniform framework for specifying all relevant aspects of a conceptual database schema, and to explain how these heterogeneous structures can be integrated. We presented a conceptual data model which is capable of tackling these tasks. We showed its pragmatics and demonstrated its appropriateness for database modelling by discussing an extensive sample application.

Database design has to deal with increasingly complex structures and must take into account different aspects of databases. Apart from modelling the static structure, database dynamics has to be modelled, too. To reduce the complexity of this modelling task, there have been made a lot of efforts for structuring the design process. In proposed database design life cycle models, conceptual design is the most demanding phase since its task is to yield the first formal description of the application. This description is formulated in some conceptual data model. In contrast to traditional approaches, our conceptual model incorporates static as well as dynamic aspects in a uniform semantical framework. Its static parts comprise the specification of data types used as domains for attributes and the specification of object structures based on an extended Entity–Relationship model.

Database dynamics is modelled by two complementary approaches. Temporal integrity constraints restrict possible database state sequences to admissible ones whereas the possible database manipulations are modelled by database (trans)actions. For the specification of temporal constraints we propose a temporal logic as specification language. For database (trans)actions we discuss two different approaches. The first one describes (trans)actions in a descriptive manner based on pre-/post-conditions. The second one allows a procedural action modelling by composing complex operations of so–called elementary ones. Elementary operations are the minimal database changes which respect all schema inherent integrity constraints. They are implicitly specified with the specification of the static part of a database and, therefore, can automatically be generated from this description of the static part.

The central link between these components is a powerful query language for the EER model. This language is the basis for specifying static and temporal integrity constraints as well as database transactions.

The presented conceptual data model is provided with a formal mathematical semantics, which has been described elsewhere [65, 58]. Here, we concentrated on pragmatic aspects of our data model.

Nevertheless, let us make some remarks on the syntactical and semantical integration of the various design components. Both the data type component and the object component define structures (types) on which certain functions and predicates are defined. For example, the data type component may contain a function **sqrt** yielding the square root of a real number or a predicate $\leq$ on integers. Similarly, the object component may contain an object type **NON-STOP-FLIGHT** with a function **Price** yieldin g the price of a flight, a function **Schedule** yielding the **TIME-TABLE** for the argument **NON-STOP-FLIGHT**, and for instance a predicate **belongs-to** which is defined on **NON-STOP-FLIGHT**s and **CONNECTION**s.

Usually, these functions and predicates are called the *database signature*. In combination with a query language they are the only way to access the contents of a database.

The syntactical difference between the two components is that functions and predicates in the data type component may only involve data types whereas a function or predicate defined in the object component may refer to object types as well as to data types. In fact, functions yielding instances of a data type as for example **Name** are the only way for an external (e.g. printable) representation of the properties of an object.

The main differences between the data type and object component are on the semantics level. The semantics of the data type component is fixed for the whole life time of the database, i.e. the sets of instances of the data types as well as the meaning of the functions and predicates is fixed for all the time.

This is not the case for the object component. On the one hand the domain of an object type may change, i.e. objects get inserted or deleted. On the other hand the meaning of the functions and predicates may change in time, e.g. the **Price** of a **NON-STOP-FLIGHT** may change. In other words, a database usually has *different states*. The instances of an object type are abstract entities with a state independent identity (they change their properties but not their identity).

The semantics of the remaining parts of a specification, i.e., static integrity constraints, the evolution component and the action component is defined in terms of the semantics of database states. Static integrity constraints are specified in form of first order formulas over the database signature. Based on the semantics of functions and predicates, the semantics of first order formulas can be defined as usual. The admissible database states are then those ones which satisfy all static integrity constraints. In the evolution component the allowed database behaviour is specified by temporal integrity constraints. Temporal constraints are formulated in a temporal logic extension of the language for static constraints. The semantical domain for our temporal logic is the set of all finite and infinite database state sequences. The semantics of temporal formulas is defined by interpreting the temporal logic operators by appropriate quantifications over the states in the state sequences.

Similarly the pre– and post–conditions characterizing the descriptive specification of database transactions can be evaluated in successive database states. Pre– and post–conditions characterize all pairs of states whose first component fulfills the pre–condition and whose second component provides the post–condition. This relation determines the valid state transitions according to a specification. Note, this view point basically relies on the fact that a database signature consists of a couple of functions and predicates and on the strict distinction between data type and object component.

More formally speaking we have the following hierarchies of logics and models:

The semantics of the datatype component is defined by an algebra [68].

