4 The Enhanced Entity-Relationship Model

Bernhard Thalheim
Department of Computer Science, Christian-Albrechts University Kiel, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

4.1 Database Design

4.1.1 Database Design and Development

The problem of information system design can be stated as follows:
Design the logical and physical structure of an information system in a given database management system (or for a database paradigm), so that it contains all the information required by the user and required for the efficient behavior of the whole information system for all users. Furthermore, specify the database application processes and the user interaction.

The implicit goals of database design are:

- to meet all the information (contextual) requirements of the entire spectrum of users in a given application area;
- to provide a "natural" and easy-to-understand structuring of the information content;

0-8493-8597-0/01/$0.00+$1.50
© 2007 by CRC Press, LLC

- to preserve the designers' entire semantic information for a later redesign;
- to meet all the processing requirements and also achieve a high degree of efficiency in processing;
- to achieve logical independence of query and transaction formulation on this level;
- to provide a simple and easy-to-comprehend user interface family.

These requirements must be related to database models. We distinguish between relational database models, hierarchical and other graphical database models, object-oriented database models, and object-relational models. The enhanced ER model discussed in this chapter is the sole database model that supports mappings to all kinds of database models.

Over the past few years database structures have been discussed extensively. Some of the open questions have been satisfactorily solved. Modeling includes, however, additional aspects:

Structuring of a database application is concerned with representing the database structure and the corresponding static integrity constraints.

Functionality of a database application is specified on the basis of processes and dynamic integrity constraints.

Distribution of information system components is specified through explicit specification of services and exchange frames that specify the architecture and the collaboration among components.

Interactivity is provided by the system on the basis of predefined stories for a number of envisioned actors and is based on media objects which are used to deliver the content of the database to users or to receive new content.

This understanding has led to the **co-design approach** to modeling by specification **structuring**, **functionality**, **distribution**, and **interactivity**. These four aspects of modeling have both syntactic and semantic elements.

There are numerous many database specification languages. Most of them are called database models. The resulting specification is called a database schema. Simple database models are the relational model and the entity-relationship model. In this chapter we introduce extensions of the entity-relationship model which are consistent, easy to use and have a translation to relational, object-relational and XML database models. Specification of databases is based on three interleaved and dependent parts:

Syntactics (syntax): Inductive specification of databases is based on a set of base types, functions and predicates, a collection of constructors and a theory of construction limiting the application of constructors by rules or by formulas in deontic logics. In most cases, the theory may be simplified to the usage of canonical rules for construction and interpretation. Structural recursion is the main specification vehicle.

Semantics: Specification of admissible databases on the basis of static and dynamic integrity constraints describes those database states and those database sequences which are considered to be legal. If structural recursion is used then a variant of hierarchical first-order predicate logic may be used for the description of integrity constraints.

Pragmatics: Description of context and intension is based either on explicit reference to the enterprize model, to enterprize tasks, to enterprize policy and environments or on intensional logics used for relating the interpretation and meaning to users depending on time, location, and common sense.

Specification is often restricted to the description of syntactical elements. This restricted approach can only be used for simple applications with simple structuring. However, most applications require analysis of data, integration or federation of data, advanced aggregation of data, and advanced basic data types.

4.1.2 Implicit Assumptions and Inherent Constraints of Database Specification Languages

Each language used should be based on a clear definition of structure, semantics, operations, behavior and environment. At the same time, languages presuppose implicit assumptions and constraints. The enhanced or extended ER (EER) model might, for instance, use the following assumptions:

Set semantics: The default semantics of entity and relationship types are set semantics. If extended type constructors are used then their semantics are explicitly defined.

Identifiability: Each entity type is identifiable. Each component type needs to be labeled whenever it cannot be distinguished from other components. In relationship types components are ordered. Their labels can be omitted whenever there is an identification. Set semantics implies identifiability of any element in the database.

Partial Unique Name Assumption: Attribute names are unique for each entity and relationship type. Entity type names and relationship type names are unique for the ER-schema.

Referential Integrity: If a type is based on component types then each value for this type can only use such values in components which *exist* as values in the component instance.

Monotonicity of Semantics: If integrity constraints Φ are added to a given set of integrity constraints Σ, then the set of possible instances which satisfy the extended set of constraints $\Sigma \cup \Phi$ is a subset of the set of instances which satisfy Σ.

We are not using the entity integrity assumption. The database can use null values in keys as well [?]. The entity integrity assumption can be enforced by the profile used during mapping schema specifications to logical database languages.

4.1.3 Storage and Representation Alternatives

The classical approach to objects is to store an object based on strong typing. Each real-life thing is thus represented by a number of objects which are either coupled by the object identifier or by specific maintenance procedures. This approach has led to the variety of types. Thus, we might consider two different approaches:

Class-wise, strongly identification-based representation and storage: Things of reality may be represented by several objects. Such choice increases maintenance costs. For this reason, we couple things under consideration and objects in the database by an injective association. Since we may be not able to identify things by their value in the database due to the complexity of the identification mechanism in real life we introduce the notion of the *object identifier* (OID) in order to cope with identification without representing the complex real-life identification. Objects can be elements of several classes. In the early days of object-orientation it was assumed that objects belonged to one and only one class. This assumption has led to a number of migration problems which do not have any satisfactory solution. The association among facets of the same thing that are represented by several objects is maintained by the object identifier.

Object-wise representation and storage: Graph-based models which have been developed in order to simplify the object-oriented approaches [?] display objects by their subgraphs, i.e. by the set of nodes associated to a certain object and the corresponding edges. This representation corresponds to the representation used in standardization.

Object-wise storage has a high redundancy which must be maintained by the system thus decreasing performance to a significant extent. Aside from performance problems such systems also suffer from low scalability and poor utilization of resources. The operating of such systems leads to lock avalanches. Any modification of data requires a recursive lock of related objects.

Therefore, objects-wise storage is applicable only under a number of restrictions:

- The application is stable and the data structures and the supporting basic functions necessary for the application do not change during the lifespan of the system.
- The data set is almost free of updates. Updates, insertions and deletions of data are only allowed in well-defined restricted 'zones' of the database.

A typical application area for object-wise storage is archiving or information presentation systems. Both systems have an update system underneath. We call such systems play-out systems. The data are stored in the way in which they are transferred to the user. The data modification system has a play-out generator that materializes all views necessary for the play-out system.

Two implementation alternatives are already in use albeit more on an intuitive basis:

Object-oriented approaches: Objects are decomposed into a set of related objects. Their association is maintained on the basis of OID's or other explicit referencing mechanisms. The decomposed objects are stored in corresponding classes.

XML-based approaches: The XML description allows null values to be used without notification. If a value for an object does not exist, is not known, is not applicable or cannot be obtained etc. the XML schema does not use the tag corresponding to the attribute or the component. Classes are hidden. Thus, we have two storage alternatives for XML approaches which might be used at the same time or might be used separately:

Class-separated snowflake representation: An object is stored in several classes. Each class has a partial view on the entire object. This view is associated with the structure of the class.

Full-object representation: All data associated with the object are compiled into one object. The associations among the components of objects with other objects are based on pointers or references.

We may use the first representation for our storage engine and the second representation for out input engine and our output engine in data warehouse approaches. The input of an object leads to a generation of a new OID and to a bulk insert into several classes. The output is based on views.

The first representation leads to an object-relational storage approach which is based on the ER schema. Thus, we may apply translation techniques developed for ER schemata[?].

The second representation is very useful if we want to represent an object with all its facets. For instance, an *Address* object may be presented with all its data, e.g., the geographical information, the contact information, the acquisition information etc. Another *Address* object is only instantiated by the geographical information. A third one has only contact information. We could represent these three object by XML files on the same DTD or XSchema.

4.1.4 The Higher-Order Entity-Relationship Model (HERM)

The entity-relationship model has been extended by more than sixty proposals in the past. Some of the extensions contradict other extensions. Within this chapter we use the higher-order (or hierarchical) entity relationship model (HERM). It is a special case of an extended entity-relationship model (EER) e.g. [**?**, **?**, **?**, **?**].

The higher-order ER model used in this chapter has the following basic and extended modeling constructs:

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

Complex attributes: Using basic types, complex attributes can be defined by means of the tuple and the set constructors. The tuple constructor is used to define complex attributes by Cartesian aggregation. The set constructor allow construction of a new complex attribute by set aggregation. Additionally, the bag, the list, and the variant constructors can be used.

Entities: Entity types are characterized by their attributes. Entity types have a set of attributes which serve to identify the elements of the class of the type. This concept is similar to the concept of key known for relational databases.

Clusters: A disjoint union $\dot{\cup}$ of types whose identification type is domain compatible is called a cluster. Cluster types (or variant types) are well known in programming languages, but are often overlooked in database models, where this absence creates needless fragmentation of the databases, confusing mixing of generalization and specialization and confusion over null values.

First-order relationships: First-order relationship types are defined as associations between single entity types or clusters of entity types. They can also be characterized by attributes.

Higher-order relationships: The relationship type of order i is defined as an association of relationship types of order less than i or of entity types and can also be characterized by attributes.

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

Operations: Operations can be defined for each type.

- The generic operations `insert, delete`, and `update` are defined for each type.
- The algebra consists of classical set operations, such as union, intersection, difference and restricted complement, and general type operations, such as selection, map (particular examples of this operation are (tagged) nest, unnest, projection, renaming), and pump (particular examples of this operation are the classical aggregation functions). The fixpoint operator is not used.

- Each type can have a set of (conditional) operations.
- Based on the algebra, query forms and transactions can be specified.

The extensions of the ER model should be safe in the sense that appropriate semantics exist. There is a large variety of proposals which are not safe. Some reasons for this include higher-order or function types, such as those used for the definition of derived attributes, or the loss of identification.
It can be observed that higher-order functions can be attached to the type system. However, in this case types do not specify sets, although their semantics can be defined by topoi [**?**, **?**]. This possibility limits simplicity for the introduction of constraints and operations. Furthermore, these semantics are far too complex to be a candidate for semantics. The ER model is simpler than OO models.

4.2 Syntax of EER Models

4.2.1 Structuring Specification

We use a classical four-layered approach to inductive specification of database structures. The first layer is the data environment, called the basic data type scheme, which is defined by the system or is the assumed set of available basic data. The second layer is the schema of a given database. The third layer is the database itself representing a state of the application's data often called micro-data. The fourth layer consists of the macro-data that are generated from the micro-data by application of view queries to the micro-data.

The second layer is diversely treated by database modeling languages. Nevertheless, there are common features, especially type constructors. A common approach in most models is the generic definition of operations according to the structure of the type. The inductive specification of structuring is based on base types and type constructors.

A type constructor is a function from types to a new type. The constructor can be supplemented

- with a *selector* for retrieval (like *Select*) with a retrieval expression and *update functions* (like *Insert*, *Delete*, and *Update*) for value mapping from the new type to the component types or to the new type,
- with *correctness criteria* and rules for validation,
- with *default* rules, e.g. CurrentDate for data assignment,
- with one or several *user representations*, and
- with a *physical representation* or properties of the physical representation.

A **base type** is an algebraic structure $B = (Dom(B), Op(B), Pred(B))$ with a name, a set of values in a domain, a set of operations and a set of predicates. A class B^C on the base type is a collection of elements from $Dom(B)$. Usually, B^C is required to be a set. It can be also a list (denoted by $< . >$), multi-set ($\{|.|\}$), tree etc. Classes may be changed by applying operations. Elements of a class may be classified by the predicates.

The value set can be discrete or continuous, finite or infinite. We typically assume discrete value sets. Typical predicates are comparison predicates such as $<, >, \leq, \neq, \geq, =$. Typical functions are arithmetic functions such as $+, -$, and $\times$.

The set of integers is given by the IntegerSet, e.g. integers within a 4-byte representation and basic operations and predicates:

integer := (*IntegerSet*, $\{0, s, +, -, *, \div\}$, $\{=, \leq\}$) .

The base type is extended to a data type by explicit definition of properties of the underlying value sets:

***Precision and accuracy*:** Data can be precise to a certain extent. Precision is the degree of refinement in the calculations. Accuracy is a measure of how repeatable the assignment of values for properties is.

***Granularity*:** Scales can be fine or coarse. The accuracy of data depends on the granularity of the domain which has been chosen for the representation of properties.

***Ordering*:** The ordering of values of a given domain can be based on ordering schemes such as lexicographic, geographic or chronological ordering or on exact ordering such as orderings on natural numbers. The ordering can also be based on ontologies or categories. Scales have a range with lowest values and highest values. These values can be finite or infinite. If they are finite then overflow or underflow errors might be the result of a computation.

***Classification*:** The data can be used for representation of classifications. The classification can be linear, hierarchical, etc. The classification can be mono-hierarchical or poly-hierarchical, mono-dimensional or poly-dimensional, analytical or synthetical, monotetical or polytetical. The classification can be based on ontologies and can be maintained with thesauri.

***Presentation*:** The data type can be mapped to different representation types dependent on several parameters. For instance, in Web applications, the format chosen for presentation types of pictures depends on the capacity of the channel, on the compression etc. The presentation might be linear or hierarchical and it can be layered.

