DATABASE DESIGN

Databases and database technology play a major role in modern companies and organizations. Information is one of the key factors of production and administration, and information has to be managed by a reliable technology: database management systems.

However, reliable software for storing and retrieving data can only provide properties like crash recovery, synchronization, availability, and efficient access. The *quality of data* can only be guaranteed by carefully designing the database structures. For this reason, the *database design process* becomes important. The best database management system is not able to correct a bad database design that does not reflect the semantics of the application information.

Database design is therefore one of the major research areas of database technology. There are several textbooks focusing on the various phases of the design process, for example, Refs. 1, 2, and 3, and whole conference series and journals are devoted to database design problems.

Usually, a database system is composed from one or several databases (DB) and a database management system (DBMS). Following this convention, the design of a database system focuses on a static database structure. However, the dynamics part of the use of the data has to be designed, too. Therefore, we often use the term *design of database application* if we want to highlight the joint design of database structure and application dynamics.

Because it is impossible to handle such a broad area in detail in a single article without restricting the scope to certain aspects, we focus on design of database applications in the presence of legacy databases and legacy applications. In such scenarios an integrated database schema cannot be designed from scratch but has to respect the existing software and data. This type of scenario is more realistic than the classical scenarios where a database infrastructure introduces electronic information management into a company or organization that has had a noncomputer-based management.

However, we will start with describing the classical database design process, which is a variant of the well-known software life-cycle models. The single phases have specific data models for describing the information structure on different abstraction levels, corresponding consistency rules, as well as normalization methods. Between these representations transformation methods support the design process. As usual in software design, later design phases influence earlier phases, leading to feedback cycles in the process.

After the description of the classical database design process, we present the concepts and architectures of multi-database and federated database systems, which allow the coexistence of local (legacy) databases in an information system and enable a global, uniform, and integrated view on the stored data.

The last part of this article describes the process of designing a global, integrated schema as an integration of the local schemata. The schema integration has to overcome heterogeneity on data model and schema level. Due to the complexity of this task an ad hoc solution for practical scenarios often fails. Therefore a design method helps to integrate the local schemata. We will give a short overview of the design problems and approaches to overcome heterogeneity.

TRADITIONAL DATABASE DESIGN

In this section we give an overview of the classical database design process. As usual in software engineering, the process of databases design can be separated in phases where an abstract informal description is transformed into a usable database implementation. Of course this process has feedback loops where problems detected in later phases influence the earlier phases. Each phase has specific types of design documents and methods.

We will start with an overview on the design process and then discuss the single phases in detail. Because we will focus on integration of databases on the conceptual and logical level in the remainder of this contribution, our focus is on those phases connected with this topic. For the other phases we describe the key principles only.

Database Design Process

The classical database design process is depicted in Fig. 1. There are numerous variations of this process in the literature. We follow roughly the presentation of the design phases in Ref. 4.

- *Requirements Analysis.* During the requirements analysis, the functions and information objects of the application are detected and analyzed using informal description techniques.
- *Conceptual Design.* Based on the output of the requirements analysis the conceptual design produces a first formal description of the database structure and application

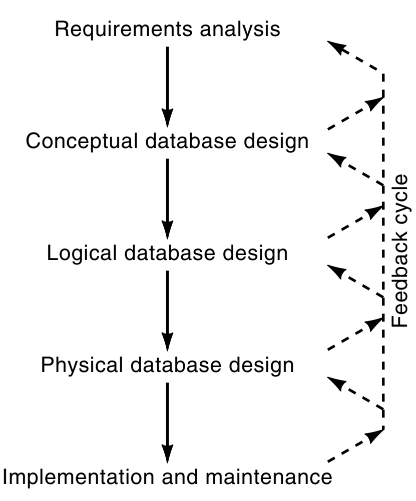

Figure 1. Phases of the database design process.

functions on an abstract implementation-independent level. A typical description model on this level is the entity–relationship (ER) model for specifying the database structure.

- *Logical Database Design.* The logical database design transforms the conceptual schema into the logical database model supported by the intended implementation platform. A typical example for this process is the transformation into the relational model and the normalization of the resulting schema.
- *Physical Database Design.* During the physical database design the logical schema is mapped onto physical database structures for efficient use of the database. A typical method for this phase is the clustering of data on pages and the definition of indexes for efficient retrieval.
- *Implementation and Maintenance.* The last phase is the coding and maintenance of the database schema and the related database transactions on an existing platform.

During the database design process, there will be of course feedback from later phases to the earlier phases. Problems or incomplete specifications may only be detected during transaction realization when they influence, for example, the conceptual design.

Requirements Analysis

The first design phase is the *requirements collection and analysis.* During this phase the expectations of the users and the intended use of the database are analyzed.

For this aim, the parts of the complete information system that will interact with the database are identified and informally specified. Possible sources for the requirements are the following:

- Interviews with representatives of the identified user groups
- Existing documentation of the application areas
- Already existing software solutions in the intended application area
- Legal or organizational requirements for the supported processes

The resulting requirements documents are usually written in an informal style. Graphical presentations support the intuitive semantics of the identified concepts, data items, and workflow processes typically do not use formal techniques. The book by Wieringa (5) gives an overview of popular requirements analysis techniques.

At the end of the classical requirements analysis process, functional requirements are separated from the data requirements. Current proposals aim at avoiding this separation with the use of object-oriented techniques.

Conceptual Database Design

The conceptual model is the first formal model of the database and the connected system functions. Its role can be compared with formal specification techniques in software development.

In the database design process the *conceptual database schema* "real-world" objects. This conceptual database schema is connected to conceptual descriptions of application func-

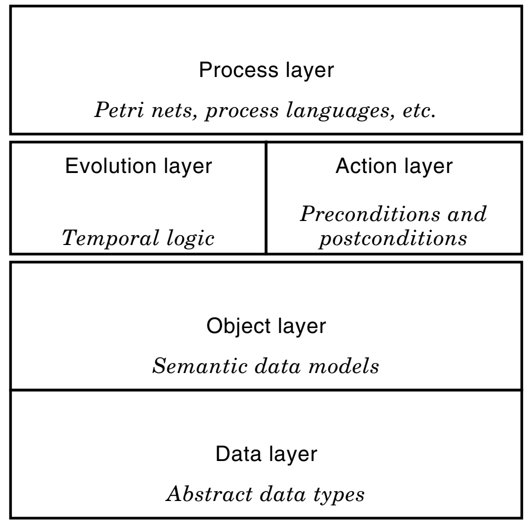

Figure 2. Layers of conceptual model descriptions.

tions and application workflow as discussed in the next section.