> The semantics of the object component is defined by the set of all database states. A database state is an algebra containing a finite interpretation structure for the object component on the one hand and additionally the algebra for the datatype component [119, 63, 65]. The meaning of the query language SQL/EER and the language for static constraints is then given in the usual way interpreting them as formulas of a predicate calculus.
>
> The semantics of temporal logic is defined on sequences of finite and infinite database states as described formally in [120, 58]
>
> The semantics of descriptive action specifications can be found in [32]. It is defined on database state transitions, i.e., on pairs of successive database states.

The semantics of the operational action specification cannot be completely formalized in terms of the semantics at the object and data type component. Therefore in [46] an operational semantics based on graph grammars is proposed.

The conceptual data model presented in this paper was the basis for several research activities at Braunschweig Technical University. One of them is the database design environment CADDY (Computer–Aided Design of Non–Traditional Databases) [121], where a set of integrated tools for conceptual database design has been realized. The current prototype of this environment provides editing, analysing, and prototyping tools for all concepts of the described conceptual data model. In detail, there are graphical resp. textual syntax–directed editors for the design of EER diagrams, data type specifications, integrity constraints, queries and actions specifications. As a prerequisite for a rapid prototyping of the designed schema, the EER schema can automatically be transformed into a relational one. Afterwards, a prototyping database is installed on a relational database system and filled with test data. A query interpreter and a graphical database browser enable a descriptive or navigating access to the prototyping database. All these tools enable a database designer to test the designed database schema already in terms of the conceptual database schema, i.e., in early design steps. CADDY is implemented in a workstation environment under UNIX and the X window system in the programming language C. The running prototype was successfully demonstrated at several conferences and other places.

Another implementation of our EER model has been described in [122]. There, the model and the complete, original calculus [63, 65] is translated into the logic programming language Prolog. In contrast to CADDY, no emphasis has been put on a user–friendly interface. The system basically consists of a set of compilers written in Prolog which translate data specifications, schema definitions, queries, and data–manipulation statements into Prolog programs.

Further extensions of the presented approach are possible. One could think of taking over the idea of graphical design also to action or query specification. This is in fact subject to part of our current research activities.

Up to now we have investigated techniques for describing real world properties in an appropriate way by means of a conceptual model. So far we concentrated on the language issues including their semantics. Future research will have to address design methodologies, too.

Acknowledgements:

We like to thank our (former) colleagues Leonore Neugebauer, Karl Neumann and Udo Lipeck for their contributions to the presented modelling approach and the students currently or formerly working within our group.

References

[1] Bancilhon, F. ; Khoshafian, S. A Calculus for Complex Objects. Proc. 5th ACM PODS, Cambridge (Mass.) (53 – 59), 1986.

[2] Karl, S. ; Lockemann, P.C. Design of Engineering Databases: A Case for More Varied Semantic Modelling Concepts. Information Systems, Band 13, Nr. 4, (335 – 358)., 1988.

[3] Encarnaçao, J.L. ; Lockemann, P.C. (Eds.):. Engineering Databases: Connecting Islands of Automation Through Databases. Spinger–Verlag, Berlin, 1990.

[4] Dittrich, K. ; Gotthard, W. ; Lockemann, P.C. DAMOKLES – A Database System for Software Engineering Environments. In Proc. Int. Workshop Advanced programming enviroments, Conradi, R. et al. (eds.). LNCS 244, Springer Verlag, Berlin (353–371), 1987.

[5] Oberweis, A. ; Lausen, G. On the Representation of Temporal Knowledge in Office Systems. C. Rolland, M. Leonard and F. Bodard (Eds.), Proc Conf on Temporal Aspects of Information Systems (TAIS–87), North Holland (131–145), 1988.

[6] Lipeck, U.W. ; Neumann, K. Modelling and Manipulating Objects in Geoscientific Databases. Proc. 5th ERA Conf. (P.P. Chen ed.) (67 – 86), 1986.

[7] Lohmann, F. ; Neumann, K. A Geoscientific Database System Supporting Cartography and Application Programming. Proc. of the 8th British National Conference on Databases; Brown, A / Hitchcock, P. (eds.). Pitman, London , (179–195), 1990.

[8] Mayr, H.C. ; Dittrich, K.R. ; Lockemann, P.C. Datenbankentwurf. Datenbank–Handbuch (P.C. Lockemann, J.W. Schmidt, eds.). Springer Verlag, (481–557), 1987.