***Implementation*:** The implementation type of the attribute type depends on the properties of the DBMS. The implementation type also influences the complexity of computations.

***Default values*:** During the design of databases, default values can be assigned in order to store properties regarding the existence of data such as 'exists but not at the moment', 'exist but not known', 'exists but under change', 'at the moment forbidden/system defined/wrong', 'not reachable', 'until now not reachable', 'not entered yet', 'not transferable/transferred', 'not applicable to the object'. Usually, only one default value is allowed. An example of a specific default value is the *null value*.

***Casting functions*:** We assume that type systems are (strongly) typed. In this case we are not able to compare values from different domains and to compute new values from a set of values taken from different domains. Casting functions can be used to map the values of a given domain to values of another domain.

It should be noted [**?**, **?**] that the data type restricts the operations which can be applied. Databases often store *units of measure* which use a scale of some sort. *Scales* can be classified [**?**] according to a set of properties such as the following: a natural origin point of scale represented usually by a meaningful 'zero' which is not just a numeric zero; applicability of meaningful operations that can be performed on the units; existence of natural orderings of the units; existence of a natural metric function on the units. Metric functions obey triangular property, are symmetric and map identical objects to the scales origin. For instance, adding weights is meaningful whereas adding shoe sizes looks odd. The plus operation can be different if a natural ordering exists. Metric values are often relative values which are perceived in different ways, e.g., the intensity of light.

Typical kinds of data types are compared in the following table:

TABLE 4.1 Data types and their main canonical assumptions

Kind of data type	Natural order	Natural zero	Predefined functions	*Example*
extension based				
absolute	+	+/-	+/-	*number of boxes*
ratio	+	+/-	+(type dependent)	*length, weight*
intension based				
nominal	-	-	(-) (except concatenation)	*names of cities*
ordinal	+	-	-	*preferences*
rank	+	+	-	*competitions*
interval	+	-	(+)(e.g., concatenation)	*time, space*

We, thus, extend basic data types to extended data types by description Υ of precision and accuracy, granularity, order, classification, presentation, implementation, special values, null, default values, casting functions, and scale.

This extended specification approach avoids the pitfalls of aggregation. Aggregation functions can be applied to absolute and ratio values without restriction. Additive aggregation and min/max functions can be applied to interval values. The average function can only be applied to equidistant interval values. The application of aggregation functions such as summarization and average to derived values is based on conversions to absolute or ratio values. Comparison functions such as min/max functions can be applied to derived values only by attribution to ratio or absolute values. The average function can only be applied to equidistant interval values. Aggregation functions are usually not applicable to nominal values, ordinal values, and rank values.

Given a set of (extended) base types $\mathcal{T}$ and a set of names U, a **data scheme** $DD = (U, \mathcal{T}, dom)$ is given by a finite set U of *type names*, by a set $\mathcal{T}$ of extended base types, and by a *domain function* $dom : U \to \mathcal{T}$ that assigns to each base type name its "domain" type.

We denote DD by $\{A_1 :: dom(A_1), ..., A_m :: dom(A_m)\}$ in the case that the set of type names $U = \{A_1, ..., A_m\}$ of the data scheme is finite. The A_i are called **atomic attributes** or basic attributes.

Given additionally a set of names NA different from U and a set of labels L distinct from NA and U, we inductively introduce the set UN of complex attribute types or shortly **complex attributes**:

- Any atomic attribute is a complex attribute, i.e. $U \subseteq UN$.
- If $X \in UN$ then $l : X \in UN$ for $l \in L$ (labelled attribute).
- If $X \in UN$ then $[X] \in UN$ (optional attribute).
- If $X_1, ..., X_n \in UN$ and $X \in NA$ then $X(X_1, ..., X_n)$ is a (tuple-valued) complex attribute in UN. This attribute type can also be used in the notion X.
- If $X' \in UN$ and $X \in NA$ then $X\{X'\}$ is a (set-valued) complex attribute in UN.
- No other elements are in UN.

The set L is used as an additional naming language. Each attribute can be labeled by names or labels from L. Labels can be used as alias names for attributes. They are useful for shortening expressions of complex attributes. They can carry a meaning but do not carry semantics. They can be omitted whenever it is not confusing.

Additionally, other type constructors can be used for defining complex attributes:

- lists of values, e.g., $< X >$,
- vectors or arrays of values, e.g., $X_{Min}^{Max}(Y)$ with an index attribute Y, and minimal and maximal index values and
- bags of values, e.g., $\{\!|X|\!\}$.

For reasons of simplicity we restrict the model to tuple and to set constructors. However, list and bag constructors can be used whenever type constructors are allowed.

Typical examples of complex attributes are

Name : (FirstNames<(FirstName,use)>, FamName, [NameOfBirth,]
Title:$\{AcadTitle\} \dot\cup$ *FamTitle)*
Contact : (Phone($\{AtWork\}$*, private), email, ...)*
DateOfBirth :: date
AcadTitel :: titleType
PostalAddress : (Zip, City, Street, HouseNumber)
for *dom(Zip) = String7, dom(City) = VarString, dom(Street) = VarString,*
dom(HouseNumber) = SmallInt.

Now we can extend the function *dom* to *Dom* on UN.

- $Dom(\lambda) = \emptyset$ for the empty word λ.
- For $A \in U$, $Dom(A) = dom(A)$.
- For $l : X \in UN$, $Dom(l : X) = Dom(X)$.
- For $[X] \in UN$, $Dom([X] = Dom(X) \cup \lambda$.
- For $X(X_1, ..., X_n) \in UN$, $Dom(X) = Dom(X_1) \times ... \times Dom(X_n)$ where $M_1 \times ... \times M_n$ denotes the Cartesian product of the sets $M_1, ..., M_n$.
- For $X\{X'\} \in UN$, $Dom(X\{X'\}) = Pow(Dom(X))$ where $Pow(M)$ denotes the powerset of the set M.

Two attribute types X, Y are called *domain-compatible* if $Dom(X) = Dom(Y)$.

For the data scheme DD the set D_{DD} denotes the union of all sets $Dom(X)$ for $X \in UN$.

The subattribute A of B is inductively defined in the same manner (denoted by $A \preceq B$). A is a non-proper subattribute of A but not a proper subattribute of A. $X(X_{i_1}, ..., X_{i_m})$ and $X(X_1', ..., X_n')$ are subattributes of $X(X_1, ..., X_n)$ for subattributes X_j' of X_j $(1 \leq j \leq j)$. $X\{X''\}$ is a subattribute of $X\{X'\}$ for a subattribute X'' of X'.

A **tuple** (or object) o on $X \subseteq UN$ and on $DD = (U, \underline{D}, dom)$ is a function

$$o : X \longrightarrow D_{DD} \qquad \text{with } t(A) \in Dom(A) \text{ for } A \in X \ .$$

An **entity type** E is defined by a triple $(attr(E), id(E), \Sigma)$, where

- E is an entity set name,
- $attr(E)$ is a set of attributes,
- $id(E)$ is a non-empty generalized subset of $attr(E)$ called the key or identifier, and
- Σ is a set of integrity constraints.

We shall use the sign $\stackrel{\circ}{=}$ for an assignment of a name to a construct, e.g. $E \stackrel{\circ}{=} (attr(E), id(E), \Sigma)$. Trivial elements may be omitted. The set $attr(E)$ of all attributes is the trivial identifier $id(E)$. The empty set $\emptyset$ of constraints is a trivial set of integrity constraints. Let us assume for the moment that $\Sigma = \emptyset$. We shall introduce integrity constraints below.

The following types are examples of entity types:

Person $\stackrel{\circ}{=}$ *({ Name, Address, Contact, DateOfBirth,* *<u>PersNo</u>: EmplNo* $\dot\cup$ *... , ... })*
Course $\stackrel{\circ}{=}$ *({ CourseNumber, CName } , { CNumber }),*
Room $\stackrel{\circ}{=}$ *({Building, RoomNumber } , {Building, RoomNumber }),*
Department $\stackrel{\circ}{=}$*({ DName, Director, Phones { Phone } } , { DName }),*
Semester $\stackrel{\circ}{=}$ *({ Year, Season }, { Year, Season }).*

The notion of entity types can be extended to entity types with key sets:

$E \doteq (attr(E), \{id_j(E) \mid 1 \leq j \leq m\})$ with m keys.

We assume that attributes are unique in an entity type. Therefore, we can omit the set brackets. Identifiers may be given by underlining the corresponding attributes.

Entities or objects o of E can be now defined as tuples on $attr(E)$.

An entity of the type *Person* is for instance the object

β: ((<(Karl,SecondChristian),(Bernhard,Christian)>, Thalheim,
{Prof., Dr.rer.nat.habil., Dipl.-Math.}),
CAU Kiel, (({ +49 431 8804472, +49 431 8804054}, _),
thalheim@is.informatik.uni-kiel.de), 10.3.1952, 637861, ...).

At any fixed moment in time t an **entity class** E^C for the entity type E is a set of objects o on $attr(E)$

- for which $id(E)$ is a key, i.e., the inequality $o_{id(E)} \neq o'_{id(E)}$ is valid for any two different objects or tuples o, o' from E^C, and
- the set of integrity constraints Σ is valid.

In some cases, (entity) types may be combined into a union of types or so called **cluster types**. Since we need to preserve identification we restrict the union operation to disjoint unions. Clusters based on entity types can be defined by the disjoint union of types. Furthermore, we require that the identification types of the components of a cluster are domain-compatible. Take now the set of types $\{R_1, ..., R_k\}$ as given.

These types can be clustered by a "category" or a *cluster type*

$C \doteq l_1 : R_1 + l_2 : R_2 + ... + l_k : R_k$.

Labels can be omitted if the types can be distinguished.

Examples of cluster types are the following types

$$JuristicalPerson \doteq Person \,\dot\cup\, Company \,\dot\cup\, Association\ ,$$

$$Group \doteq Senate \,\dot\cup\, WorkingGroup \,\dot\cup\, Association\ .$$

For a cluster type $C \doteq l_1 : R_1 + l_2 : R_2 + ... + l_k : R_k$ we can similarly define the **cluster class** C^C as the 'disjoint' union of the sets $R_1^C, ..., R_k^C$. If $R_1, ..., R_k$ are entity types then C is a cluster of entity types. The cluster is defined if $R_1^C, ...R_k^C$ are disjoint.

Entity types $E_1, ..., E_k$ are now given. A **(first-order) relationship type** has the form $R \doteq (ent(R), attr(R), \Sigma)$ where

- R is the name of the type,
- $ent(R) = l_1 : R_1, ..., l_n : R_n$ is a sequence of (labelled) entity types and of clusters of these,
- $attr(R) = \{B_1, ..., B_k\}$ is a set of attributes from UN, and
- Σ is a set of integrity constraints.

First-order relationship types that have only one entity type are called *unary*, those with two entity types are called *binary* and those with several labelled occurrences of the same entity type are called *recursive*. Labels can be omitted if the types can be distinguished.

Example of a first-order relationship types are the types

InGroup $\doteq$ *(Person, Group, { Time(From [,To]), Position }),*
DirectPrerequisite $\doteq$ *(hasPrerequisite : Course, isPrerequisite : Course),*
Professor $\doteq$ *(Person, { Specialization }),*
Student $\doteq$ *(Person, { StudNo }),*
Enroll = (Student, CourseHeld, { Result }),
Major $\doteq$ *(Student, Program, ∅),*
Minor $\doteq$ *(Student, Program, ∅).*

If each R_i is an entity type then a first-order relationship set R^C is a set of relationships, i. e., $R^C \subseteq R_1^C \times ... \times R_n^C \times dom(B_1) \times ... \times dom(B_k)$.
An assumption that is typically made for representation is that relationships use only the identification part of the objects which participate in the relationship.

If R_i is a cluster $R_{i,1} + ... + R_{i,k}$ then the relationships in R^C are distinct according to R_i, i.e., for $r, r' \in R^C$ either $r.R_i \neq r'.R_i$ and $R_{i,j}^C \cap R_{i,j'}^C$ for distinct j, j' or $r.R_i$ determines the component $R_{i,j}$ of R_i, i.e. $r.R_i \in R_{i,j}^C \setminus \cup_{j' \neq j} R_{i,j'}^C$.
The latter disjointness can be weakened by labels.

We may generalize the notion of first-order relationship types to relationship types of arbitrary order. Given now entity types $E_1, ...E_k$ and relationship and cluster types $R_1, .., R_l$ of orders not higher than i for $i > 0$, an **(i+1)-order relationship type** has the form
$R \doteq (compon(R), attr(R), \Sigma)$ where

- R is the name of the type,
- $compon(R)$ is a sequence of (labelled) entity and relationship types or clusters from $\{E_1, ..., E_k, R_1, ..., R_l\}$, and
- $attr(R) = \{B_1, ..., B_k\}$ is a set of attributes from UN, and
- Σ is a set of integrity constraints.

We may also use constructors $\times, \cup, \dot{\cup}, \{.\}, \{\!|.|\!\}, < . >$ (Cartesian product, union, disjoint union, powerset, bags (multisets), list) for definition of complex components.
The disjointness for clusters can be weakened for relationship types.