Conceptual database models offer several high-level abstraction mechanisms to model the data structures on an abstract level. These models are variants of knowledge representation formalisms and often derivates of the entity-relationship model [see (2)]. Currently, the structural model of object-oriented design approaches is often used as the conceptual database language.

Layered Conceptual Models. A layered approach to designing conceptual database applications was presented in Refs. (6, 7). These approaches aim at capturing all aspects of database application development on an abstract conceptual level, that is, by abstracting from concrete realizations. Such frameworks have to model the database structure as a submodel but also have to capture the application dynamics.

In this subsection we give a short introduction to layered conceptual modeling in order to show the whole design task before focusing on the structural aspect of designing the database itself for the rest of the section.

Overview on Model Layers. A conceptual model of a database application can be structured into several *specification layers* following Refs. 6, 7, 8, 9, and 10. Those layers use specification formalisms building a hierarchy of semantic interpretation structures. At the level of describing database evolution, the strict hierarchy of layers is broken because we have two independent description formalisms building complementary specifications of the same target. Concepts of lower layers are integrated appropriately into upper layers. The hierarchy of layers is shown in Fig. 2.

Data Layer. On the *data layer* we have to describe the state-independent basic data structures. In software specifications these data structures are encapsulated in *abstract data types* together with related functions and predicates on the data elements. These state-independent data structures define the structures of basic data items stored in the database as properties of persistent objects. They are also known as *printable* or *lexical* object types in the database modeling literature [see (10) for a more detailed discussion and a literature overview].

In general, modern implemented database management systems support a small set of standard data types but—in contrast to the area of programming languages— no constructors are offered for building arbitrarily complex data types on top of the standard ones. A specification formalism merging the fields of these both classical disciplines has to offer more powerful specification concepts as done in classical database design. In recent years the development of *extensible* database management systems tries to bridge this gap at the level of database implementations too.

Examples for user-defined data types are geometric data types like `point`, `line`, or `circle` with related operations like `circle_cut` or `distance`. Other examples are enumeration types, data types for engineering applications like vector, as well as types for large unstructured data like bitmap pictures or video sequences.

As specification formalism for abstract data types, we can choose from the well-established description formalisms that can be found in the related software engineering literature. Specification of abstract data types is not specific to database applications, and we will not go into detail here.

An established specification formalism is the *algebraic* specification of abstract data types using equational specification [see (11) for a textbook]. The following example shows a part of the specification of the geometric data type point done in the equational framework.

Example 1 The geometric type point together with related operations can be specified explicitly as follows:

```
DATATYPE point BASED ON real;
SORTS point;
OPERATIONS distance : (point × point): real;
           xcoord, ycoord : (point): real;
           createpoint : (real × real): point;
           add : (point × point): point;
           ...
VARIABLES p,q : point;
          x,y,x1,y1 : real;
EQUATIONS
 x = xcoord(createpoint(x,y));
 y = ycoord(createpoint(x,y));
 distance(createpoint(x,y),createpoint(x1,y1))
   = sqrt((x-x1)*(x-x1) + (y-y1)*(y-y1));
 add(p,q)
   = createpoint(xcoord(p)+xcoord(q),ycoord(p)+
     ycoord(q));
...
```

To support the usual mechanisms in constructing new types from already defined ones, we additionally have a collection of *parameterized data type constructors* like `set` or `list` construction. With each of these constructors a family of operations is associated. For example, the operations `in`, `insert`, and `union` are associated with the `set` constructor (among others). These constructors can be used to build a family of polymorphic types to simplify the use of data types in specifications. The data type constructors are also used for defining the result structures of queries and for the definition of the type of multivalued attributes.

Object Layer. At the *object layer,* the *consistent database states* are described. A database state can be seen as a snapshot of the persistent objects representing the information stored in the database. The modeling of the object layer is done by way of the classical design techniques for database structure using conceptual data models like the ER model, semantic data models, or object-oriented models.

The description of the object layer consists of two parts, the description of the proper database *structure* in terms of a data model and the description of *correct extensions* of this structure definition in terms of integrity constraints.

As mentioned before, we want to describe *collections of persistent objects carrying information.* The information carried by objects is expressed in terms of data-valued object properties (called *attributes*) and relationships between objects. These concepts are the basic modeling concepts offered by the entity-relationship approach (12).

Experiences with modeling complex applications, especially in the area of so-called nonstandard applications like engineering databases, have shown that we need further concepts to support special relationships between persistent objects like ISA or PART_OF relations. These additional concepts originating from the development of *semantic data models* (13,14) can be integrated into the ER approach (4,10,15).

The discussion of *object models* has brought new aspects to the discussion on appropriate modeling constructs, among them inheritance along subclass hierarchies and temporal object identity independent of current attribute values (16). Another interesting extension is to use rules to derive implicitly expressed objects, properties, and relationships.

It should be mentioned here that each schema of a conceptual data model defines the *signature* of a many-sorted predicate logic where the sort symbols are given by the object sorts (and data type sorts, too) and functions and predicates are induced by the attribute and relationship definitions of the schema. This logic is the basis for query formalisms and results in a language for integrity constraints. Another language induced by the object schema is a language for elementary updates (15).

Up to now we have concentrated on the proper structure of our object collections. If we want to express additional restrictions and knowledge from the application area in the object layer specification, we have to state *integrity constraints* restricting the correct database states. Some common integrity constraint patterns are usualy directly supported by specific language features, for example, cardinality constraints on relationships. On the conceptual level, other constraints are formulated in a first-order logic induced by the conceptual schema.

Example 2 The constraint that each employee of a department has to earn less than the manager of her/his department can be formulated as follows:

```
FOR ALL (P : PERSON), (M : MANAGER),
 (D : DEPARTMENT) :
 ( Works_For(P,D) AND D.manager = M
 AND NOT P = PERSON(M) )
 IMPLIES P.salary < M.salary;
```

In this example we have used an explicit conversion of a MANAGER object into a PERSON object along the subtype hierarchy defined by a specialization relationship.

Another way to express additional application semantics is by use of *rules* to derive information from explicitly stored objects. For the modeling of database states, it is common to use model-based semantics because it is appropriate for specifying database states being implemented by concrete interpretations of a data model. Therefore rules are used only in a restricted way, namely to compute derived attributes, objects, and relationships in a determined fashion. A commonly used derivation is the definition of so-called computed or derived attributes by a data-valued function.

There is a close relationship between rules and integrity constraints. If derived information is modeled explicitly on the object layer, the derivation rules can be read as special integrity constraints. On the conceptual level, both views are equivalent and need not to be distinguished. However, the modeling of derivation rules is an important part of the application modeling and should be supported by appropriate language constructs.