[9] Elmasri. R.A. ; Navathe, S.B. Fundamentals of Database Systems. Benjamin/Cummings Publ., Redwood City , 1989.

[10] Brodie, M.L. On the Development of Data Models. On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages (Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.)) (19-47), 1984.

[11] Hull, R. ; King, R. Semantic Database Modeling: Survey, Applications, and Research Issues. ACM Computing Surveys 1987, Vol. 19, No. 3 (201 – 260), 1987.

[12] Peckham, J. ; Maryansky, F. Semantic Data Models. ACM Computing Surveys , Vol. 20, No. 3 (153 – 189), 1988.

[13] Chen, P.P. The Entity–Relationship Model – Towards a Unified View of Data. ACM Transactions on Database Systems , Vo l. 1, No. 1 (9 – 36), 1976.

[14] Hammer, M. ; McLeod, D. Database Description with SDM: A Semantic Database Model. ACM Transactions on Database Systems, Vol. 6, No. 3 (351 – 386), 1981.

[15] Abiteboul, S. ; Hull, R. IFO – A Formal Semantic Database Model. ACM Transactions on Database Systems , Vol. 12, No. 4 (525 – 565), 1987.

[16] Lyngbaek, P. ; Kent, W. A Data Modeling Methodology for the Design and Implementation of Information Systems. K.R. Dittrich, U. Dayal (ed.), Proc. of the Int. Workshop on Object–Oriented Database Systems, Pacific Grove (California) (6 – 17), 1986.

[17] Mylopoulos, J. ; Wong, H.K.T. Some Features of the TAXIS Data Model. Proc. 6th VLDB, Montreal (Canada) (399 – 410), 1980.

[18] Nixon, B. (ed.):. TAXIS'84: Selected Papers. Technical Report CSRG-160 Dept. of CS, U. of Toronto, 1984.

[19] Elmasri, R.A. ; Weeldreyer, J. ; Hevner, A. The Category Concept: An Extension to the Entity–Relationship Model. Data & Knowledge Engineering , Vol. 1 (75 – 116), 1985.

[20] Makowski, J.A. ; Markowitz, V.M. ; Rotics, N. Entity–Relationship Consistency for Relational Schemes. G. Ausiello, P. Atzeni (eds.), Proc. International Conference on Database Theory ICDT, Springer LNCS 243 (306 – 322), 1986.

[21] Teorey, T.J. ; Yang, D. ; Fry, J.P. A Logical Design Methodology for Relational Databases Using the Extended Entity–Relationship Model. ACM Computing Surveys 1986, Vol. 18, No. 2 (197 – 222), 1986.

[22] Hohenstein, U. ; Neugebauer, L. ; Saake, G. ; Ehrich, H.–D. Three–Level Specification Using an Extended Entity–Relationship Model. R.R. Wagner, R. Traunmüller, H.C. Mayr (eds.), Informationsbedarfsermittlung und –analyse für den Entwurf von Informationssystemen. Informatik–Fachberichte Band 143, Springer Verlag (58 – 88), 1987.

[23] Shoshani, A. CABLE: A Language Based on the Entity–Relationship Model. Report UCID–8005, Computer Science and Applied Mathematics Department, Lawrence Berkeley Laboratory, Berkeley (California) , 1978.

[24] Poonen, G. CLEAR: A Conceptual Language for Entities and Relationships. W. Chu, P.P. Chen (eds.), Centralized and Distributed Systems. IEEE Computer Society, Silver Springs (Maryland) 1980 (194 – 215), 1978.

[25] Campbell, D.M. ; Embley, D.W. ; Czejdo, B. A Relationally Complete Query Language for an Entity–Relationship Model. Proc. 4th ERA Conf. (P.P. Chen ed.) (90 – 97), 1985.

[26] Demo, B. ; DiLeva, A. ; Giolito, P. An Entity–Relationship Query Language. A. Sernadas, J. Bubenko, A. Olive (eds.), Proc. IFIP Work. Conf. on TFAIS 1985, Sitges (Spain) (19 – 32), 1985.

[27] Elmasri, R.A. ; Wiederhold, G. Gordas: A Formal High–Level Query Language for the Entity–Relationship Model. Proc. 2nd ERA Conf. (P.P. Chen ed.) (49 – 72), 1981.

[28] Ursprung, P. ; Zehnder, C.A. HIQUEL: An Interactive Query Language to Define and Use Hierarchies. Proc. 3rd ERA Conf. (P.P. Chen ed.) (299 – 314), 1983.