Examples of a higher-order relationship types are the types
Has $\doteq$ *(Project, PrimaryInvestigator:Professor + Member:Person ,* $\emptyset$ *),*
Supervisor $\doteq$ *(Student, Professor, { Since }),* .

Higher-order types allow a convenient description of classes that are based on other classes. Let us consider a course planning application. Lectures are courses given by a professor within a semester and for a number of programs. Proposed courses extend lectures by description of who has made the proposal, who is responsible for the course, which room is requested and which time proposals and restrictions are made. Planing of courses assigns a room to a course that has been proposed and assigns a time frame for scheduling. The kind of the course may be changed. Courses that are held are based on courses planned. The room may be changed for a course. We use the following types for the specification:
ProposedCourse $\doteq$ *(Teacher: Professor, Course, Proposal : Kind, Request : Room, Semester, Set4 : { Program }, Responsible4Course: Person, InsertedBy : Person, { Time(Proposal, SideCondition) }),*
PlannedCourse $\doteq$ *(ProposedCourse, [Kind] , [Room], { TimeFrame }),*
CourseHeld $\doteq$ *(PlannedCourse, [Room]).*
The last two types use optional components in the case that a proposal or a planning of rooms or kinds is changed.

Given now a relationship type $R \doteq (R_1, ..., R_n, \{B_1, ..., B_k\})$ and classes $R_1^C, ..., R_n^C$. A **relationship** r is an element of the Cartesian product
$R_1^C \times ... \times R_n^C \times Dom(B_1) \times ... \times Dom(B_k)$.
The **relationship class** R^C is a set of relationships, i. e.
$R^C \subseteq R_1^C \times ... \times R_n^C \times Dom(B_1) \times ... \times Dom(B_k)$.
If R_i is a cluster $R_{i,1} + ... + R_{i,k}$ then the relationships in R^C are distinct according to R_i, i.e., for $r, r' \in R^C$ either $r.R_i \neq r'.R_i$ and $R_{i,j}^C \cap R_{i,j'}^C$ for distinct j, j', or $r.R_i$ determines the component $R_{i,j}$ of R_i, i.e., $r.R_i \in R_{i,j}^C \setminus \bigcup_{j' \neq j} R_{i,j'}^C$.

The last disjointness condition can be weakened by labels. If we use extended relationship types then identifying subcomponents can be used instead of the full representation.

A set $\{E_1, ...E_n, R_1, ..., R_m\}$ of entity, cluster and (higher-order) relationship types on a data scheme DD is called complete if any relationship types use the types from $\{E_1, ...E_n, R_1, ..., R_m\}$ as components. A complete set of types is also called **EER schema** $\mathcal{S}$. The EER schema is going to be extended by constraints. The EER schema is defined by the pair $(\mathcal{S}, \Sigma)$.

We can represent a complete set of entity, cluster and relationship types by ER diagrams. One possible kind of diagram is displayed in Figure 4.1. Entity types are represented graphically by rectangles. Attribute types are associated with the corresponding type. Relationship vertices are represented graphically by diamonds. Clusters are represented by diamonds labelled with root illustrated $\bigoplus$ or simply as a common input point to a diamond.

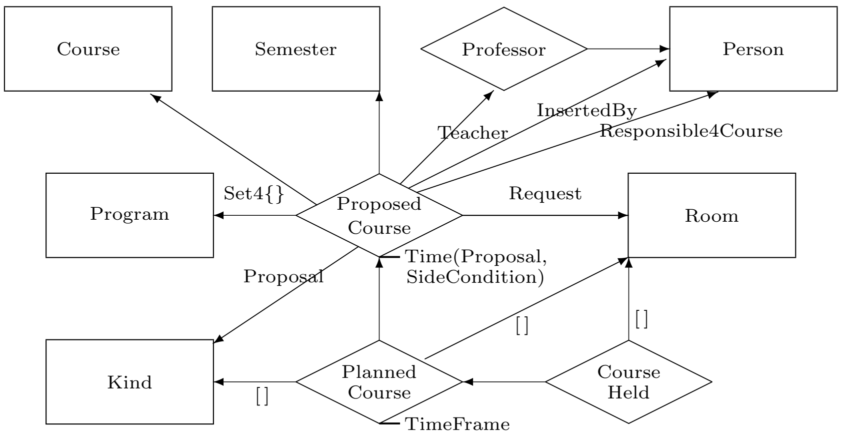

FIGURE 4.1: A Sample HERM Diagram With Higher-Order Relationship Types

This style of drawing diagrams is one of many variants that have been considered in the literature. The main difference of representation is the style of drawing unary types. Three different styles are depicted in Figure 4.2. We prefer the compact style in the left diagram.

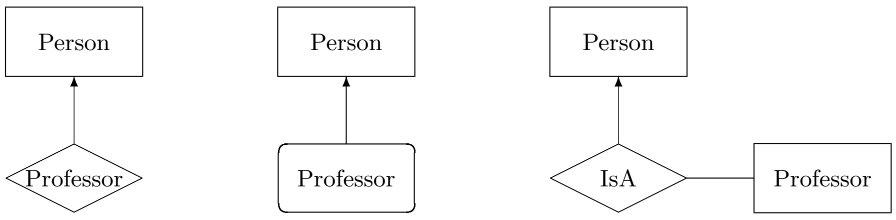

FIGURE 4.2: Variants for Representation of Unary Relationship Types

The notion of subattributes can be extended to substructures of an EER structure in a similar form. Substructures use the components of a type. Given two substructures X and Y of a type T, the generalized union $X \sqcup_T Y$ is the smallest substructure of T that has both X and Y as its substructures. The generalized intersection $X \sqcap_T Y$ is the largest substructure of both X and Y.

We have introduced essentials of extended entity-relationship models. The entity-relationship model has attracted a lot of research. A large variety of **extensions** have been proposed. We conclude this subsection by a brief introduction of main extensions.

We do not use so-called **weak types** that use associations among a schema for definition of the the identification of a type. We refer the reader to [**?**] for a discussion on the pitfalls of these types.

A number of domain-specific extensions have been introduced to the ER model. One of the most important extension is the extension of the base types by **spatial data types** [**?**] such as: point, line, oriented line, surface, complex surface, oriented surface, line bunch, and surface bunch. These types are supported by a large variety of functions such as: meets, intersects, overlaps, contains, adjacent, planar operations, and a variety of equality predicates.

We distinguish between **specialization types** and **generalization types**. Specialization is used whenever objects obtain more specific properties, may play a variety of roles, and use more specific functions. Typically, specialization is specified through *IS-A* associations. Specific relationship types that are used for description of specialization are the following types: *Is-Specific-Kind-Of*, *Is-Role-Of*, *Is-Subobject-Of*, *Is-Subset-Of*, *Is-Property-Of*, *Has-Effect-Of*, *Has-Function-Of*, *Is-Partial-Synonym-Of*, *Entails*, and *Part-Of*. Functions are typically inherited downwards, i.e., from the supertype to the subtype.

Generalization is used for types of objects that have the same behavior. The behavior and some of the components may be *shifted* to the generalization if they are common for all types that are generalized. Typical generalizations are *Is-Kind-Of*, *Considered-As*, and *Acts-In-Role*. Generalization tends to be an abstraction in which a more general (generic) type is defined by extracting common properties of one or more types while suppressing the differences between them. These types are subtypes of the generic type. Thus, generalization introduces the Role-Of relationship or the Is-A relationship between a subtype entity and its generic type. Therefore, the constructs are different. For generalization the generic type must be the union of its subtypes. Thus, the subtypes can be virtually clustered by the generic type. This is the case for specialization. The specialization of a role of the subtype can be changed. This is the case for generalization.

Generalization is usually defined through a cluster type. The cluster type can be translated to a relational type or to a relational view. Functions are inherited upwards, i.e., from the type to the abstraction. Abstractions typically do not have their own functions.

The distinction into specialization and generalization may be based on an introduction of *kernel types*. These types are either specialized or generalized and form a 'center' of the hierarchy. A pragmatical rule for detection of such types is based on independent existence of objects. Those object sets that are existing (relatively) independent from other object sets are candidates for kernel sets. They form an *existence ridge* within the schema.

Hierarchy abstraction allows considering objects in a variety of levels of detail. Hierarchy abstraction is based on a specific form of the general join operator [**?**]. It combines types which are of high adhesion and which are mainly modeled on the basis of star sub-schemata. *Specialization* is a well-known form of hierarchy abstraction. For instance, an *Address* type is specialized to the *GeographicAddress* type. Other forms are *role hierarchies* and *category hierarchies*. For instance, *Student* is a role of *Person*. *Undergraduate* is a category of *Student*. The behavior of both is the same. Specific properties have been changed.

Variations and *versions* can be modeled on the basis of hierarchy abstraction.

Hierarchies may be combined and the root types of the hierarchies are generalized to a common root type. The combination may lead to a graph which is not a tree but a forest, i.e., an acyclic graph with one root. The variation of the root type is formed by a number of dimensions applicable to the type. For instance, addresses have specialization dimension, a language dimension, an applicability dimension and a classification dimension.

Schemata may have a number of **dimensions**. We observe that types in a database schema are of very different usage. This usage can be made explicit. Extraction of this utilization pattern shows that each schema has a number of internal dimensions : *Specialization dimension* based on roles objects play or on categories into which objects are separated; *association dimension* through bridging related types and in adding meta-characterization on data quality; *usage, meta-characterization or log dimension* characterizing log information such as the history of database evolution, the association to business steps and rules, and the actual usage; *data quality, lifespan and history dimension*. We may abstract from the last two dimensions during database schema development and add these dimensions as the last step of conceptual modeling. In this case, the schema which is considered until this step is concerned with the main facets of the application.

The *metadata* dimension describes the databases data. Metadata guide utilization of evaluation functions. The management of metadata is based on a number of steps:

Creation of meaningful metadata is based on a proper characterization of the data by their history, their association to the documents from which the data have been taken, and an evaluation of the quality of the source. Metadata must be appropriate to their use within the model suite. At the same time they must be adequate. Meaningful metadata help to understand the data that are available and provide information how these data can be used.

Maturing metadata can be based on adding the context of the data and the metadata, e.g., by a characterization of time and integrity. Any change in the data must be reflected by changes in the metadata. This change management is based on explicit maintenance of data heritage with respect to the real world at the time the data was initially captured and stored and on data lineage depending on the path taken by the data from initial capture to its present location and how it was altered along that path.

Management of metadata becomes important whenever the redundancy of data increases, e.g. in data warehouses, in files of data that are reusing, aggregating and combining their data from other or own data. Simple metadata management has been built into data dictionaries, is used in data modelling tools, etc. Extended metadata management has been incorporated into data repositories. Nowadays metadata are becoming an essential source of information and a supporting facility for sophisticated search. Metadata management supports sophisticated storage, concurrent and secured access, and ease of use with browsing, searching, indexing, integration and association facilities.

Maintenance of metadata must be automated, includes quality assurance, reprocessing, uses intelligent versioning, configuration and combination, and has refreshment cycles.

Migration and sharing of metadata becomes crucial if data source are kept distributed, are heterogeneous, if they are shared among different kinds of usages, if interoperability of different data sources is necessary, and if portability requires a change due to different operational use while maintaining old applications.

Schemata may have a metastructure and may consist of several **components** [?]. These components may be interrelated and may intersect. Some of them are completely independent of interrelated through specific associations (connector types). This meta-structure is captured by the *skeleton* of the schema. This skeleton consists of the main modules without capturing the details within the types. The skeleton structure allows to separate parts of the schema from others. The skeleton displays the structure at a large. At the same time, schemata have an internal *meta-structure*.

Component-based ER modelling does not start with the singleton type. First, a skeleton of components is developed. This skeleton can be refined during the evolution of the schema. Then, each component is developed step by step. Each refinement leads to a component database schema. If this component is associated to another component then its development must be associated with the development of the other component as long as their common elements are concerned.

Temporality is described for extended ER models in a variety of ways.

- Data may be temporal and depend directly on one or more aspects of time. We distinguish three orthogonal concepts of time: temporal data types such as instants, intervals or periods, kinds of time, and temporal statements such as current (now), sequenced (at each instant of time) and non-sequenced (ignoring time). Kinds of time are: *existence time*, *lifespan time*, *transaction time*, *change time*, *user-defined time*, *validity time*, and *availability time*. The first two kinds of time are not considered in databases since they are integrated into modelling decisions. Temporal data are supported by specific temporal functions. These functions generalize Allen's time logic [?].

- The database schema may be temporal as well. The evolution covers the aspect of changes in the schema. The best approach to handle evolution is the separation of parts of the schema into those types that are stable and into those types of the schema that are changing. The change schema is a meta-schema on those components that evolve.

- Database systems have to support different temporal viewpoints and temporal scopes of users. Therefore, the database schema has a temporal representation dimension. Typical large systems such as SAP R/3 support this multi-view concept by providing views on the same data. These views are updateable and are equivalent to each other. Another extension of ER model which supports multi-scoping is the explicit introduction of multiple representations of each type. The types are enhanced by a controller that supports the generation of the appropriate view on the type.