Evolution Layer. Until now, we have described the static aspects of database states only. The next specification layer, the *evolution layer,* specifies the *temporal evolution* of the persistent objects. This is done completely without referring to the concrete modification actions changing the stored information. The reference time scale is the causal time induced by the sequence of database modifications.

The semantics domain to be specified is the set of *correct database state sequences.* This is done independently from concrete transactions or application processes. The temporal evolution of the stored information is specified by restricting the *life cycles* of persistent database objects. Such restrictions are called *dynamic* or *temporal* constraints. In other words, we state which long-term evolutions of object (or object combinations) properties and relations are desired. Examples of such long-term dynamic constraints are as follows:

- *Salaries of employees must not decrease.*
- *Airplanes have to be maintained at least once in a year, or at least every 50,000 miles.*
- *Employees have to spend their yearly holidays by May of the following year.*

There are several specification formalisms for such dynamic constraints proposed in the literature:

- *Temporal logic* specifications offer a descriptive formalism for temporal constraints. Their semantics is directly expressed using sequences of predicate logic interpretations, namely of database state sequences.

  Several temporal logic dialects for temporal constraints are proposed in the literature, for example, in Refs. 7, 9, 17, 18, 19, and 20.
- An alternative, more procedural way to express temporal constraints is to use transition automata or simple Petri nets. This technique is, for example, proposed by Ref. 21.

Both approaches are equivalent in the sense that a given specification using one approach can be automatically compiled into a specification using the alternative approach [see (7,22) for the transformation from temporal logic into automata]. This transformation into transition automata can be interpreted also as a transformation into *transitional constraints* restricting local state transitions instead of whole state sequences. As an interesting extension of dynamic constraints, Ref. 23 additionally propoes to distinguish between dynamic constraints and *deontic constraints* separating the correct database sequences and the desired temporal evolutions.

It should be noted that both approaches need a formal semantics of *temporal object identity* because temporal logic formulas or transition automata are formulated locally for single objects changing their properties during database evolution. For example, a PERSON object remains the same object even if all its observable properties are changing [assuming an implicitly given temporal object identity as offered by object-oriented models (24)]. We give two examples for temporal logic constraints.

Example 3 The dynamic constraint that salaries of employees must not decrease can be formulated as follows:

```
FOR ALL (E : EMPLOYEE) (s : integer):
 ALWAYS (E.salary = s IMPLIES
         ALWAYS NOT E.salary < s );
```

The temporal operator ALWAYS denotes a temporal quantification over *all* future states. The first ALWAYS defines the bound subformula as an invariant; that is, the formula must be satisfied for an inserted PERSON object in all future database tail sequence. The inner implication states that if once the salary of an EMPLOYEE is equal to an integer value s, it must be greater or equal to s for all future states (due to the inner quantification by ALWAYS).

Example 4 The second dynamic constraint states that salaries of employees must not decrease while working at the same company—even if she/he has worked for another company in the meanwhile:

```
FOR ALL (E : EMPLOYEE) (s : integer)
 (C : COMPANY):
 ALWAYS ((E.salary = s AND Works_For(E,C))
  IMPLIES ALWAYS ( Works_For(E,C) IMPLIES
   NOT E.salary < s) );
```

The interesting point of the second example is that this constraint implicitly uses historical information, namely the former salaries of persons earned at companies, even if the explicit information that a specific person had worked for a company in the history is not modeled in the object schema directly. The identification and consideration of such additional object structure induced by dynamic constraints is an important part of the conceptual database design process. This problem is discussed in more detail in (25).

Action Layer. In the previous subsection we have presented a specification method to describe database evolutions independently of concrete modification transactions. The *action layer* offers the complementary description of database sequences in terms of correct *database state transitions* by so-called actions.

Actions are schema-specific database updates, namely functions from database states into new correct database states. They are the elementary building blocks of transactions preserving integrity.

Examples of actions are insertion of an employee, or a salary upgrade, while respecting the constraints on employees' salaries and more typically a flight reservation in a travel agency database.

There are several proposals on specification techniques for database actions. Popular specification techniques are used in the behavior part of the OMT- and the UML-approach (cf. 14,26,27). A language proposal combining the structural and specification description into object specifications is `TROLL` (28). Since an action is a function on database states, we can use specification mechanisms for functions on values of a complex structured data type, for example, algebraic specification. However, this approach neglects somehow our more abstract view on database states as interpretation structures of a logic theory. We prefer to use specification formalisms interpreting action specifications as a relation between first-order logic models fitting to the semantic domains used for the evolution layer.

A natural way to describe transitions between interpretation structures is to use *pre- and postconditions.* This descriptive style of action specifications fits well to the use of temporal logic for describing database evolutions. A detailed language proposal independent of a fixed data model and its formal semantics can be found in Ref. 29. A language proposal for an extended ER model is presented in Ref. 10.

Pre- and postconditions are a restricted form of a modal or action logic using explicit logic operators referring to actions. Such specification frameworks are used in Refs. 30, 31, to specify actions using arbitrary modal/action logic formulae.

An example of an action specification using pre- and postconditions is the action `FireEmployee` specified in the following example.

Example 5 The action specification `FireEmployee` removes a person from the database if she or he is not currently a manager of another person:

```
ACTION FireEmployee (person_name : string);
 VARIABLES P : PERSON;
 PRECONDITION P.name = person_name IMPLIES
  NOT EXISTS (PP : PERSON)
   P = PERSON(PP.manager);
 POSTCONDITION NOT EXISTS (P : PERSON)
   P.name = person_name;
```

The object variable `P` is implicitly universally quantified over all currently existing persons.

A specification using pre- and postconditions describes the desired effects of an action only. There are usually several transition functions between database states satisfying such a specification. To capture desired and undesired side effects of state transitions satisfying the specification, we need two implicit rules to choose *minimal correct transitions* as a standard semantics:

- The *frame rule* states that an action effect should be as minimal as possible. The existence of a minimal transition is, however, an undecidable problem, for example, if we have disjunctive postconditions. An elaborate discussion of the frame rule and related problems can be found in Ref. 18. The frame rule forbids undesired side effects of actions ("no junk").
- The *consistency rule* states that each action has to obey the (static and dynamic) integrity constraints. It handles the desired side effects of actions like update propagation.

Both rules work complementally: The consistency rule extends the action specification such as by additional postconditions to guarantee integrity. The frame rule, on the other hand, has to add invariants, guaranteeing that only object modifications can occur that are explicitly enforced by postconditions or by the need of integrity preservation.