[29] Zhang, Z.Q. ; Mendelzon, A.O. A Graphical Query Language for Entity–Relationship Databases. Proc. 3rd ERA Conf. (P.P. Chen ed.) (441 – 448), 1983.

[30] Veloso, P.A.S. ; Furtado, A.L. Towards Simpler and Yet Complete Formal Specifications. A. Sernadas (ed.), Proc. IFIP WG 8.1 Conference on "Theoretical and Formal Aspects of Information Systems" TFAIS 1985, (175 – 189), 1985.

[31] Lipeck, U.W. Stepwise Specification of Dynamic Database Behaviour. C. Zaniolo (ed.), Proc. International ACM SIGMOD–RECORD Conference on Management of Data 1986, Washington D.C. (387 – 397), 1986.

[32] Lipeck, U.W. Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung. Habilitationsschrift, TU Braunschweig 1988, Also: Informatik–Fachberichte 209, Springer Verlag 1989.

[33] Kung, C.H. A Temporal Framework for Database Specification and Verification. VLDB 1984 (91 – 99), 1984.

[34] Sernadas, A. ; Sernadas, C. ; Ehrich, H.–D. Object–Oriented Specification of Databases: An Algebraic Approach. P.M. Stocker, W. Kent (eds.), Proc. 10th Int. Conf. on VLDB 1987, Brighton (107 –116), 1987.

[35] Sernadas, A. Temporal Aspects of Logical Procedure Definition. Information Systems 5, 1980 (167 – 187), 1980.

[36] Golshani, F. ; Maibaum, T.S.E. ; Sadler, M.R. A Modal System for Database Specification and Query Language Support. M. Schkolnik, C. Thanos (eds.), Proc. 9th Int. Conf. on Very Large Data Bases 1983, Florence (Italy) (331 – 339), 1983.

[37] Khoshla, S. ; Maibaum, T.S.E. ; Sadler, M. Database Specification. T.B. Steel, R. Meersmann (eds.), Proc. IFIP Conference on Data Semantics DS–1, Albufeira (Portugal) (141 – 158), 1985.

[38] Fiadeiro, J. ; Sernadas, A. Specification and Verification of Database Dynamics. Acta Informatica 25, (625 – 661), 1988.

[39] Kim, W. ; Lochovsky, F.H. Object–Oriented Concepts, Databases, and Applications. ACM Press Frontier Series. Addison Wesley Publ. Reading (Mass.) , 1989.

[40] Schmidt, J.W. ; Mall, M. Pascal/R Report. Bericht Nr. 667, Fachbereich Informatik, Universität Hamburg, 1980.

[41] Schmidt, J.W. ; Eckhardt, H. ; Mall, M. DBPL–Report. DBPL–Memo 111–88, Fachbereich Informatik, Universität Hamburg , 1988.

[42] Hull, R. ; Morrison, R. ; Stemple, D. Proc. 2nd Int. Workshop on Database Programming Languages. Morgan Kaufman Publ., San Mateo , 1989.

[43] Richter, G. ; Durchholz, R. IML–inscribed Petri–Nets. Proc. IFIP Working Conference on Comparative Review of Information System Design Methodologies (T.W. Olle, A.A. Verijn–Stuart, eds.) North–Holland, Amsterdam , (335–368), 1982.

[44] Kappel, G. Schrefl, M. Object/Behaviour Diagrams. 7th Int. Conference on Data Engineering, Kobe (Japan), April (8–12) , 1991.

[45] Martin, J. Fourth Generation Languages. Vol. I. Prentice–Hall, Englewood Cliffs , 1985.

[46] Engels, G. Elementary Actions on an Extended Entity-Relationship Database. Proc. Workshop on Graph Grammars and Their Application to Computer Science, Bremen 1990. LNCS 532, Berlin, Springer Verlag 1991, (344–362) .

[47] Eswaran, K.D. ; Chamberlin, D.D. Functional Specification of a Subsystem for Data Base Integrity. VLDB 1975 (48–68), 1975.

[48] Vianu, V. Dynamic Constraints and Database Evolution. Proc. 2nd ACM SIGACT–SIGMOD Symp. on Princ. of Database Systems (Atlanta), ACM, New York 1983, 389–399, 1983.

[49] Vianu, V. Dynamic Functional Dependencies and Database Aging. Journal of the ACM 34, 1,(28–59), 1987.