4.2.2 Functionality Specification

The higher-order entity-relationship model uses an inductive structuring. This inductive structuring can also be used for the introduction of the functionality. Functionality specification is based on the HERM-algebra and can easily be extended to HERM/QBE, query forms and transactions. This framework for functionality specification supports introduction of some kinds of dynamic integrity constraints and consideration of behavior. The greatest consistent specification (GCS) approach is used for integrity enforcement instead of rule triggering pitfalls. Another advantage of this approach is that interactivity may be introduced in integrated form based on dialogue scenes, cooperation, messages, and scenarios [?]. The translation portfolio may be used for translation and for compilation of functionality to object-relational, relational and semi-structured languages.

The EER algebra uses type-preserving functions and type-creating functions. Simple type-preserving functions generalize the classical set-theoretic functions. Let R^{C_1} and R^{C_2} be classes over the same type R.
The *union* $R^{C_1} \cup R^{C_2}$ is the standard set union of R^{C_1} and R^{C_2}.
The *difference* is $R^{C_1} \setminus R^{C_2}$ is the standard set difference of R^C and R^{C_2}.
The *intersection* $R^{C_1} \cap R^{C_2}$ is the set intersection of R^{C_1} and R^{C_2}.
Clearly, the types of the union, the intersection and the difference are T.

Type-creation functions on type systems can be defined by *structural recursion* [?, ?, ?]. Given types T, T' and a collection type C^T on T (e.g. set of values of type T, bags, lists) and operations such as generalized union $\cup_{C^T}$, generalized intersection $\cap_{C^T}$, and generalized empty elements $\emptyset_{C^T}$ on C^T. Given further an element h_0 on T' and two functions defined on the types $h_1 \; : \; T \to T'$ and $h_2 \; : \; T' \times T' \to T'$, we define the structural recursion by insert presentation for R^C on T as follows:
$srec_{h_0,h_1,h_2}(\emptyset_{C^T}) \; = \; h_0$
$srec_{h_0,h_1,h_2}(\{\!|s|\!\}) \; = \; h_1(s)$ for singleton collections $\{\!|s|\!\}$ consisting of an object s
$srec_{h_0,h_1,h_2}(\{\!|s|\!\} \cup_{C^T} R^C) \; = \; h_2(h_1(s), srec_{h_0,h_1,h_2}(R^C))$ iff $\{\!|s|\!\} \cap_{C^T} R^C = \emptyset_{C^T}$.

All operations of the relational database model and of other declarative database models can be defined by structural recursion.
Given a selection condition α. *Selection* is defined by $\sigma_\alpha = srec_{\emptyset,\iota_\alpha,\cup}$ for the function

$$\iota_\alpha(\{o\}) \quad = \quad \begin{cases} \{o\} & \text{if } \{o\} \models \alpha \\ \emptyset & \text{otherwise} \end{cases}$$

and the type $T' = C^T$.
Selection is a type-preserving function.
Projection is defined by $\pi_X = T[X] = srec_{\emptyset,\pi_X,\cup}$ for the subtype X of T, the function

$$\pi_X(\{o\}) \quad = \quad \{o|_X\}$$

that restricts any object to the components of X and the type $T' = C^X$.
(Natural) Join is defined by $\bowtie = srec_{\emptyset,\bowtie_T,\cup}$ for the type $T = T_1 \times T_2$, the function

$$\bowtie_T(\{(o_1, o_2)\}) \quad = \quad \{o \in Dom(T_1 \cup T_2) \,|\, o|_{T_1} = o_1 \wedge o|_{T_2} = o_2\}$$

and the type $T' = C^{T_1 \cup T_2}$.
The natural join is equal to the Cartesian product of the intersection of T_1 and T_2 is empty and is equal to the intersection if T_1 and T_2 coincide.
The *Cartesian product* $R^C \times S^C$ is a class of the scheme $T = R \circ S$ and equals

$$\{t \in Dom(T) | t[R] \in R^C \wedge t[S] \in S^C\} \quad .$$

The concatenation of types is denoted by $\circ$.
Renaming is defined by $\rho_{X,Y} = srec_{\emptyset,\rho_{X,Y},\cup}$ for the subtype X of T and a type Y with $Dom(X) = Dom(Y)$ and for the function

$$\rho_{X,Y}(\{(o)\}) \quad = \quad \{o' \in Dom((T \setminus X) \circ Y) \,|\, o|_{T \setminus X} = o'|_{T \setminus X} \wedge o|_X = o'|_Y\}$$

and the type $T' = C^{(T \setminus X) \circ Y}$.
Nesting is defined by $\nu_X = srec_{\emptyset,\rho_{X,\{X\}},h_2}$ for the subtype X of $T = R$,
for the type $T' = C^{(R \setminus X) \sqcup_R \{X\}}$, and for the function

$$h_2(\{o'\}, T'^C) \quad = \quad \{o'\} \cup T'^C \qquad \text{if } \; o'|_X \notin \pi_X(T'^C)$$

$$\begin{aligned} h_2(\{o'\}, T'^C) \quad = \quad \{\, o \in Dom(T') \,|\, \exists o' \in T'^C : o|_{R \setminus X} = o'|_{R \setminus X} \\ \wedge\, o(X) \; = \; \{\, o''[X] \,|\, o'' \in T'^C \wedge o'|_{R \setminus X} = o''|_{R \setminus X}\}\} \end{aligned}$$

in the case that $o'|_X \in \pi_X(T'^C)$.
Unnesting is defined by $\mu_X = srec_{\emptyset,\rho_{X,\{X\}},h_2}$ for the set subtype $\{X\}$ of $T = R$,
for the type $T' = C^{(R \setminus \{X\}) \circ X}$, and for the function

$$\begin{aligned} h_2(\{o'\}, T'^C) \quad = \quad \{o'\} \cup \\ \{\, o \in Dom(T') \,|\, \exists o'' \in R^C : o[R \setminus \{X\}] = o''[R \setminus \{X\}] \wedge o|_X \in o''|_X\} \end{aligned}$$

We distinguish aggregation functions according to their computational complexity:

- The simplest class of aggregation functions use simple (one-pass) aggregation. A typical example are the simple statistical functions of SQL: *count (absolute frequency), average (arithmetic mean), sum (total), min, max.*
- More complex aggregation functions are used in cumulative or moving statistics which relate data subsets to other subsets or supersets, e.g. growth rates, changes in an aggregate value over time or any dimension set (banded reports, control break reports, OLAP dimensions). Typical examples are queries like:
  "What percentage does each customer contribute to total sales?"
  "Total sales in each territory, ordered from high to low!"
  "Total amount of sales broken down by salesman within territories".

Aggregation functions distinguish between normal values and null values. We use two functions for null values

$$h_f^0(s) \quad = \quad \begin{cases} 0 & \text{if } s = \text{NULL} \\ f(s) & \text{if } s \neq \text{NULL} \end{cases}$$

$$h_f^{\text{undef}}(s) \quad = \quad \begin{cases} \text{undef} & \text{if } s = \text{NULL} \\ f(s) & \text{if } s \neq \text{NULL} \end{cases}$$

Then we can introduce main aggregation through structural recursion as follows:

- *Summarization* is defined by $\texttt{sum}_0^{null} = \quad srec_{0,h_{Id}^0,+}$ or $\texttt{sum}_{undef}^{null} = \quad srec_{0,h_{Id}^{\text{undef}},+}$.
- The *counting* or *cardinality* function counts the number of objects:
  $\texttt{count}_1^{null} = \quad srec_{0,h_1^0,+}$ or $\texttt{count}_{undef}^{null} = \quad srec_{0,h_1^{\text{undef}},+}$.
- *Maximum* and *minimum* functions are defined by
  - $\texttt{max}_{\text{NULL}} = srec_{\text{NULL},Id,max}$ or $\texttt{min}_{\text{NULL}} = \quad srec_{\text{NULL},Id,min}$
  - $\texttt{max}_{\text{undef}} = srec_{\text{undef},Id,max}$ or $\texttt{min}_{\text{undef}} = \quad srec_{\text{undef},Id,min}$
- The *arithmetic average* functions can be defined by

  $$\frac{\texttt{sum}}{\texttt{count}} \quad \text{or} \quad \frac{\texttt{sum}_0^{null}}{\texttt{count}_1^{null}} \quad \text{or} \quad \frac{\texttt{sum}_{undef}^{null}}{\texttt{count}_{undef}^{null}} \quad .$$

  SQL uses the following doubtful definition for the average function

  $$\frac{\texttt{sum}_0^{null}}{\texttt{count}_1^{null}} \quad .$$

One can distinguish between distributive, algebraic and holistic aggregation functions:

Distributive or inductive functions are defined by structural recursion. They preserve partitions of sets, i.e. given a set X and a partition $X = X_1 \cup X_2 \cup ... \cup X_n$ of X into pairwise disjoint subsets. Then for a distributive function f there exists a function g such that $f(X) = g(f(X_1), ..., f(X_n))$. Functions such as `count, sum, min, max` are distributive.

Algebraic functions can be expressed by finite algebraic expressions defined over distributive functions. Typical examples of algebraic functions in database languages are `average` and `covariance`. The average function for instance can be defined on the basis of an expression on `count` and `sum`.

Holistic functions are all other functions. For holistic functions there is no bound on the size of the storage needed to describe a sub-aggregate. Typical examples are `mostFrequent, rank` and `median` . Usually, their implementation and expression in database languages require tricky programming. Holistic functions are computable over temporal views.

We use these functions for definition of derived elementary modification functions:

Insertion of objects: The *insert*-function `Insert(`R^C, o`)` is the union $R^C \cup \{o\}$ for classes R^C and objects o of the same type R.

Deletion/removal of objects: The *delete*-function `Delete(`R^C, o`)` is defined through the difference $R^C \setminus \{o\}$ for classes R^C and objects o of the same type R.

Update of objects: The modification `Update(`R^C, α, γ`)` of classes R^C uses logical conditions α and replacement families $\gamma = \{(o, R^{C_o})\}$ that specify which objects are to be replaced by which object sets. The *update*-function `Update(`R^C, α, γ`)` is defined through the set

$$R^C \setminus \sigma_\alpha(R^C) \cup \bigcup_{o \in \sigma_\alpha(R^C)} R^{C_o} \quad .$$

We notice that this definition is different from the expression $R^C \setminus \sigma_\alpha(R^C) \cup R^{C'}$ that is often used by DBMS instead of update operations, e.g. by `Delete(`R^C, o`);` `InsertUpdate(`R^C, o'`)` . If for instance $\sigma_\alpha(R^C) = \emptyset$ and $R^{C'} \neq \emptyset$ then the effect of the update operation is lost.

Structural recursion on collection types together with the canonical operations provide us with a powerful programming paradigm for database structures. If collections are restricted to 'flat' relations then they express precisely those queries that are definable in the relational algebra. By adding a fixed set of aggregate operations such as *sum, count, max, min* to comprehension syntax and restricting the input to 'flat' relations, we obtain a language which has the expressive power of SQL. It should be noticed that structural recursion is limited in expressive power as well. Nondeterministic tuple-generating (or object generating) while programs cannot be expressed. The definition of structural recursion over the union presentation uses the separation property since the h_2 function is only defined for disjoint collections. Therefore, programs which are not definable over the disjoint union of parts are not representable. However, most common database operations and all database operations in SQL are based on separation. The *traversal combinator* [**?**] concept is more general. It capture a large family of type-safe primitive functions. Most functions that are expressed as a sequence of recursive programs where only one parameter becomes smaller at each recursive call are members of this family. However, this restriction excludes functions such as structural equalities and ordering because they require their two input structures to be traversed simultaneously. The uniform traversal combinator generalizes this concept by combinators each takes functions as inputs and return a new function as output which performs the actual reduction.

An **EER query** is simply an EER expression of the EER algebra. The expression EER query is defined on the EER types $R_1, ..., R_n$ and maps to the target type $S_1, ..., S_m$. Any database schema $\mathcal{D}$ that contains $R_1, ..., R_n$ is therefore mapped to the schema $\mathcal{S}^q = \{S_1, ..., S_m\}$. EER queries may be enhanced by parameters. We can consider these parameters as an auxiliary schema $\mathcal{A}$. Therefore an EER query is a mapping from $\mathcal{D}$ and $\mathcal{A}$ to $\mathcal{S}$, i.e.

$$q \, : \, \mathcal{D} \times \mathcal{A} \, \rightarrow \, \mathcal{S}^q \quad .$$

A EER query for which the schema $\mathcal{S}^q$ consists of a singleton atomic attributes and base type is called *singleton-value query*.

The relational database model only allows target schemata consisting of one type. This restriction is not necessary for EER model. The target schema of the EER query is an EER schema as well. Therefore, we can build query towers by applying queries to the schemata obtained through query evaluation. Therefore, an EER query is a *schema transformation*.