Process Layer. In Ref. 10 the four specification layers described until now are identified as being relevant to describe databases as stand-alone components. However, to describe *database applications* as software systems consisting of a database and further components, we have to add an additional layer describing these system components and their interaction in a suitable framework. Moreover this description framework should be compatible to the semantics of the pure database description layers.

At this *process layer* we describe a database application as a *collection of interacting processes.* The database described using the four lower layers is handled as one special persistent process where the actions determine the event alphabet of the process. The database process is purely reactive; actions are triggered from other processes only. This approach is powerful enough to handle distributed applications, formal user modeling, and multiple database applications in the same framework.

Semantically the database process can be described as a linear life cycle over the event alphabet together with an observation function mapping prefixes of the life cycle into database states. This semantics is conform with the semantic models used for the pure database specification, namely with linear sequences of database states.

The database process is only one among others that together build the database application. The application consists of several independent software components communicating by sending and receiving messages. Examples for such components are

- *interaction interfaces* communicating with users using an application-specific communication protocol
- long-term *engineering transactions* performing complex activities in cooperation with several users and databases
- other integrated *software systems*
- several *data and object bases* possibly implemented using different DBMSs and data models

The formal specification of interacting processes is still a vivid field of software engineering research. Languages are

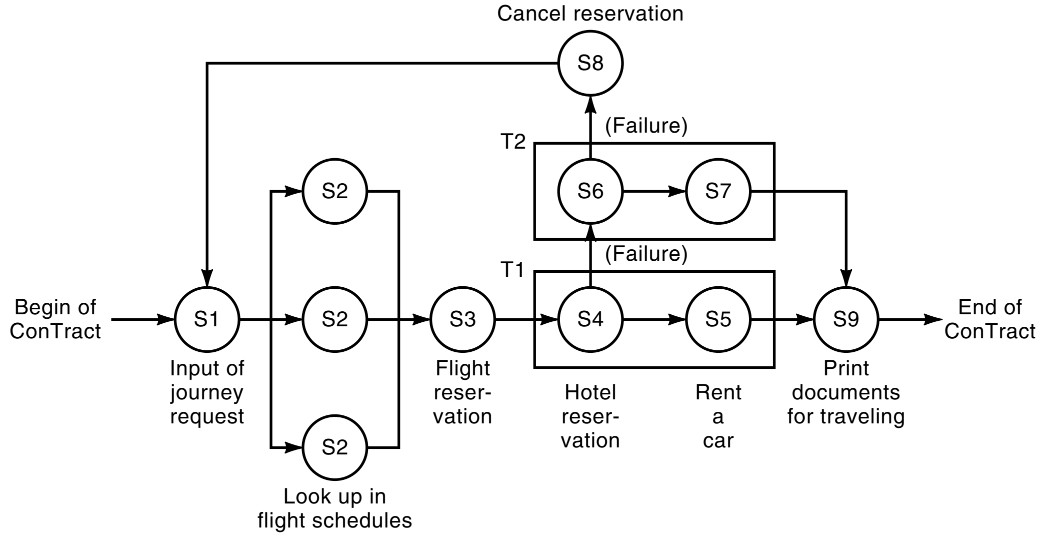

Figure 3. Example process using the CONTRACT notation.

proposed in the area of engineering transactions as well as in the area of *workflow management.*

A typical approach from this area is the CONTRACT model described in Ref. 32. Figure 3 shows a process description in the CONTRACT model and gives an impression of the necessary modeling primitives. For example, `S4` and `S5` belong to one atomic transaction `T1` which is part of the larger process.

Abstraction Principles. On the conceptual level, data models should support the four abstraction principles known from information modeling:

- *Classification.* Objects having the same set of properties are classified into classes.
- *Specialization / Generalization.* A class is a specialization of another class if the subclass inherits the properties of the superclass and the population (extension) of the subclass is a subset of the population of the superclass.
- *Aggregation.* Objects are composed from other objects.
- *Grouping.* A group of objects builds conceptually a new composed object. A typical example is the `Team` as a set of persons.

These basic principles lead to several modeling principles which should be supported by a suitable conceptual database model. The following list of basic modeling concepts should be supported by an appropriate language for describing the conceptual object layer:

- The first modeling primitive is the concept of abstract entities called *objects* or *entities.* Objects are abstract in the sense that they can only be observed by the values of their properties. Properties are data- or object-valued functions for objects and are called *attributes.*
- Objects with the same set of properties can be grouped into *object types.* Examples for object types are the types `PERSON` or `COMPANY` with corresponding data-valued attributes, for example, `name` of type `string` or `location` of type `point`. An example for an object-valued attribute would be the attribute manager of type `PERSON` associated with an object type `DEPARTMENT`. Object-valued attributes can often be adequately modeled by functional relationships or complex object construction, too.
- Objects are abstract entities observable by their attributes only. To distinguish different objects having the same properties, we have to introduce an *object identification mechanism* (24). Object identity can be specified explicitly by *key functions,* namely by choosing some object properties as object "separators" inside one object type. An alternative solution is to introduce an *implicit object identity* as a property of the data model as it is done in some object-oriented approaches (33,34).
- With an object type we associate the class of currently existing objects of this type. Usually these classes are disjoint. But there are several interesting cases where this intuitively is not the case. In these cases we talk about *type* or *class construction* by generalization, specialization, or partition. Constructed classes inherit the identification from their base types. For the formal semantics of type constructions, see Ref. 35.
  1. *Specialization* is used to build a subclass hierarchy, for example, starting with the type `PERSON` and defining `MANAGER` and `PATIENT` as independent subclasses of `PERSON`. Specialization induces a subset relation (ISA hierarchy) between the current object class populations and a inheritance of properties of the input type.
  2. *Partition* is a special case of specialization where a class is partitioned into several disjoint subclasses. An example is the partition of `PERSON` into `WOMAN` and `MAN`.
  3. *Generalization* works the other way round—several input classes are generalized into a new class. An example is the generalization of `PERSON` and `COMPANY` into `LEGAL_PERSON`.
- Another modeling concept known from the ER approach are arbitrary *relationships* between objects, for example,

the relationship `Works_For` between persons and companies. There are several interesting special relations between objects that should be explicitly modeled in a specification. Examples are the already mentioned ISA relation or functional relationships (being equivalent to object-valued attributes in the binary case).

- Another special relation which should be made explicit is the PART_OF relation leading to the notion of *complex objects.* In particular in engineering applications, the appropriate definition of complex objects is a mandatory feature of a conceptual data model (36,37). There are several properties associated with the notion of complex objects, among them weak object types (a component object cannot exist outside its aggregate object), the distinction between disjoint and nondisjoint complex objects, and the problem of update propagation for complex objects.