[50] Wieringa, R. ; Meyer, J.–J. ; Weigand, H. Specifying Dynamic and Deontic Integrity Constraints. Data & Knowledge Engineering, Vol. 4 No. 2 , (157–191), 1989.

[51] Castilho, J.M.V. de ; Casanova, M.A. ; Furtado, A.L. A Temporal Framework for Database Specification. Proc. Int. Conference on Very Large Databases, (280–291), 1982.

[52] Ehrich, H.–D. ; Lipeck, U.W. ; Gogolla, M. Specification, Semantics and Enforcement of Dynamic Integrity Constraints. U. Dayal, G. Schlageter, L.H. Seng (eds.), Proc. 10th Int. Conf. on Very Large Data Bases, Singapore (301 – 308), 1984.

[53] Lipeck, U.W. ; Ehrich, H.–D. ; Gogolla, M. Specifying Admissibility of Dynamic Database Behaviour Using Temporal Logic. A. Sernadas (ed.), Proc. IFIP WG 8.1 Conf. Theoretical and Formal Aspects of Information Systems (145 – 157), 1985.

[54] Lipeck, U.W. ; Saake, G. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. Information Systems , Vol. 12, No. 3 (255 – 269), 1987.

[55] Schiel, U. ; Furtado, A.L. ; Neuhold, E.J. ; Casanova, M.A. Towards Multi– Level and Modular Conceptual Schema Specifications. Information Systems , Vol. 9, No. 1 (43 – 57), 1984.

[56] Carmo, J. ; Sernadas, A. A Temporal Logic Framework for a Layered Approach to Systems Specification and Verification. C. Rolland, F. Bodart, M. Levard (eds.), Proc. IFIP TC 8 WG 8.1 Working Conference on Temporal Aspects of Information Systems. Sophia–Antipolis (France) 1987 (31 – 46), 1987.

[57] Saake, G. Conceptual Modeling of Database Applications. Proc. 1st Workshop, Information Systems and Artificial Intelligence: Integration Aspects (D. Karagiannis, ed.) Ulm 1990. LNCS 474 Springer Verlag, (213–232), , 1991.

[58] Saake, G. Descriptive Specification of Database Object Behaviour. Data & Knowledge Engineering, North–Holland, Vol.6, No.1, (47–74), 1991.

[59] Smith, J.M. ; Smith, D.C.P. Database Abstractions: Aggregation and Generalization. ACM Transactions on Database Systems , Vol. 2, No. 2 (105 – 133), 1977.

[60] Brodie, M.L. ; Ridjanovic, D. On the Design and Specification of Database Transactions. On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages (Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.)) (277 – 306), 1984.

[61] Schrefl, M. ; Tjoa, A M. ; Wagner, R.R. Comparison Criteria for Semantic Data Models. Proc. International Conference on Data Engineering 1984, Los Angeles (California) (120 – 125), 1984.

[62] Hohenstein, U. ; Neugebauer, L. ; Saake, G. An Extended Entity–Relationship Model for Non–Standard Databases. Proc. "Workshop über Relationale Datenbanken" (A. Heuer, ed.). Lessach (Austria), Technical Report TU Clausthal–Zellerfeld Nr. 86–3, (185 – 211), 1986.

[63] Hohenstein, U. ; Gogolla, M. A Calculus for an Extended Entity–Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. Proc. 7th ERA Conf. (P.P. Chen ed.) (129 –148), 1988.

[64] Hohenstein, U. Ein Kalkül für ein erweitertes Entity–Relationship–Modell und seine Übersetzung in einen relationalen Kalkül. Dissertation. TU Braunschweig , 1990.

[65] Gogolla, M. ; Hohenstein, U. Towards a Semantic View of an Extended Entity–Relationship Model. ACM Transactions on Database Systems, (369 – 416), 1991.

[66] Ramm, I. ; Neumann, K. ; Lipeck, U.W. ; Ehrich, H.–D. Eine Benutzerschnittstelle für geowissenschaftliche Anwendungen. Technische Universität Braunschweig, Informatik–Berichte Nr. 85–08, 1985, 1985.

[67] Neumann, K. ; Lohmann, F. ; Ehrich, H.–D. An Experimental Geoscientific Database System. To appear in Proc. Int. Coll. on Digital Maps in Geosciences, Würzburg 1989. Geologisches Jahrbuch A122, Hannover , 1991.