Transactions combine modification operations and queries into a single program. Following [**?**], we define a transaction T over $(\mathcal{S}, \Sigma)$ as a finite sequence $o_1; o_2; o_3; ...; o_m$ of modification and retrieval operations over $(\mathcal{S}, \Sigma)$. Let $read(T)$ and $write(T)$ the set of basic read and write operations in T.

Transactions may be applied to the database state $\mathcal{D}^C$ sequentially and form a transition $T(\mathcal{D}^C) = o_m(...(o_2(o_1(\mathcal{D}^C))))$. The result of applying the transaction T to a database (state) $\mathcal{D}^C$ is the transition from this database to the database $T(\mathcal{D}^C)$. The transaction semantics is based on two assumptions for a databases schema $\mathcal{D} = (\mathcal{S}, \Sigma)$:

Atomicity and consistency: The program is either executed entirely or not executed at all and does preserve the semantics of the database.

Exhaustiveness: The transaction is executed only once.

The effect of application of T to $\mathcal{D}^C$ is defined as an *atomic constraint preserving transition*

$$T(\mathcal{D}^C) = \begin{cases} T(\mathcal{D}^C) & \text{if } T(\mathcal{D}^C) \models \Sigma \\ \mathcal{D}^C & \text{if } T(\mathcal{D}^C) \not\models \Sigma \end{cases}$$

We notice that atomic constraint preserving transitions can only be executed in a form *isolated* from each other. Transactions T_1 and T_2 are *competing* if $read(T_1) \cap write(T_2) \neq \emptyset$ or $read(T_2) \cap write(T_1) \neq \emptyset$ or $write(T_2) \cap write(T_1) \neq \emptyset$.

Parallel execution of transactions $T_1 \parallel T_2$ is *correct* if either the transactions are not competing or the effect of $T_1 \| T_2(\mathcal{S}^C)$ is equivalent to $T_1(T_2(\mathcal{S}^C))$ or to $T_2(T_1(\mathcal{S}^C))$ for any database $\mathcal{S}^C$. If parallel execution is correct transaction execution can be scheduled in parallel.

Exhaustiveness can be implemented by assigning two states to each transaction: inactive (for transactions that are not yet executed) and completed (for transactions that have lead to a constraint preserving transition).

A large variety of approaches to **workflow** specification has been proposed in the literature. We use basic computation step algebra introduced in [**?**]:

- Basic control commands are sequence ; (execution of steps in a sequence), parallel split $|^{\wedge}|$ (execute steps in parallel), exclusive choice $|^{\oplus}|$ (choose one execution path from many alternatives), synchronization $|^{sync}|$ (synchronize two parallel threads of execution by a synchronization condition ^{sync}, and simple merge $+$ (merge two alternative execution paths). The exclusive choice is considered to be the default parallel operation and is denoted by $||$.

- Structural control commands are arbitrary cycles * (execute steps w/out any structural restriction on loops), arbitrary cycles $^+$ (execute steps w/out any structural restriction on loops but at least once), optional execution $[\,]$ (execute the step zero times or once), implicit termination $\downarrow$ (terminate if there is nothing to be done), entry step in the step $\nearrow$ and termination step in the step $\searrow$.

The basic computation step algebra may be extended by advanced step commands:

- Advanced branching and synchronization control commands are multiple choice $|^{(m,n)}|$ (choose between m and n execution paths from several alternatives), multiple merge (merge many execution paths without synchronizing), discriminator (merge many execution paths without synchronizing, execute the subsequent steps only once) n-out-of-m join (merge many execution paths, perform partial synchronization and execute subsequent step only once), and synchronizing join (merge many execution paths, synchronize if many paths are taken, simple merge if only one execution path is taken).

- We may also define control commands on multiple objects (CMO) such as CMO with a priori known design time knowledge (generate many instances of one step when a number of instances is known at the design time), CMO with a priori known runtime knowledge (generate many instances of one step when a number of instances can be determined at some point during the runtime (as in for loops)), CMO with no a priori runtime knowledge (generate many instances of one step when a number of instances cannot be determined (as in while loops)), and CMO requiring synchronization (synchronization edges) (generate many instances of one activity and synchronize afterwards).
- State-based control commands are deferred choice (execute one of the two alternative threads, the choice which tread is to be executed should be implicit), interleaved parallel executing (execute two activities in random order, but not in parallel), and milestone (enable an activity until a milestone has been reached).
- Finally, cancellation control commands are used, e.g. cancel step (cancel (disable) an enabled step) and cancel case (cancel (disable) the case).

These control composition operators are generalizations of workflow patterns and follow approaches developed for Petri net algebras.

Operations defined on the basis of this general frame can be directly translated to database programs. So far no theory of database behavior has been developed that can be used to explain the entire behavior and that explain the behavior in depth for a run of the database system.

4.2.3 Views in the Enhanced Entity-Relationship Models

Classically, (simple) views are defined as singleton types which data is collected from the database by some query.

```
create view NAME (PROJECTION VARIABLES) as
   select PROJECTION EXPRESSION
     from  DATABASE SUB-SCHEMA
     where SELECTION CONDITION
     group by EXPRESSION FOR GROUPING
       having  SELECTION AMONG GROUPS
     order by ORDER WITHIN THE VIEW;
```

Since we may have decided to use the class-wise representation simple views are not the most appropriate structure for exchange specification. A **view schema** is specified on top of an EER schema by

- a schema $\mathcal{V} = \{S_1, ..., S_m\}$,
- an auxiliary schema $\mathcal{A}$ and
- a query $q : \mathcal{D} \times \mathcal{A} \rightarrow \mathcal{V}$ defined on $\mathcal{D}$ and $\mathcal{V}$.

Given a database $\mathcal{D}^C$ and the auxiliary database $\mathcal{A}^C$. The view is defined by $q(\mathcal{D}^C \times \mathcal{A}^C)$.

Additionally, views should support services. Views provide their own data and functionality. This object-orientation is a useful approach whenever data should be used without direct or remote connection to the database engine.

We generalize the view schema by the frame:

```
generate MAPPING :   VARS → OUTPUT STRUCTURE
    from DATABASE TYPES
    where SELECTION CONDITION
    represent using GENERAL PRESENTATION STYLE
        & ABSTRACTION (GRANULARITY, MEASURE, PRECISION)
        & ORDERS WITHIN THE PRESENTATION   & POINTS OF VIEW
        & HIERARCHICAL REPRESENTATIONS   & SEPARATION
    browsing definition CONDITION   & NAVIGATION
    functions SEARCH FUNCTIONS   & EXPORT FUNCTIONS   & INPUT FUNCTIONS
        & SESSION FUNCTIONS   & MARKING FUNCTIONS
```

The extension of views by functions seems to be superficial during database design. In distributed environments we save efforts of parallel and repetitive development due to the development of the entire view suite instead of developing each view by its own.

Let us consider an *archive view* for the schema in Figure 4.1. The view may be materialized and may be used as a read-only view. It is based on a slice that restricts the scope to those course that have been given in the summer term of the academic year 2000/01

Archive.Semester := e(Semester) for $e = \sigma_{\text{SemesterShortName}=\text{``SS00/01''}}$.

The view is given through the expressions

Archive.Course := e(CourseHeld [Course])
Archive.Person := e(CourseHeld[PlannedCourse[ProposedCourse [Responsible4Course : Person]]])
Archive.CourseHeld := e(CourseHeld[PlannedCourse[ProposedCourse[Course, Program, Teacher:Professor, Responsible4Course : Person], Kind]])

The types *Program, Kind, Professor* are given by similar expressions.
We additionally specify the functions that can be used for the archive view. The type *Semester* consists of one object and thus becomes trivial. This type is denoted by a dashed box. The view schema obtained is displayed in Figure 4.3. We observe that this view can be used directly for a representation through an XML schema.

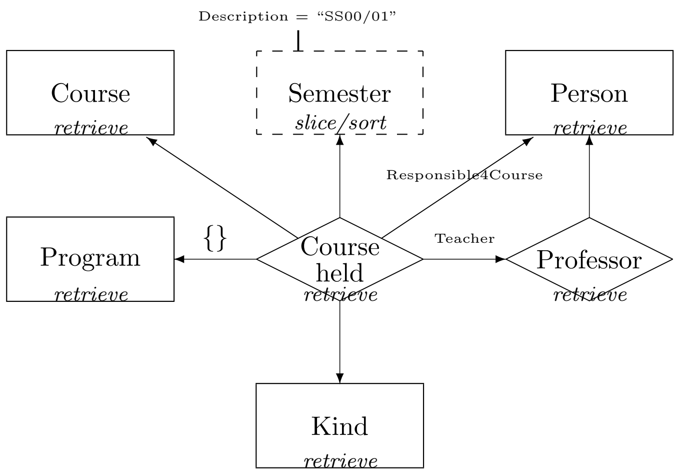

FIGURE 4.3: The EER View for Archiving Courses

Views may be used for distributed or collaborating databases. They can be enhanced by functions. **Exchange frames** are defined by

- an *exchange architecture* given through a system architecture integrating the information systems through communication systems,
- a *collaboration style* specifying the supporting programs, the style of cooperation and the coordination facilities, and
- *collaboration pattern* specifying the data integrity responsibilities, the potential changes within one view and the protocols for exchange of changes within one view.

4.2.4 Advanced Views and OLAP Cubes

The extended entity-relationship model can be used to define an advanced data warehouse model. Classically, the data warehouse model is introduced in an intuitive form by declaring an association or a relationship among components of the cube (called dimensions), by declaring attributes (called fact types) together with aggregation functions. Components may be hierarchically structured. In this case, the cube schema can be represented by a *star schema*. Components may be interrelated with each other. In this case the cube schema is represented by a *snowflake schema*. Star and snowflake schemata can be used for computing views on the schemata. View constructors are functions like drill-down, roll-up, slice, dice, and rotate. We demonstrate the power of the extended entity-relationship model by a novel, formal and compact definition of the OLAP cube schema and the corresponding operations.

The data warehouse model is based on **hierarchical data types**. Given an extended base type $B = (Dom(B), Op(B), Pred(B), \Upsilon)$, we may define a number of equivalence relations eq on $Dom(B)$. Each of these equivalence relations defines a **partition** Π_{eq} of the domain into equivalence classes. These equivalence classes c may be named by n_c. Let us denote named partitions by Π^*. The trivial named partition that only relates elements to themselves is denoted by $\perp^*$. We denote the named partition that consists of $\{Dom(B)\}$ and a name by $\top^*$.

Equivalence relations and partitions may be ordered. The *canonical order* of partitions on $DOM(B)$ relates two partitions Π^*, Π'^*. We define $\Pi^* \preceq \Pi'^*$ iff for all (c, n_c) from Π^* there exists one and only one element $(c', n_{c'}) \in \Pi'^*$ such that $c \subseteq c'$.
We also may consider non-classical orderings such as the *majority order* $\preceq_m^{\texttt{choice}}$ that relates two named partitions iff for all (c, n_c) from Π^* there exists one and only one element $(c', n_{c'}) \in \Pi'^*$ such that
either $|c \cap c'| > max\{|c \cap c''| \mid (c'', n_{c''}) \in \Pi'^*,\ c'' \neq c'\}$
or $(c', n_{c'}) \in \Pi'^*$ is determined by a (deterministic) `choice` operator among
$\{c^+ \in \Pi'^* \mid |c \cap c^+| = max\{|c \cap c''| \mid (c'', n_{c''}) \in \Pi'^*\}\}$.
If the last case does not appear then we omit the choice operator in $\preceq_m^{\texttt{choice}}$.

The *DateTime* type is a typical basic data type. Typical equivalence relations are eq_{hour} and eq_{day} that relate values from $Dom(DateTime)$ that belong to the same hour or day. The partitions $\perp^*$, $Days$, $Weeks$, $Months$, $Quarters$, $Years$, and $\top^*$ denote the named partitions of highest granularity, the named partitions of *DateTime* by days, by weeks, by months, by quarters, by years, and the trivial no-granularity named partition, correspondingly. We observe $\perp^* \preceq \Pi^*$ and $\Pi^* \preceq \top^*$ for any named partition in this list. We also notice that $Days \preceq Months \preceq Quarters \preceq Years$.
$Weeks \preceq_m Months$ is a difficult ordering that causes a lot of confusion.

This notion of hierarchical data types can be extended to complex attribute types, entity types, cluster types and relationship types. These extended types are also called *hierarchical* *types*. Aggregation functions are defined for extension based data types. The cube definition uses the association between attribute types and aggregation functions

The **grounding schema** of a cube is given by a (cube) relationship type $R = (R_1, ..., R_n, \{(A_1, q_1, f_1,), ..., (A_m, q_m, f_m)\})$ with

- hierarchical types $R_1, ..., R_n$ which form component (or dimension) types,
- ("fact") attributes $A_1, ..., A_m$ which are defined over extension based data types and instantiated by singleton-value queries $q_1, ..., q_m$ and
- aggregation functions $f_1, ..., f_m$ defined over $A_1, ..., A_m$.

The grounding schema is typically defined by a *view* over a database schema.