Modern conceptual database languages support most of these modeling principles.

Conceptual Database Models. The previous subsection listed modeling constructs important for the conceptual design of database structures. One can choose from a multitude of conceptual database models for these design tasks. The most important directions are the following:

- *ER Models and Extended ER Models.* Based on the basic ER model presented by Chen in Ref. 12, several extended ER models are proposed as conceptual database models. The basic ER model has three modeling primitives: entities, relationships, and attributes. Extended ER models add concepts for specialization and generalization (2,4,10,15).
- *SDM (Semantic Data Models).* Semantic data models are based on the presented abstraction concepts. Usually they support functions, aggregation, and specialization hierarchies (38,39,40).
- *OOD (Object-Oriented Design Models).* Object-oriented design models combine the concepts of semantic database models with concepts from object-oriented programming languages. Popular models are OMT (41) and OOD (42). These models are currently combined toward the Unified Modeling Language (UML) to become the future standard of object-oriented design notations (27).

Besides these closely related main stream models, some other frameworks are used for conceptual modeling based on other paradigms. Examples are functional database models (43,44) and binary-relationship object models, also known as object-role models [e.g., NIAM (45)]

View Integration. The aim of the conceptual design phase is to produce an *integrated* abstract model of the complete database. As a result of the requirements analysis, the starting points are the different and usually inconsistent views of different user groups on the application data.

Therefore the process of *view integration* plays a central role in conceptual design. There are several phases of the view integration process:

1. View Modeling: The different perspectives identified during the requirements collection are modeled using the conceptual database design model.
2. View Analysis: These views are analyzed to detect synonyms and homonyms, to identify structural conflicts, and to find corresponding elements. This process is very similar to the preintegration (homogenization) process in database federation, which will be discussed in detail later in this article.
3. View Integration: Based on the results of the view analysis, an integrated database schema is constructed.

The process of view integration is very similar to the process of databases integration described in the section entitled "Schema Merging." In contrast to view integration, the process of database integration has to analyze existing databases and may have to preserve them in a federated environment.

Logical Database Design

Mapping to Logical Database Models. The first phase of a logical database design is the transformation of the conceptual schema into the logical database model. This transformation can be done "by hand" or using a database design tool. As an example, we will discuss the mapping from ER to the relational model.

For the transformation process, we can state a quality property of the mapping: *Capacity Preservation*—Both schemata are able to store exactly the same database contents.

The ER model supports the concepts of entity, relationship, and attributes. Key attributes denote identifying properties of entities. In contrast, the relational model supports only relations (with keys) and attributes. There is no explicit relationship construct; however, foreign keys can manage interrelation relationships.

Table 1 [taken from (46)] summarizes the mapping from ER to the relational model. As shown in the table, the mapping of attributes and entities to relations is straightforward. The mapping of relationship types, however, has to consider the different types of relationships available in the ER model. Especially cardinalities of binary relationships influence the choice of key attributes for the relation derived from an ER relationship.

During the mapping process already some additional optimizations are possible. For example, relations can be merged

Table 1. Mapping of ER Schemata onto Relational Ones

ER Concept	Mapped onto Relational Contruct
Entity type E_i	Relation R_i
Attributes of E_i	Attributes of R_i
Key P_i	Primary key P_i
Relationship type RS_j	Relation schema RS_j with attributes P_1, P_2
Attributes of RS_j	Additional attributes of RS_j
$1:n$	P_2 primary key of RS_j
$1:1$	P_1 and P_2 both keys of RS_j
$m:n$	$P_1 \cup P_2$ primary key of RS_j
IsA relationship	R_1 has additional key P_2

Note: E_1, E_2: entities participating in relationship RS_j; P_1, P_2: primary keys of E_1, E_2; $1:n$ relationship: E_2 is on the n-side; IsA relationship: E_1 is specialized entity type.

depending on the cardinality and optinality of the mapped ER relationship.

Relational Database Design. Based on the relational schema resulting from the mapping from the conceptual schema, further optimizations and normalization are possible. This process is especially important if the conceptual phase is skipped and database designers model directly in the logical database model.

Relational database design is an important area of database theory in itself. Several books, among them Refs. 1 and 47, deal with this area in detail. We will present very shortly some basic concepts that have found their way into practical database design.

One major part of relational database design is the theory of functional dependencies and resulting normal forms:

- A *functional dependency (FD)* describe dependencies between attribute values in a relation. An FD is denoted as follows:

  `ISBN` $\rightarrow$ `Title, Publisher`

  This FD specifies that two rows of a relation having the same value for `ISBN` should also have the same value for the attributes `Title` and `Publisher`. The semantics may be formalized using the following formula:

  $$X \rightarrow Y \equiv \forall t_1, t_2 \in r : t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$$

  This formalization says that for two rows (= tuples) of a concrete relation r, whenever they have the same values for the X attributes, they have to have the same values for the Y attributes too.
- A *key* of a relation is a (minimal) set K of attributes, where $K \rightarrow R$ for R being all attributes of a relation. In other words, a key identifies the rows of a relation uniquely.
- There are rules for manipulating FDs. The *closure* of a set $\mathscr{F}$ of functional dependencies is the set of all FD which are logical consequences of $\mathscr{F}$. Logical consequence for functional dependencies is efficiently computable.

In general dependency theory, several other dependency classes are important. Among them are multi-valued dependencies, inclusion and exclusion dependencies, and joint dependencies. We will not detail this area but refer to the relevant literature.

Normal Forms and Normalization. One popular application of functional dependencies is the *normalization* of relational schemata. The aim of normalization is to remove redundant storage of attributes from a relational database. This is done by analyzing the functional dependencies and afterward constructing a database schema, where all functional dependencies are enforced by key constraints of relations.

- The *first normal form (1NF)* characterizes the relational model as having only atomic values for attributes (excluding repeating groups for attributes).
- The *second normal form (2NF)* excludes relations, where some nonkey attributes are partially dependent on a composed key. Since 2NF is implied by 3NF, it is enough to enforce 3NF.
- The *third normal form (3NF)* excludes relations, where a nonkey attribute is transitively dependent on a key. These transitive dependent attributes should be moved to a separate relation avoiding redundancy.
- The *Boyce–Codd normal form (BCNF)* generalizes 3NF to dependencies inside a composed key.

There are efficient algorithms that enforce 3NF and respect the information capacity of the schema. The BCNF removes more redundancy than 3NF, but it cannot be guaranteed that the normalized schema will enforce all constraints expressed as functional dependencies.