[68] Ehrig, H. ; Mahr, B. Fundamentals of Algebraic Specification I – Equations and Initial Semantics. Springer, Berlin , 1985.

[69] Ehrich, H–D. ; Gogolla, M. ; Lipeck, U.W. Algebraische Spezifikation Abstrakter Datentypen – Eine Einführung in die Theorie. Teubner Verlag, Stuttgart , 1989.

[70] Cohen, B. ; Harwood, W.T. ; Jackson, M.I. The Specification of Complex Systems. Addison–Wesley (Reading) , 1986.

[71] Loeckx, J. Algorithmic Specifications: A Constructive Method for Specification of Abstract Data Types ACM TOPLAS, Vol. 9 (1987), pp. 664–685, 1987.

[72] Markowitz, V.M. ; Raz, Y. ERROL: An Entity–Relationship, Role Oriented Query Language. Proc. 3rd ERA Conf. (P.P. Chen ed.) (329 – 345), 1983.

[73] Elmasri, R.A. ; Larsen, J.A. A Graphical Query Facility for ER Databases. Proc. 4th ERA Conf. (P.P. Chen ed.) (236 – 255), 1985.

[74] Kuntz, M. ; Melchert, R. Ergonomic Schema Design and Browsing with More Semantics in the Pasta–3 Interface for E–R DBMS. ERA 89 (263 – 278), 1989.

[75] Velez, F. LAMBDA: An Entity–Relationship Based Query Language for the Retrieval of Structured Documents. Proc. 4th ERA Conf. (P.P. Chen ed.) (72 – 81), 1985.

[76] Roesner, W. DESPATH: An ER Manipulation Language. Proc. 4th ERA Conf. (P.P. Chen ed.), 1985.

[77] Schiefer, B. ; Rehm, S. Eine Anfragesprache für ein strukturell–objektorientiertes Datenmodell. T. Härder (ed.), Proc. of the GI/SI–Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft". Zürich 1989, Informatik–Fachbericht 204, Springer Verlag (373 – 388), 1989.

[78] Neumann, K. Eine geowissenschaftliche Datenbanksprache mit benutzerdefinierten geometrischen Datentypen. Dissertation, TU Braunschweig , 1988.

[79] Pistor, P. ; Andersen, F. Designing a Generalized NF Model with an SQL–Type Language Interface. Proc. 12th International Conference on Very Large Data Bases 1986, Kyoto (Japan) (278 – 285), 1986.

[80] Pistor, P. ; Traunmüller, R. A Database Language for Sets, Lists and Tables. Information Systems 1986, Vol. 11, No. 4 (323 – 336), 1986.

[81] Saake, G. ; Linnemann, V. ; Pistor, P. ; Wegener, L. Sorting, Grouping, and Duplicate Elimination in the Advanced Information Management Prototype. P.M.G. Apers, G. Wiederhold (eds.), Proc. 15 Int. Conference on Very Large Databases VLDB'89, Amsterderdam 1989 (307 – 316).

[82] Roth, M.A. ; Korth, H.F. ; Batory, D.S. SQL/NF: A Query Language for Non–1NF Relational Databases. Information Systems , Vol. 12, No. 1 (99 – 114), 1987.

[83] Date, C.J. A Critique of the SQL Database Language. Proc. International ACM SIGMOD–RECORD Conf. on Management of Data , Vol. 14, No. 3 (8 – 54), 1984.

[84] Hohenstein, U. ; Engels, G. Formal Semantics of Entity–Relationship–Based Query Language. Proc. of the 9th Int. Conf. on the Entity–Relationship Approach, Lausanne (177–194), 1990.

[85] Jaeschke, G. ; Schek, H.–J. Remarks on the Algebra of Non First Normal Form Relations. Proc. 1st ACM Symposium on Principles of Database Systems 1982, Los Angeles (California) (124 – 138), 1982.

[86] Schek, H.–J. ; Scholl, M.H. The Relational Model with Relation–Valued Attributes. Information Systems 1986, Vol. 11, No. 2 (137 – 147), 1986.

[87] Date, C.J. The SQL Standard. Addison–Wesley, Reading (Massachusetts) , 1987.

[88] Stonebraker, M. Implementation of Integrity Constraints and Views by Query Modification. SIGMOD 1975 (65–78), 1975.

[89] Astrahan, M.M. ; et al. System R: A Relational Approach to Database Management. TODS 1 (97–137), 1976.