Given a grounding schema $R = (R_1, ..., R_n, \{(A_1, q_1, f_1,), ..., (A_m, q_m, f_m)\})$, a class R^C, and partitions Π_i on $DOM(R_i)$ for any component $R_1, ..., R_n$. A *cell* of R^C is a non-empty set $\sigma_{R_1 \in c_1, ... R_n \in c_n}(R^C)$ for $c_i \in \Pi_i$ and for the selection operation σ_α. Given named partitions $\Pi_1^*, ..., \Pi_n^*$ for all component types, a **cube** $cube^{\Pi_1^*, ..., \Pi_n^*}(R^C)$ on R^C and on Π_i^*, $1 \le i \le n$ consists of the set

$$\{\sigma_{R_1 \in c_1, ... R_n \in c_n}(R^C) \neq \emptyset \,|\, c_1 \in \Pi_1, ..., c_n \in \Pi_n\}$$

of all cells of R^C for the named partitions Π_i^*, $1 \le i \le n$. If $\Pi_i^* = \top^*$ then we may omit the partition Π_i^*.

Therefore, a cube is a special view. We may materialize the view. The view may be used for computations. Then each cell is recorded with its corresponding aggregations for the attributes. For instance, $sum(\pi_{\text{PriceOfGood}}(\sigma_{\text{SellingDate} \in Week_x}(R^C)))$ computes the total turnover in week x.

Spreadsheet cubes are defined for sequences $\Pi_1^* \preceq ... \preceq \Pi_n^*$ of partitions for one or more dimensional components. For instance, the partitions $Days$, $Months$, $Quarters$, $Years$ define a spreadsheet cube for components defined over $DateTime$.

The cube can use another representation: instead of using cells as sets we may use the names defining the cells as the cell dimension value. This representation is called **named cube**.

This definition of the cube can be now easily used for a precise mathematical definition of the main operations for cubes and extended cubes. For instance, given a cube with a partitions Π^*, Π'^* for one dimensional component with $\Pi^* \preceq \Pi'^*$, the `drill-down` operation transfers a cube defined on Π'^* to a cube defined on Π^*. `Roll-up` transfers a cube defined on Π^* to a cube defined on Π'^*. The `slice` operation is nothing else but the object-relational selection operation. The `dice` operation can be defined in two ways: either using the object-relational projection operation or using $\top$ partitions for all dimensional components that are out of scope. More formally, the following basic OLAP query functions are introduced for a cube $cube^{\Pi_1^*, ..., \Pi_n^*}(R^C)$ defined on the cube schema $R = (R_1, ..., R_n, \{(A_1, q_1, f_1,), ..., (A_m, q_m, f_m)\})$, a dimension i, and partitions $\Pi_i^* \preceq \Pi_i'^* \preceq \top_i^*$:

Basic drill-down functions map the cube $cube^{\Pi_1^*, ..., \Pi_i'^*, ..., \Pi_n^*}(R^C)$ to the cube $cube^{\Pi_1^*, ..., \Pi_i^*, ..., \Pi_n^*}(R^C)$.

Basic roll-up functions map the cube $cube^{\Pi_1^*, ..., \Pi_i^*, ..., \Pi_n^*}(R^C)$ to the cube $cube^{\Pi_1^*, ..., \Pi_i'^*, ..., \Pi_n^*}(R^C)$.
Roll-up functions are the inverse of drill-down functions.

Basic slice functions are similar to selection of tuples within a set. The cube $cube^{\Pi_1^*, ..., \Pi_n^*}(R^C)$ is mapped to the cube $\sigma_\alpha(cube^{\Pi_1^*, ..., \Pi_n^*}(R^C))$.

The slice function can also be defined through cells. Let *dimension*(α) the set of all dimensions that are restricted by α. Let further

$$\sigma_\alpha^\sqcap(c_i) \quad = \quad \begin{cases} \emptyset & \text{if } R_i \in \mathit{dimension}(\alpha) \wedge \sigma_\alpha(c_i) \neq c_i \\ c_i & \text{otherwise} \end{cases}$$

Close slice functions restrict the cube cells to those that entirely fulfill the selection criterion α, i.e., $\{\sigma_{R_1 \in \sigma_\alpha^\sqcap(c_1),\ldots R_n \in \sigma_\alpha^\sqcap(c_n)}(R^C) \neq \emptyset \,|\, c_1 \in \Pi_1, ..., c_n \in \Pi_n\}$.
Liberal slice functions restrict the cells to those that partially fulfill the selection criterion α, i.e. to cells $\{\sigma_{R_1 \in \sigma_\alpha(c_1),\ldots R_n \in \sigma_\alpha(c_n)}(R^C) \neq \emptyset \,|\, c_1 \in \Pi_1, ..., c_n \in \Pi_n\}$.
Lazy and eager slice functions apply the selection functions directly to values in the cells.

Basic dice functions are similar to projection in the first-order query algebra. They map the cube $\mathit{cube}^{\Pi_1^*,\ldots,\Pi_i^*,\ldots,\Pi_n^*}(R^C)$ to the cube $\mathit{cube}^{\Pi_1^*,\ldots,\top_i^*,\ldots,\Pi_n^*}(R^C)$.
Basic dice functions are defined as special roll-up functions. We also may omit the dimension i. In this case we loose the information on this dimension.

Generalizing the first-order query algebra, [?] defines additional OLAP operations such as

join functions for mergers of cubes,

union functions for union of two or more cubes of identical type,

rotation or pivoting functions for rearrangement of the order of dimensions, and

rename functions for renaming of dimensions.

Our new definition of the cube allows to generalize a large body of knowledge obtained for object-relational databases to cubes. The *integration* of cubes can be defined in a similar form [?].

4.3 Semantics of EER Models

4.3.1 Semantics of Structuring

Each structuring also uses a set of **implicit model-inherent integrity constraints**:

Component-construction constraints are based on existence, cardinality and inclusion of components. These constraints must be considered in the translation and implication process.

Identification constraints are implicitly used for the set constructor. Each object either does not belong to a set or belongs only once to the set. Sets are based on simple generic functions. The identification property may be, however, only representable through automorphism groups [?]. We shall later see that value-representability or weak-value representability lead to controllable structuring.

Acyclicity and finiteness of structuring supports axiomatization and definition of the algebra. It must, however, be explicitly specified. Constraints such as cardinality constraints may be based on potential infinite cycles.

Superficial structuring leads to the representation of constraints through structures. In this case, implication of constraints is difficult to characterize.

Implicit model-inherent constraints belong to the performance and maintenance traps. We distinguish between constraints with a semantical meaning in the application. These constraints must either be maintained or their validity can be controlled by a controller. Constraints can either be declared through logical formulas are given by four layers:

- Constraints are declared at the *declaration layer* based on logical formulas or constructs and based on the schema declaration.
- Constraints are extended at the *technical layer* by methods for their maintenance, by rules treatment of their invalidity, by enactment strategies and by auxiliary methods.
- Constraint maintenance is extended at the *technological layer* under explicit consideration of the DBMS programs (e.g., for update in place, in private or in separation) and by transformation to dynamic transition constraints.
- Constraints specification is extended at the *organizational layer* by integration into the architecture of the system, by obligations for users that impose changes in the database, and for components of the system.

Relational DBMS use a constraint specification at the declaration and technical layers. For instance, foreign key constraints explicitly specify which constraint enforcement technics are applied in the case of invalidity of constraint. The systems DB2, Oracle and Sybase use distinct scheduling approaches for constraint maintenance at the technological layer. Constraints may be mainly maintained through the DBMS or be partially maintained through interface programs restricting invalidation of constraints by users.

Implicit language-based integrity constraints are typically based on the mini-semantics of the names that are used for denotation of concepts. EER modelling is based on a *closed-world approach*. Constructions that are not developed for the schema are neither not existing in the application nor not important for the database system.

Synonyms form typical implicit constraints. Given two queries q_1, q_2 on $\mathcal{D}$, an empty auxiliary schema $\mathcal{A}$ and the target schema $\mathcal{S}$. A synonym constraint $q_1 \approx q_2$ is valid for the database $\mathcal{D}^C$ iff $q_1(\mathcal{D}^C) = q_2(\mathcal{D}^C)$. *Homonyms* $S \between T$ describe structural equivalence combined at the same time with different meaning and semantics. Homonyms may be simply seen as the negation or inversion of synonymy. *Hyponyms* and *hypernyms* $S \preccurlyeq T$ hint on subtype associations among types under consideration. The type T can be considered to be a more general type than S. *Overlappings* and *compatibilites* $S \Cup T$ describe partial similarities.

Exclusion constraints state that two schema types or in general two expressions on the schema will not share any values or objects. Given two queries q_1, q_2 on $\mathcal{D}$, an empty auxiliary schema $\mathcal{A}$ and the target schema $\mathcal{S}$. An exclusion constraint $q_1 || q_2$ is valid for the database $\mathcal{D}^C$ iff $q_1(\mathcal{D}^C) \cap q_2(\mathcal{D}^C) = \emptyset$.

Implicit model-based exclusion constraints exclude common values for basic data types. For instance, the constraint *Semester[Year]* || *Room[Number]* is assumed in the schema in Figure 4.1 without any explicit specification. Implicit language-based exclusion constraints use the natural understanding of names. For instance, the constraint *Course[Title]* || *Professor[Title]* is valid in any university application.

Inclusion constraints state that two schema types or in general two expressions on the schema are in a subtype association. Given two queries q_1, q_2 on $\mathcal{D}$, an empty auxiliary schema $\mathcal{A}$ and the target schema $\mathcal{S}$. An inclusion constraint $q_1 \subseteq q_2$ is valid for the database $\mathcal{D}^C$ iff $q_1(\mathcal{D}^C) \subseteq q_2(\mathcal{D}^C)$.

Implicit model-based inclusion constraints form the most important class of implicit constraints. The EER model assumes $R_1[R_2[ID]] \subseteq R_2[ID]$ for any relationship type R_1, its component type R_2, and the identification ID of R_2.

The axiomatization of exclusion and inclusion is rather simple [**?**]. It may be either based on logics of equality and inequality systems or on set-theoretic reasoning.

Explicit integrity constraints can be declared based on the B(eeri-)V(ardi)-frame, i.e. by an implication with a formula for premises and a formula for the implication. BV-constraints do not lead to rigid limitation of expressibility. If structuring is hierarchic then BV-constraints can be specified within the first-order predicate logic. We may introduce a variety of different classes of integrity constraints:

Equality-generating constraints allow to generate equalities among these objects or components for a set of objects from one class or from several classes.

Object-generating constraints require the existence of another object set for a set of objects satisfying the premises.

A class $\mathcal{C}$ of integrity constraints is called *Hilbert-implication closed* if it can be axiomatized by a finite set of bounded derivation rules and a finite set of axioms. It is well-known that the set of join dependencies is not Hilbert-implication closed for relational structuring. However, an axiomatization exists with an unbounded rule, i.e., a rule with potentially infinite premises.

Functional dependencies are one of the most important class of equality-generating constraints. Given a type R and substructures X, Y of R.
The functional dependency $R : X \longrightarrow Y$ is valid in R^C if $o|_Y = o'|_Y$ whenever $o|_X = o'|_X$ for any two objects o, o' from R^C.
A **key dependency** or simply *key* X is a functional dependency $R : X \longrightarrow R$. A key is called *minimal* if none of its proper substructures forms a key. The set of all minimal keys of R is denoted by $Keys(R)$. We notice that this set may be very large. For instance, an entity type E with n atomic attributes may have $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ minimal keys which is roughly $\frac{2^n}{c\sqrt{n}}$ for a constant c.

The example depicted in Figure 4.1 uses the following functional dependencies and keys:
Keys(Person) = { { *PersNo* }, { *Name, DateOfBirth* } }
Keys(PlannedCourse) = { { *Course, Semester* }, {*Semester, Room, Time* }, {*Semester, Teacher, Time*} }
PlannedCourse : {*Semester, Time, Room*} $\longrightarrow$ {{*Program*}, *Teacher, Course*}
PlannedCourse : {*Teacher, Semester, Time*} $\longrightarrow$ {*Course, Room*}
ProposedCourse : {*Semester, Course*} $\rightarrow$ {*Teacher*}

The following axiomatization is correct and complete for functional dependencies in the EER [?] for substructures X, Y, Z of the EER type R:
Axioms: $R : X \sqcup_R Y \longrightarrow Y$
Rules:

$$\frac{R : X \longrightarrow Y}{R : X \longrightarrow X \sqcup_R Y} \qquad \frac{R : X \longrightarrow Y,\ R : Y \longrightarrow Z}{R : X \longrightarrow Z} \qquad \frac{R' : X \longrightarrow Y}{R : R'[X] \longrightarrow R'[Y]}$$

The type R' denotes a component type of R if R is a relationship type of order i and R' is of order $i - 1$.

Domain constraints restrict the set of values. Given an EER type R, a substructure T' of R, its domain $Dom(T')$, and a subset D of $Dom(T')$. The domain constraint $R : T' \subseteq D$ is valid in R^C if $\pi_{T'}(R^C) \subseteq D$.
For instance, we may restrict the year and the description of semesters by the constraints
Semester.Year $\subseteq$ { *1980, ..., 2039* } and
Semester.Description $\subseteq$ { *WT, ST* }.