If we take more kinds of dependencies into account, more normal forms can be defined that eliminate further sources of redundancy but are not expressible using functional dependencies alone.

Database Definition: Coding in SQL-DDL. The last part of the logical database design is the mapping of the logical description onto a data definition language. An example is the coding of a relational database structure using the standardized SQL language (48).

Important parts of this coding are the following steps:

- Choice of the correct data types for the attributes
- Choice and definition of primary keys
- Definition of uniqueness constraints for the remaining keys
- Definition of referential integrity constraints resulting from the mapping of ER relationships
- Formulation of suitable check constraints for attributes
- Transformation of complex constraints into triggers

Physical Database Design

The logical database schema still abstracts from the internal realization of the data. Modern database systems support several data structures and storage management techniques for efficient management of large databases.

The physical design step has to be system-specific because commercial database vendors support different techniques for optimizing the internal structure of databases.

Typical methods to optimize the internal structure of a relational database are the following:

- The step of *denormalization* reverses the normalization step of the logical design. The motivation is to introduce redundant storage of data to fasten specific kinds of queries. Typical denormalization steps are to store frequently occurring joins as materialized relations or to realize a specialization relationship by adding possibly null-valued attributes of the specialized class to the base class. Some books (e.g., 49) on database design present some typical patterns for denormalization for relational databases.
- The definition of *indexes* allows specification of efficient access structures for attributes or attribute combinations. Indexes are typically variants of B-tree structures, but some systems also support hash-based indexes or bit-

map-indexes for data warehouse applications. Data access structures for indexes are part of the realization of DBMS and therefore not part of the database design phase. Typical index structures are presented in most textbooks on database systems (e.g., 50,51,52). A good survey on common algorithms can be found in A. L. Tharp's *File Organization and Processing* (53). The optimal choice of indexes for a given application profile is part of the process of database tuning (54) and an important phase of the physical database design.

- The *table organization* defines the way relations are stored. Besides storing the rows of a table sequentially in operating system blocks, one may choose to store rows of a table sorted, in a hash order or in a tree structure. As for indexes, the table organization is not covered by the SQL standard and it therefore differs for commercial databases. Again, typical textbooks on database systems (50,51,52) give detailed introductions in this area.
- The *clustering* of database objects aims at storing rows from different relations in such a way that database items commonly retrieved together in queries are located on the same file system blocks. Clustering can especially improve the execution of join queries. Some commercial DBMS like Oracle8 support different clustering methods. Again, the methods and language constructs are not part of the SQL standard. The basic principles are also handled in the above mentioned database texts.
- For distributed and parallel databases the *partition, allocation,* and *replication* of database relations are important steps in optimizing their internal structure. *Partitioning* splits a relation into several parts to be distributed on several nodes. The *allocation* establishes the relation between partitions and actual nodes, whereas a partition can be *replicated* onto several nodes as part of the allocation process. All these steps can be used to reach a higher performance in a distributed environment. Textbooks on distributed databases (e.g., 55) give detailed descriptions of these design processes.

For other database models additional techniques are supported, for example, specific indexes for supporting path queries in object-oriented databases.

Implementation and Maintenance

The last phase of the design process is the implementation and maintenance of the database application. Besides the concrete definition of database structures, this phase also contains the coding of database transactions.

The data must be loaded into the database. If data are imported from other systems, they may be reformatted using conversion routines. The maintenance of the database is one of the most time-consuming steps of database design. A database may have a lifetime of decades, and both the software environment and the application requirements will change several times during its lifetime. The changes will affect all levels of database definitions, and it is very important that changes are documented on all design document levels to allow further maintenance even after several years.

A problem often occurring in maintaining a database is schema evolution. Changing requirements of the applications may require changes of the database schema. However, there are a lot of problems caused by schema evolution. For instance, Refs. 56, 57, 58, 59, 60 consider those problems for object-oriented databases and propose different approaches to overcome several of these problems.

Another problem that may arie in connection with schema evolution is database (schema) versioning. An evolution of a database schema may lead to the necessity of having several versions of the database schema on hand. In general, schema evolution produces new versions of an existing database schema. A general overview on versioning and configuration management can be found in Ref. 61. Several models for versioning in object-oriented database systems are discussed in Refs. 62, 63. However, database and database schema versioning should be mainly considered as a matter of conceptual and logical database design. Planning an adequate schema versioning concept during the early phases of database design may help improve database maintenance in case of requirements for schema evolution.

DATABASE INTEGRATION AND INTEROPERATION

Interoperability of databases (or database systems) plays a more and more important role in today's development of information systems within companies and other organizations. Facing the fact that during the last decades a large number of different information systems have been developed and a huge amount of data is currently stored in numerous and often heterogeneous databases, it becomes clear that the development of a completely new information system covering all aspects relevant for an organization is usually impossible.

Preserving the investments made over years as well as guaranteeing the smooth continuation of everyday business are only two essential reasons for taking care of existing systems within organizations. Nevertheless, new requirements ask for interoperability of existing systems.

In this section we focus on multi-database systems and federated database systems as basic architectures for database interoperability. Of course there are other possible architectures for implementing interoperability among database systems. Due to the fact that this article is dedicated to the general theme of "database design," those architectures having inherent design-relevant aspects come to the fore of our discussion. Although we mainly consider the structural part of databases or information systems, the role of the behavioral part is not to be underestimated. The intended behavior of database objects and database applications provides a lot of information that has to be respected during database integration. Another important aspect is the integration of behavior. Because there are only very few and preliminary results concerning behavior integration so far (see, for instance, Refs. 64, 65), we here do not consider this aspect in more detail.

In the following sections, we first discuss basic properties that are often used for distinguishing different database architectures. Next we present three basic architectures for multi-database systems and federated database systems. Finally, we discuss major requirements for design approaches in this context.

Basic Characteristics

For characterizing database architectures the following three properties are frequently used:

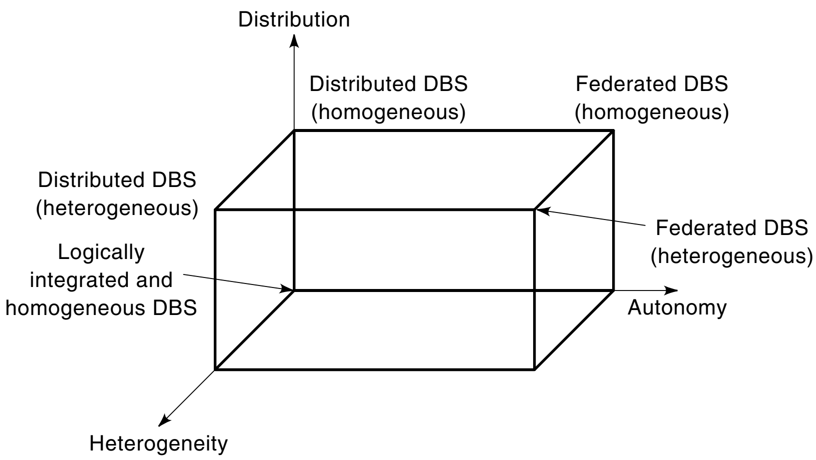

Figure 4. Classification of database architectures based on distribution, autonomy, and heterogeneity, Refs. (66,67).