[90] Bernstein, P.A. ; Blaustein, B.T. ; Clarke, E.M. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. VLDB (126–137), 1980.

[91] Cremers, A.B. ; Domann, G. AIM, An Integrity Monitor for the Database System INGRES. VLDB (167–170), 1983.

[92] Bertino, E. ; Apuzzo, D. Integrity Aspects in Data Base Management Sytems. Proc. IEEE Trends and Applications Conference, Making Database Work, (43–52), 1984.

[93] Abiteboul, S. ; Vianu, V. Transactions and Integrity Constraints. Proc. ACM Principles of Database Systems , (193 – 204), 1985.

[94] Qian, X. ; Wiederhold, G. Knowledge–based Integrity Constraint Validation. VLDB (417–425), 1986.

[95] Hammer, M.M. ; McLeod, D.J. Semantic Integrity in a Relational Database System. VLDB (25–47), 1975.

[96] Nicolas, J.–M. Logic for Improving Integrity Checking in Relational Data Bases. Acta Informatica, 18, (227–253), 1982.

[97] Weber, W. ; Stucky, W. ; Karszt, J. Integrity Checking in Deductive Database Systems. Information Systems 8 (125–136), 1983.

[98] Nicolas, J.–M. ; Yasdanian, K. Improving Integrity Checking in Deductive Databases. Logic and Databases, H. Gallaire / J. Minker (eds.), Plenum Press, New York (325–344), 1978.

[99] Bry, F. ; Manthey, R. Checking Consistency of Database Constraints: A Logical Basis. VLDB (13–20), 1986.

[100] Decker, H. Integrity Enforcement on Deductive Databases. Proc. 1st Int. Conference on Expert Systems, L.Kerschberg (ed.) (271–285), 1986.

[101] Kowalski, R. ; Sadri, F. ; Soper, P. Integrity Checking in Deductive Databases. Proc. 19th Int. Conference VLDB, (61–69), 1987.

[102] Lloyd, J.W. ; Sonnenberg, E.A. ; Topor, R.W. Integrity Constraint Checking in Stratified Databases. Journal of Logic Programming 4 (331–344), 1987.

[103] Scheuermann, P. ; Schiffner, G. ; Weber, H. Abstraction Capabilities and Invariant Properties Modelling in the Entity–Relationship Approach. Proc. 1st Int. Conf. on the Entity–Relationship Approach (P.P. Chen ed.) (121 – 140), 1979.

[104] Lausen, G. ; Németh, F. ; Oberweis, A. ; Schönthaler, F. ; Stucky, W. The INCOME Approach for Conceptual Modelling and Prototyping of Information Systems. Proc. First Nordic Conf. on Advanced Systems Engineering (CASE 89), 1989.

[105] Eder, J. ; Kappel, G. ; Tjoa, A M. ; Wagner, R.R. BIER: The Behaviour Integrated Entity Relationship Approach. Proc. 5th int. Conf. on the Entity–Relationship Approach (P.P. Chen ed.) (147 – 166), 1986.

[106] Manna, Z. ; Pnueli, A. The Modal Logics of Programs. 6th Colloquium on Automata, Languages and Programming, H.A. Maurer (ed.), LNCS 71, Springer Verlag, Berlin 1979, (385–409), 1979.

[107] Manna, Z. ; Pnueli, A. Verification of Concurrent Programs: The Temporal Framework. The Correctness Problem in Computer Science (R.S.Boyer et al., eds.). Academic Press London 1981, (215–273), 1981.

[108] Manna, Z. ; Wolper, P. Synthesis of Communicating Processes from Temporal Logic Specifications. ACM Vol.6, 68–93., 1984.

[109] Fiadeiro, J. ; Sernadas, C. ; Maibaum, T. ; Saake, G. Proof-Theoretic Semantics of Object-Oriented Specification Constructs. North-Holland, Amsterdam, 1990.

[110] Hülsmann, K. ; Saake, G. Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring. Acta Informatica 28, (365–407) , 1991.

[111] Hülsmann, K. ; Saake, G. Representation of the Historical Information Necessary for Temporal Integrity Monitoring. Proc. 2nd Int. Conference Extending Database Technology, Venice , 1990.

[112] Ehrig, H. ; Kreowski, H.J. ; Weber, H. Algebraic Specification Schemes for Database Systems. S.B. Yao (ed.), Proc. 4th VLDB, West–Berlin (427 – 440), 1978.