Cardinality constraints are the most important class of constraints of the EER model. We distinguish two main kinds of cardinality constraints. Given a relationship type $R \doteq$

$(compon(R), attr(R), \Sigma)$, a component R' of R, the remaining substructure $R'' = R \setminus R'$ and the remaining substructure $R''' = R'' \sqcap_R compon(R)$ without attributes of R.

The ***participation constraint*** $card(R, R') = (m, n)$ holds in a relationship class R^C if for any object $o' \in R'^C$ there are at least m and at most n objects o with $o|_{R'} = o'$, i.e.,

$m \leq |\{ o \in R^C \mid o|_{R'} = o' \}| \leq n$ for any $o' \in \pi_{R'}(R^C)$.

The ***lookup constraint*** $look(R, R') = m..n$ holds in a relationship class R^C if for any object $o'' \in Dom(R'')$ there are at least m and at most n related objects o' with $o|_{R'} = o'$, i.e., $m \leq |\{ o' \in \pi_{R'}(R^C) \mid o \in R^C \wedge o|_{R'} = o' \wedge o|_{R'''} = o''' \}| \leq n$ for any $o''' \in Dom(R''')$.

Lookup constraints were originally introduced by P.P. Chen [**?**] as cardinality constraints. UML uses lookup constraints. They are easy to understand for binary relationship types without attributes but difficult for all other types. Lookup constraints do not consider attributes of a relationship type. They cannot be expressed by participation constraints. Participation constraints cannot be expressed by lookup constraints. In the case of a binary relationship type $R \doteq (R_1, R_2, \emptyset, \Sigma)$ without attributes we may translate these constraints into each other:

$card(R, R_1) = (m, n)$ iff $look(R, R_2) = m..n$.

Furthermore, participation and lookup constraints with the upper bound 1 and the lower bound 0 are equivalent to functional dependencies for a relationship type R:

$card(R, R') = (0, 1)$ iff $R : R' \longrightarrow R''$ and
$look(R, R') = 0..1$ iff $R : R''' \longrightarrow R'$.

The lower bounds of lookup and participation constraints are not related to each other. They are however related for binary relationship types. The lower bound 1 expresses an inclusion constraint:

$card(R, R') = (1, n)$ iff $R[R'] \subseteq R'$ and
$look(R, R') = 1..n$ iff $R[R'''] \subseteq R'''$.

The diagram can be enhanced by an explicit representation of cardinality and other constraints. If we use participation constraints $card(R, R') = (m, n)$ then the arc from R to R' is labelled by (m, n) If we use lookup constraints $look(R, R') = m..n$ for binary relationship types then the arc from R to R' is labelled by $m..n$.

Cardinality constraints are restrictions of combinatorics within a schema. Sets of cardinality constraints defined on a subschema $\mathcal{S}'$ may be *finitely inconsistent* in the sense that any database on $\mathcal{S}'$ has either a class for a type in $\mathcal{S}'$ that is empty or that is infinite.

Consider for instance the following relationship type:

DirectPrerequisite $\doteq$ (HasPrerequisite : Course, IsPrerequisite : Course, $\emptyset$, Σ)
card(DirectPrerequisite, HasPrerequisite) = (0,2) and
card(DirektVoraussetz, IsPrerequisite) = (3,4) .

These cardinality constraints are only satisfied in a database with either an empty set of courses or an infinite set of courses.

Participation and lookup constraints can be extended to substructures and intervals. Given a relationship type R, a substructure R' of R,the remaining substructure $R'' = R \setminus R'$ and the remaining substructure $R''' = R'' \sqcap_R compon(R)$ without attributes of R. Given furthermore an interval $I \subseteq \mathbb{N}_0$ of natural numbers including 0.

The ***(general) cardinality constraint*** $card(R, R') = I$ holds in a relationship class R^C if for any object $o' \in \pi_{R'}(R^C)$ there are $i \in I$ objects o with $o|_{R'} = o'$, i.e.,

$|\{ o \in R^C \mid o|_{R'} = o' \}| \in I$ for any $o' \in \pi_{R'}(R^C)$.

A participation constraint $card(R, R') = (m, n)$ is just a general cardinality constraint with the interval $\{m, ..., n\}$. A lookup constraint $look(R, R') = m..n$ is just a general cardinality constraint $card(R, R''') = I$ with the interval $\{m, ..., n\}$.

General cardinality constraints are necessary whenever we consider sets of cardinality constraints. There are examples of participation constraints which cannot be expressed through lookup or participation constraints. At the same time general cardinality constraints can be inferred from these participation constraints.

If $R = R'$ then the general cardinality constraint specifies the cardinality bounds of the relationship class.

The definition of general cardinality constraints can be extended to entity types as well.

Cardinality constraints restrict relationships. We are not able to defer equalities. Consider for instance the relationship type

$\mathit{Spouse} \stackrel{\circ}{=} = (\mathit{IsSpouse} : \mathit{Person}, \mathit{OfSpouse} : \mathit{Person}, \{ \mathit{From}, \mathit{To} \}, \Sigma)$.

Neither $card(\mathit{Spouse}, \mathit{IsSpouse\ From}) = (0,1)$ and $card(\mathit{Spouse}, \mathit{OfSpouse\ From}) = (0,1)$ nor $look(\mathit{Spouse}, \mathit{IsSpouse}) = 0..1$ and $look(\mathit{Spouse}, \mathit{OfSpouse}) = 0..1$ express the statement of monogamic societies that the spouse of the spouse is the person itself. The lookup constraints are only valid if a person can be married once. The Islamic rule would be that either the spouse of a spouse is the person itself or the spouse of the spouse of the spouse is the spouse.

Cardinality constraints combine equality-generating and object-generating constraints into a singleton construct. This convenience for declaration is paid back by the impossibility to axiomatize these constraints.

Participation cardinality constraints cannot be axiomatized [?]. If axiomatization is restricted to the upper bound then an axiomatization can be based on the following system for a one-type derivations [?]:

Axioms: $card(R, X) = (0, \infty)$

Rules:

$$\frac{card(R, X) = (0, b)}{card(R, X \sqcup_R Y) = (0, b)} \qquad \frac{card(R, X) = (0, b)}{card(R, X) = (0, b + c)}$$

Functional dependencies can be defined through generalized cardinality constraints, i.e. the functional dependency $R : X \longrightarrow Y$ is equivalent to $card(R[X \sqcup_R Y], X) = (0, 1)$. The Armstrong axiomatization provided above can be combined with the axiomatization for upper bound of participation constraints and the following system:

$$\frac{R : X \longrightarrow Y \ , \ card(R, Y) = (0, b)}{card(R, X) = (0, b)} \qquad \frac{card(R, X) = (0, 1)}{R : X \longrightarrow R}$$

These three systems are complete and correct for derivation of upper bounds.

We may also conclude rules for many-type derivations [?]. A typical rule is the following one:

$$\frac{card(R, R') = (m, n), \ card(R', R'') = (m', n')}{card(R, R'') = (m \cdot m', n \cdot n')} \quad .$$

Multivalued dependencies are best fitted to the ER model and are difficult to define, to teach, to handle, to model and to understand within relational database models. Given an EER type R and partition of components of R into X, Y and Z, the multivalued dependency $X \twoheadrightarrow Y|Z$ is ER-valid in a class R^C defined over the type R (denoted by $R^C \models_{ER} X \twoheadrightarrow Y|Z$) if the type can be decomposed into three types representing X, Y, and Z and two mandatory relationship types defined on $X \cup Y$ and $X \cup Z$, respectively.

The multivalued dependency can be represented by a decomposition of the type R displayed in Figure 4.4.

We may use more compact schemata given in Figure 4.5. In this case, the relationship type with the components ${}_{(X)}Z$ is based on the X-components. We denote the relationship

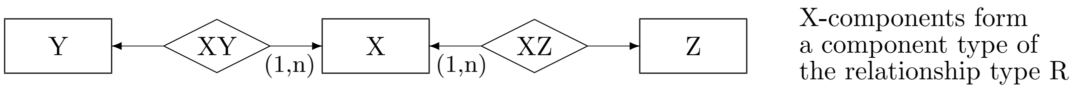

FIGURE 4.4: The ER Representations of a Multivalued Dependency for a First-Order Relationship Type

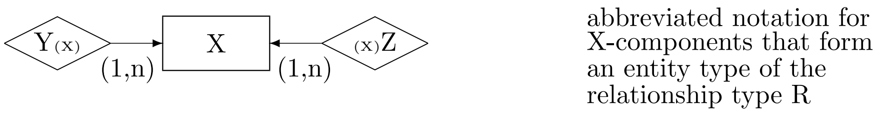

FIGURE 4.5: Compact Representations of a Multivalued Dependency

type that relates X with Z by $_{(X)}Z$ or by $Z_{(X)}$. This notion shows the direct decomposition imposed by the multivalued dependency.

The deductive system in Figure 4.6 consisting of the trivial multivalued dependency, the root reduction rule, and the weakening rule is correct and complete for inference of multivalued dependencies. We observe that the axiomatization of functional and multivalued dependencies can be derived in a similar way.

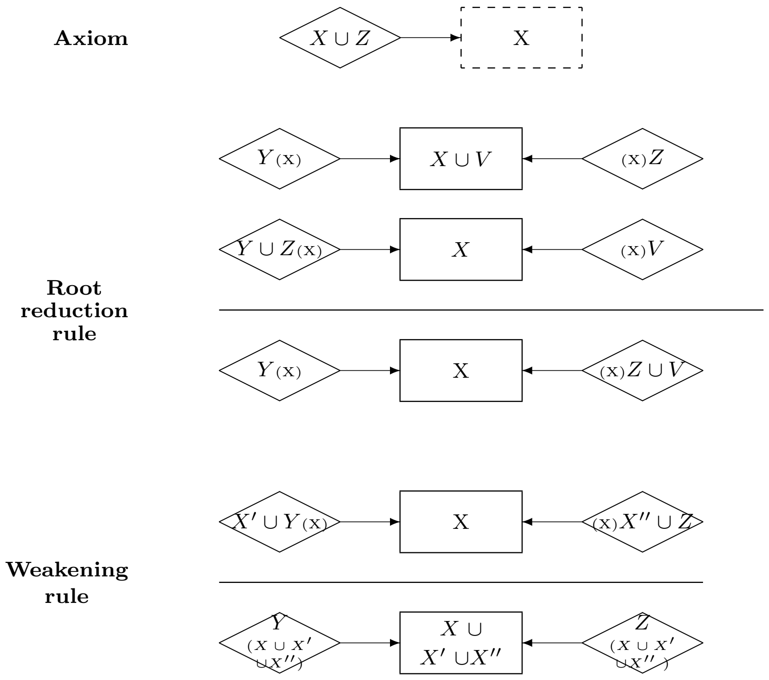

FIGURE 4.6: The Deductive System for ER Schema Derivation of Multivalued Dependencies

The constraints discussed above can be used for decomposition of types and restructuring of schemata. Pivoting has been introduced in [?] and allows to introduce a relationship type of higher order after factoring out constraints that are applicable to some but not all of the components of the relationship type.

Let us consider the schema depicted in Figure 4.1. The relationship type *ProposedCourse* has the arity 8. In the given application the following integrity constraints are valid:

$$\{ \textit{Course} \} \quad \rightarrow\!\!\!\!\rightarrow \quad \{ \textit{Program} \}$$
$$\{ \textit{Program} \} \quad \rightarrow\!\!\!\!\rightarrow \quad \{ \textit{Course} \}$$
$$\{ \textit{Course, Program} \} \quad \longrightarrow \quad \{ \textit{Responsible4Course : Person} \} \quad .$$

The last functional dependency is a ‘dangling’ functional dependency, i.e., the right side components are not elements of any left side of a functional or multivalued dependency. Considering the axiomatization of functional and multivalued dependencies we conclude that right side components of a dangling functional dependency can be separated from other components. The first two constraints can be used for separation of concern of the type *ProposedCourse* into associations among

Course, Program, Responsible4Course : Person

and

Course, Kind, Semester, Professor, Time(Proposal,SideCondition), Room, InsertedBy : Person.

According to decomposition rules we find additional names for the independent component sets in *ProposedCourse*, i.e., in our case *CourseObligation*.

We additionally observe that *InsertedBy : Person* can be separated from the last association. This separation is based on *pivoting*, i.e. building a relationship type ‘on top’ of the ‘remaining’ type

$$\textit{ProposedCourseRevised} \mathrel{\hat{=}} (\textit{Teacher: Professor, Course, Proposal : Kind, Request : Room, Semester, } \{ \textit{ Time(Proposal, SideCondition) } \}).$$

Finally, let us consider a constraint on *Course* and *Kind*. In the given application we assume that the selection of the kind for a course is independent of the other components, i.e.

$$\textit{ProposedCourseRevised : } \{ \textit{ Course, Semester } \} \quad \rightarrow\!\!\!\!\rightarrow \quad \{ \textit{ Kind } \} \quad .$$

This constraint hints on a flaw in modelling. The association between *Course* and *Kind* may vary over semesters for lectures. *Kind* is an enumeration type that categorizes the style of how lectures are given. The selection of the kind becomes easier if we develop an abstraction *Applicable* that collects all possible associations between *Course* and *Kind*.