- Distribution
- Autonomy
- Heterogeneity

For instance, Refs. 66 and 67 present a classification based on these properties. Figure 4 depicts this classification and shows how the most important database architectures occurring in practice fit into this classification.

Distribution. The property of *distribution* refers to the storage of data. A distribution of data is given in case the data are stored at different sites. Distributed storage of data may be for either of two reasons: The distribution of data is intended, or the distribution of data has occurred accidentally.

A typical example for intended distribution is a *distributed database* (68,69,70). A distributed database is based on a common database schema for which a reasonable *partition* has been fixed. Following this partition the database is split into parts being stored at different sites. In a narrower sense a partition means that no data are stored redundantly at several sites. For allowing a more efficient query processing or for improving the availability of data, a controlled kind of redundancy (called *replication*) is often introduced.

Besides the intended distribution of data by means of distributed database systems, we frequently find an accidental and usually uncontrolled distribution of data. Within organizations several information systems have usually been developed independently for different purposes. Thereby different database management systems as well as other data management systems have been introduced into the same organization. Each of these systems manages a certain portion of data. Usually the corresponding database schemata have been designed independently, and no common database schema exists. In consequence uniform access to all the data is currently not possible. Furthermore, consistency for all the data cannot be checked. This is a typical situation in which the construction of a federated database system incorporating the existing systems is worth considering.

Autonomy. The notion of *autonomy* has several facets that play an important role in the context of federated database or multi-database systems. In particular, we distinguish the following three aspects of autonomy (71):

- Design autonomy
- Communication autonomy
- Execution autonomy

Design Autonomy. Implicit in complete design autonomy are the following characteristics:

- The databases of the component systems have been designed independently of each other.
- Changing the local database schemata cannot be required for building a federation.
- A global system (e.g., a federation layer for uniform access) also cannot cause changes in the local database schemata later on.

In principle, design autonomy w.r.t. the component databases further means that a designer of a component database may change his or her local database schema without restriction. It is quite obvious that design autonomy must be limited to a certain degree in allowing the global system to have such functionalities like global integrity control.

Communication Autonomy. We speak of communication autonomy in cases where a database system can be decided independently of other systems with which the system communicates. This kind of decision is usually made by the database administrator. An additional aspect of communication autonomy is that the decision to join a federation or to leave a federation can be made independently as well.

Communication autonomy is particularly important for architectures in which the component systems have to negotiate with each other about access to data. In other architectures only the communication with a global component (e.g., a federation layer) is of great importance.

Execution Autonomy. The notion of execution autonomy covers the question whether a component system can independently decide on the execution of local application programs as well as on the processing of queries and manipulation operations. Execution autonomy implies that a federation layer or a component system cannot, for instance, force another component system to execute or not to execute certain application programs. Furthermore the component system is independent w.r.t. its decision on execution order of local transactions.

Heterogeneity. Heterogeneity can occur on different levels. There are *system-dependent heterogeneities* that occur when we federate or integrate different database systems. For integrating database schemata the resolution of *schematic heterogeneities* is important. Schematic heterogeneities can often be found as differences between local schemata. For integrating given schemata correctly, these differences must be found (the possible kinds of schematic conflicts are surveyed in Section 3.4). To a certain extent schematic heterogeneities result from heterogeneities on the system level. Beside this, a lack of common understanding of the meaning and the usage of data can be a source of schematic heterogeneities. Another kind of heterogeneity is *data heterogeneity*. In the following we consider the different kinds of heterogeneities in more detail.

System-Dependent Heterogeneity. Database systems can be heterogeneous with regard to a large number of aspects. Examples for such aspects are as follows:

- Data model (or database model)
- Query language and database programming language
- Query processing and optimization
- Transaction processing
- Mechanisms for integrity control

Here we mainly focus on aspects that are relevant from a database design point of view.

The first aspect is the heterogeneity of data(base) models. In organizations we often have to face the situation that different database systems were purchased over the course of time. Thereby database systems may be comprised of hierarchical database models, network models, relational models, object-oriented database models, and any number of other models.

The problems caused by such heterogeneous databases are due to the fact that different data models offer different sets of modeling concepts for describing the universe of discourse. Obviously this implies that we are usually faced with quite different database schemata—even in the database schemata that describe the same universe of discourse. Figure 5 gives an example of two different database models describing the same real-world aspect (the schema on the left-hand side is based on an object-oriented model or on an extended entity-relationship-model, whereas on the right-hand side a relational description is given).

While the heterogeneity on the data model level can be overcome by transforming the local database schemata into one common data model, such a transformation usually does not resolve all problems caused by *data model heterogeneity*. There are schematic heterogeneities caused by the modeling concepts offered by different data models. We describe these schematic heterogeneities below, and in addition a classification of schematic conflicts occurring during schema integration is given in the section entitled "Classification of Schematic Conflicts."

Another source of heterogeneity can be found in the use of integrity constraints in modeling and in their support by existing database systems. Depending on the data model, certain kinds of integrity constraints do not need to be expressed explicitly because they are already inherent in the modeling concepts of the data model. All other kinds of constraints must be expressed explicitly. Nevertheless, there are rather great differences w.r.t. the support of explicit constraints by existing database systems. For instance, the current standard for the relational database language SQL [SQL-92 (48)] provides a variety of means for expressing explicit integrity constraints. However, existing relational database systems do not support everything that is described in the standard. This holds in particular for older releases of these systems being still in use.