[113] Casanova, M.A. ; Veloso, P.A.S. ; Furtado, A.L. Formal Database Specification – An Eclectic Perspective. PODS 1984, (110 – 118), 1984.

[114] Stemple, D. ; Sheard, T. Database Theory for Supporting Specification–Based Database System Development. Proc. 8th Int. Conference on Software Engineering. 1985 (43 – 49), 1986.

[115] Furtado, A.L. ; Neuhold, E.J. Formal Techniques for Data Base Design. Springer Verlag, Berlin , 1986.

[116] Qian, X. ; Waldinger, R. A Transaction Logic for Database Specifications. Proc. SIGMOD . (243 – 250), 1988.

[117] Brown, F.M. (ed.). The Frame Problem in Artificial Intelligence. Proc. of the 1987 Workshop, Los Altos, Morgan Kaufman Publ. , 1987.

[118] Lipeck, U.W. Transformation of Dynamic Intergity Constraints into Transactiv specifications. Proc. 2nd. Conf. on Database Theory (M. Gyssens et al., eds.) LNCS 326, Springer Verlag 1988,(322 – 337).

[119] Ehrich, H.–D. ; Drosten, K. ; Gogolla, M. Towards an Algebraic Semantics for Database Specification. R.A. Meersman, A.C. Sernadas (eds.), Proc. IFIP 2.6 Work. Conference on Database Semantics 'Knowledge & Data' (DS–2), Albufeira (Portugal) 1986 (119 – 135), 1986.

[120] Lipeck, U.W. Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung. Habilitationsschrift, TU Braunschweig 1988, Also: Informatik–Fachberichte 209, Springer Verlag 1989.

[121] Engels, G. ; Hohenstein, U. ; Hülsmann, K. ; Löhr–Richter, P. ; Ehrich, H.–D. CADDY: Computer–Aided Design of Non–Standard Databases. N. Madhavji, H. Weber, W. Schäfer (eds.), Int. Conference on System Development Environments & Factories. Berlin, May 1989.

[122] Gogolla, M.,Meyer, B. ; Westerman, G.D. Drafting Extended Entity–Relationship Schemas with QUEER. Proc. 10th Int. Conf. on the Entity–Relationship Approach, San Mateo (CA), 1991.

A Data-valued Attributes of Entities and Relationships

NON-STOP-FLIGHT	Flight#	string	key
(nsf)	Charter?	bool	
	Price	real+	
	Occupation	nat	derived
	(No. of extensions of "booked-for" in which the nsf object takes part)		

TIME-TABLE	DepartureTime	time
(tt)	ArrivalTime	time
	DepartureDay	date
	ArrivalDay	date
	Route	line

CONNECTION	Flighttype#	string	key
(c)	NoOfExtensions	nat	derived
		(No. of extensions of c in "belongs-to")	
	Occupationrate	real+	derived
		(Σ Occupation of a connection per year)	
		(Σ NoOfSeats per assigned airplane)	

AIRPLANE	Airplane#	string	key
(a)	AirplaneModel	string	
	CrewSize	nat	
	NoOfSeats	nat	
	YearOfConstruction	year	
	DateOfMaintenance	list(date)	
	Miles	real+	
	Airworthy	bool	

AIRLINE-COMPANY	Name	string	key
(ac)	Trademark	graphic	
	HeadOffice	string	

AIRPORT	Name	string	key
(ap)	Location	point	key
	Hotel	set(string)	optional
	NoOfPassengers	nat	
	NoOfArrivals	nat	
	Kind	{ domestic, continental, intercontinental }	
TOWN	Name	string	key
(t)	Location	circle	
	Country	string	
	NoOfInhabitants	nat	

PERSON	Name	string	key
(p)	Address	addr	
	DateOfBirth	date	key
	Tel#	string	optional

PASSENGER	Passport#	string
(pa)	Nationality	list(string)

STAFFMEMBER	JobTitle	string
(sm)	LengthOfService	nat

FLIGHT-STAFF-MEMBER	RoutineExamination	date	
(fsm)	HealthCertificate	bool	
	AlternativeService	string	optional

GROUND-STAFF-MEMBER	Department	string
(gsm)	WorkingHours	nat

MAINTENANCE-SHED	Name	string	key
(ms)	MaxCapacity	nat	
	Cost	real+	

has-agency	NoOfStaff	nat
shuttle-service	Timetable	table
booked-for	BookingDay	date
	PriceReduction	real+
	Account	real+

B Complete EER–Diagram