We may also use a pragmatical rule for naming. If a decomposition leads to a separation based on roles then we may use the role name for the relationship type name.

One of our separation rules allows us to separate optional components of a relationship type by pivoting the type into a new relationship type. We use this rule for pivoting *PlannedCourse* and *CourseHeld*.

We finally derive the schema displayed in Figure 4.7. This schema is based on *CourseProposal*, *CoursePlanned* and *Lecture*.

4.3.2 Semantics of Functionality

Static constraints in a schema $(\mathcal{S}, \Sigma)$ can be transformed to **transition constraints** [?]. A transition constraint $(\Psi_{pre}, \Psi_{post})$ defines the preconditions and postconditions for state transitions of databases defined over $\mathcal{S}$. Given a transition τ converting the database $\mathcal{S}^{C_1}$ to the database $\mathcal{S}^{C_2} = \tau(\mathcal{S}^{C_1})$. The transition constraint $(\Psi_{pre}, \Psi_{post})$ is valid for the transition $(\mathcal{S}^{C_1}, \tau(\mathcal{S}^{C_1}))$ if from $\mathcal{S}^{C_1} \models \Psi_{pre}$ follows that $\mathcal{S}^{C_2} \models \Psi_{post}$.

Static constraints Σ are therefore converted to a transition constraint (Σ, Σ).

Database dynamics is defined on the basis of transition systems. A *transition system* on the schema S is a pair $\mathcal{TS} = (\mathcal{S}, \{\xrightarrow{a} \mid a \in \mathcal{L}\})$

where $\mathcal{S}$ is a non-empty set of state variables,

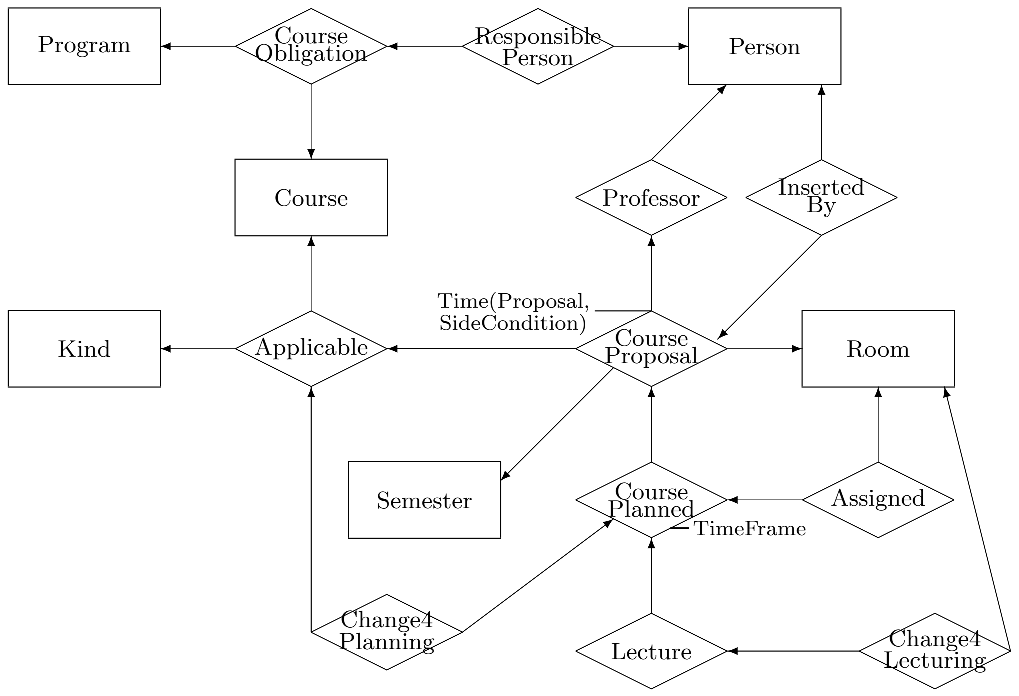

FIGURE 4.7: The Decomposition of the HERM Diagram in Figure 4.1

$\mathcal{L}$ is a non-empty set (of labels), and
$\xrightarrow{a} \subseteq \ \mathcal{S} \times (\mathcal{S} \cup \{\infty\})$ for each $a \in \mathcal{L}$.

State variables are interpreted by database states. Transitions are interpreted by transactions on S.

Database lifetime is specified on the basis of paths on $\mathcal{TS}$. A *path* π through a transition system is a finite or ω length sequence of the form $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ....$ The length of a path is its number of transitions.

For the transition system $\mathcal{TS}$ we can introduce now a *temporal dynamic database logic* using the quantifiers $\forall_f$ (always in the future)), $\forall_p$ (always in the past), $\exists_f$ (sometimes in the future), $\exists_p$ (sometimes in the past).

First-order predicate logic can be extended on the basis of temporal operators for timestamps ts. Assume a temporal class (R^C, l_R). The validity function I is extended by time and is defined on $S(ts, R^C, l_R)$. A formula α is valid for $I_{(R^C,l_R)}$ in ts if it is valid on the snapshot defined on ts, i.e. $I_{(R^C,l_R)}(\alpha, ts) = 1$ iff $I_{S(ts,R^C,l_R)}(\alpha, ts)$.

- For formulas without temporal prefix the extended validity function coincides with the usual validity function.
- $I(\forall_f \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for all $ts' > ts$;
- $I(\forall_p \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for all $ts' < ts$;
- $I(\exists_f \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for some $ts' > ts$;
- $I(\exists_p \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for some $ts' < ts$.

The modal operators $\forall_p$ and $\exists_p$ ($\forall_f$ and $\exists_f$ respectively) are dual operators, i.e. the two formulas $\forall_h \alpha$ and $\neg\exists_h \neg\alpha$ are equivalent. These operators can be mapped onto classical modal logic with the following definition:

$$\Box\alpha \quad \equiv \quad (\forall_f \alpha \wedge \forall_p \alpha \wedge \alpha);$$
$$\Diamond\alpha \quad \equiv \quad (\exists_f \alpha \vee \exists_p \alpha \vee \alpha).$$

In addition, temporal operators *until* and *next* can be introduced.

The most important class of dynamic integrity constraint are state-transition constraints $\alpha\, O\, \beta$ which use a pre-condition α and a post-condition β for each operation O. The state-transition constraint $\alpha\, O\, \beta$ can be expressed by the the temporal formula $\alpha \xrightarrow{O} \beta$. A state-transition constraint $\alpha\, O\, \alpha$ is called an *invariant* of O.

Each finite set of static integrity constraints can be expressed equivalently by a set of state-transition constraints or invariants $\{ \bigwedge_{\alpha\in\Sigma} \alpha \xrightarrow{O} \bigwedge_{\alpha\in\Sigma} \alpha) \mid O \in Alg(M) \}$.

Integrity constraints may be enforced

- either at the procedural level by application of
  - trigger constructs [?] in the so-called active event-condition-action setting,
  - greatest consistent specializations of operations [?],
  - or stored procedures, i.e., full-fledged programs considering all possible violations of integrity constraints,
- or at the transaction level by restricting sequences of state changes to those which do not violate integrity constraints,
- or by the DBMS on the basis of declarative specifications depending on the facilities of the DBMS,
- or at the interface level on the basis of consistent state changing operations.

Database constraints are classically mapped to transition constraints. These transition constraints are well-understood as long as they can be treated locally. Constraints can thus be supported using triggers or stored procedures. Their global interdependence is, however, an open issue.

The transformation to event-condition-action rules is not powerful enough. Consider the following example [?]:

$R_1 \doteq (R_3, R_4, \emptyset, \Sigma_1), \quad card(R_1, R_4) = (1, n)$,
$R_2 \doteq (R_5, R_6, \emptyset, \Sigma_2), \quad card(R_2, R_5) = (0, 1), \quad card(R_2, R_6) = (1, n)$,
$R_3 \doteq (R_6, ..., \emptyset, \Sigma_3), \quad card(R_3, R_6) = (0, 1)$, and $R_4 || R_5$.

The greatest consistent specialization of the operation Insert($R_1, (a, c)$) is the operation

```
Insert($R_1, (a, c)$) $\leftrightsquigarrow$
    if $c \notin R_2[R_5]$ then fail
    else begin Insert($R_1, (a, c)$);
         if $a \notin R_1[R_3]$ then Insert($R_2, (a, d)$) where $d \notin R_1[R_4] \cup R_2[R_5]$ end ;
```

This operation cannot be computed by trigger constructs. They result in deletion of a from $R_1[R_3]$ and deletion of c from $R_2[R_5]$ and thus permit insertion into R_1.

4.4 Problems with Modelling and Constraint Specification

The main deficiency is the constraint acquisition problem. Since we need a treatment for sets a more sophisticated reasoning theory is required. One good candidate is visual or graphical reasoning that goes far beyond logical reasoning [?].

Most modelling approaches assume *constraint set completeness*, i.e. all constraints of certain constraint classes which are valid for an application must be explicitly specified or derivable. For instance, normalization algorithms are based on the elicitation of complete knowledge on all valid functional dependencies. Therefore, the designer should have tools or theories about how to obtain all functional dependencies that are independent from the functional dependencies already obtained and that are not known to be invalid.

Excluded functional constraints $X \not\longrightarrow Y$ state that the functional dependency $X \longrightarrow Y$ is not valid.

Excluded functional constraints and functional dependencies are axiomatizable by the following formal system [?].

Axioms

$$X \cup Y \;\rightarrow\; Y$$

Rules

$$(1)\ \frac{X \longrightarrow Y}{X \cup V \cup W \longrightarrow Y \cup V} \qquad (2)\ \frac{X \longrightarrow Y\,,\quad Y \longrightarrow Z}{X \longrightarrow Z}$$

$$(3)\ \frac{X \longrightarrow Y\,,\quad X \not\longrightarrow Z}{Y \not\longrightarrow Z}$$

$$(4)\ \frac{X \not\longrightarrow Y}{X \not\longrightarrow Y \cup Z} \qquad (5)\ \frac{X \cup Z \not\longrightarrow Y \cup Z}{X \cup Z \not\longrightarrow Y}$$

$$(6)\ \frac{X \longrightarrow Z\,,\quad X \not\longrightarrow Y \cup Z}{X \not\longrightarrow Y} \qquad (7)\ \frac{Y \longrightarrow Z\,,\quad X \not\longrightarrow Z}{X \not\longrightarrow Y}$$

Rules (3) and (7) are one of the possible inversions of rule (2) since the implication $\alpha \wedge \beta \rightarrow \gamma$ is equivalent to the implication $\neg\gamma \wedge \beta \rightarrow \neg\alpha$. Rules (4) and (5) are inversions of rule (1). Rule (6) can be considered to be the inversion of the following union rule valid for functional dependencies:

$$(8)\ \frac{X \longrightarrow Y\,,\quad X \longrightarrow Z}{X \longrightarrow Y \cup Z}$$

This rule can be derived from the axiom and rule (2).

Constraint elicitation can be organized by the following approach:

Specification of the set of valid functional dependencies Σ_1***:*** All dependencies that are known to be valid and all those that can be implied from the set of valid and excluded functional dependencies.

Specification of the set of excluded functional dependencies Σ_0***:*** All dependencies that are known to be invalid and all those that can be implied from the set of valid and excluded functional dependencies.

This approach leads to the following simple elicitation algorithm:

1. *Basic step: Design obvious constraints.*
2. *Recursion step: Repeat until the constraint sets* Σ_0 *and* Σ_1 *do not change.*

- *Find a functional dependency* α *that is neither in* Σ_1 *nor in* Σ_0*.*
    - *If* α *is valid then add* α *to* Σ_1*.*
    - *If* α *is invalid then add* α *to* Σ_0*.*
- *Generate the logical closures of* Σ_0 *and* Σ_1*.*

This algorithm can be refined in various ways. Elicitation algorithms known so far are all variation of this simple elicitation algorithm.

The constraint acquisition process based on this algorithm is illustrated in Figure 4.8.

References

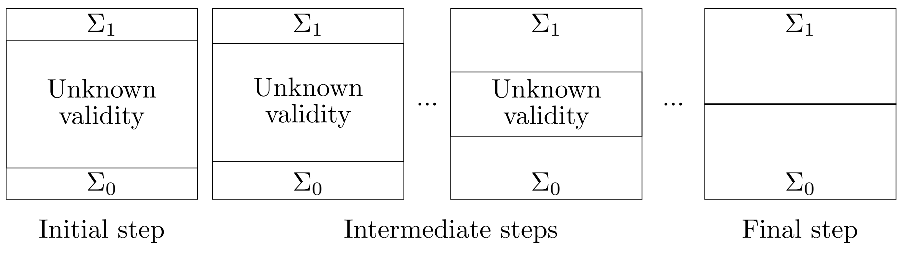

FIGURE 4.8: Constraint Acquisition Process