Other system-dependent heterogeneities often refer to query languages. While it may be obvious that there are different query languages coming with different data models, we sometimes find in practice that there are different query languages or different versions (dialects) of one query languages used for the same data model. Taking the relational model as an example, we find SQL to be the query language for almost all existing systems. However, there are still some "legacy systems" having other relational query languages like QUEL (4). There are also differences between systems offering SQL as the query language. Then there are not only different standards for SQL fixed over time (SQL-89, SQL-92) but also even

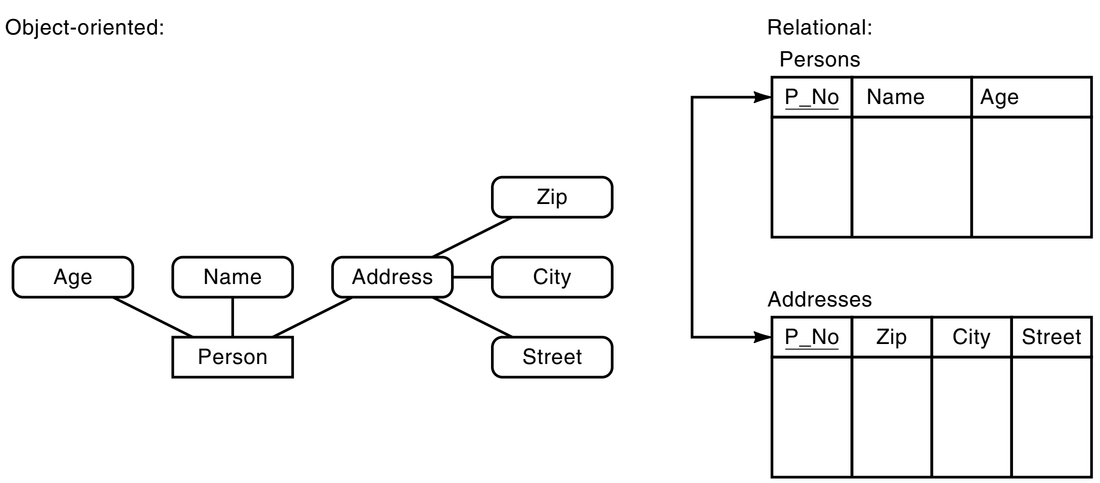

Figure 5. Using heterogeneous data models.

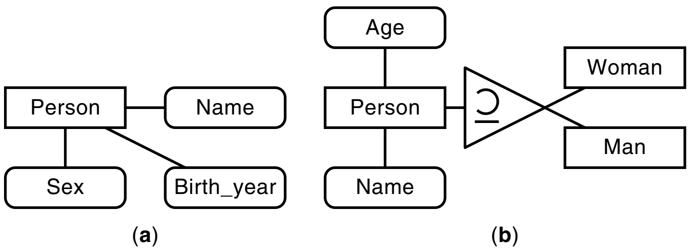

Figure 6. Heterogeneous modeling.

different levels of "compatibility" such as are defined in the current SQL standard.

Schematic Heterogeneity. Having covered system-dependent heterogeneities, we can now focus on the schema level. Schematic heterogeneity can occur in manifold ways (see the classification of schematic conflicts later in this article). Here we give a basic idea of the different origins and manifestations of schematic heterogeneity.

Object-oriented models, for instance, offer a concept of specialization that is not available in the relational model. As a consequence, missing modeling concepts must be simulated by means of other concepts. Unfortunately, there is in general no uniquely determined way of simulating a missing concept. This is due to the fact that there usually exist several ways to model a real-world fact within a single data model. Because of design autonomy we cannot exclude these different possibilities of modeling.

Figure 6 depicts a simple example of heterogeneous modeling using the same data model. Parts (a) and (b) of the figure represent the same real-world fact within the same data model [here an extended entity-relationship model (10) with specialization]. Note that in part (a) the attribute `sex` allows the system to distinguish persons by their sex, but there are two subclasses `woman` and `man` used for the same purpose in part (b).

This example shows that a database designer can have several possibilities for modeling the same real-world facts. If we want to integrate database schemata designed by different persons (or even by the same person at different moments), we must seriously take into account this heterogeneity (which is often called *structural heterogeneity* as well).

In general, it is not very difficult to detect such differences and to find a way to resolve them. However, it is much more difficult to detect and resolve another form of heterogeneity, sometimes called *semantic heterogeneity.* This kind of heterogeneity results from the fact that there is often no common understanding and usage of data stored redundantly in several systems or of data which are in some way related. In order to give an impression of this particular problem, we now consider some simple examples.

If we, for instance, want to integrate two databases in which prices of products are stored, we cannot decide what the relationship between these prices is without having additional knowledge. We first have to find out whether these prices are given w.r.t. the same currency. Then we need to know the current rate of exchange for the currencies used. Besides this currency problem the prices stored in the two databases may differ from each other because in one database the value-added tax (VAT) is included in prices, whereas in the other database the VAT is excluded. Often such differences can only be detected by inspecting the way the database applications use the data.

Different precisions in the representation of numerical values, which can also be considered as a kind of schematic heterogeneity problem, complicate the comparability of values. In general, it is not possible to decide whether two values stored in different databases represent the same value in the real world. On the surface we may see two equal values, but this equality may be due to the fact that one database had a restricted precision that caused the value to be rounded off.

Beside the problem of different precisions for numerical values, which is usually due to design autonomy, we frequently have to face another problem. Even when values can be stored with the same precision in different databases, we often cannot decide on actual equality of two values stored in existing databases. The reason is that application programs (or users) do not always store values with maximal precision. Values are rounded off for convenience or because of laziness. In certain cases values are only estimated or guessed because no precise values are available. In effect we always have to be extremely careful in comparing two values.

To a certain extent the examples we just gave can be considered as special kinds of data heterogeneity.

Data Heterogeneity. A heterogeneity often neglected that unfortunately occurs almost always in practice concerns the data. Heterogeneity of data can occur even if all other kinds of heterogeneities are not present or have already been resolved.

Figure 7 shows an example of data heterogeneity. The two databases considered correspond completely with regard to data model and database schema. Differences in correct data values can result, for instance, from different conventions for putting properties down in writing or from using synonyms or words with similar meanings.

In the example there are the different conventions for writing names of persons whereby the order of first name and last name is different. The terms Poet and Writer could be interpreted as synonymous, but there could be a slight intentional difference in the meaning of these terms. Then misspellings of words as well as typing errors may lead to further undesired differences. Yet another problem frequently occurring in practice is that databases can contain obsolete data.

Obviously these kinds of heterogeneities cannot be easily separated from each other. System-dependent heterogeneity frequently causes schematic heterogeneity. And certain forms of schematic heterogeneity may result in data heterogeneity as well.

Persons

Name	Birth_year	Profession
Zuse, Konrad	1902	Scientist
Wolff, Christa	1928	Poet
…	…	…

Persons

Name	Birth_year	Profession
Konrad Zuse	02	Scientist
Christa Wolff	28	Writer
…	…	…

Figure 7. Heterogeneous data.

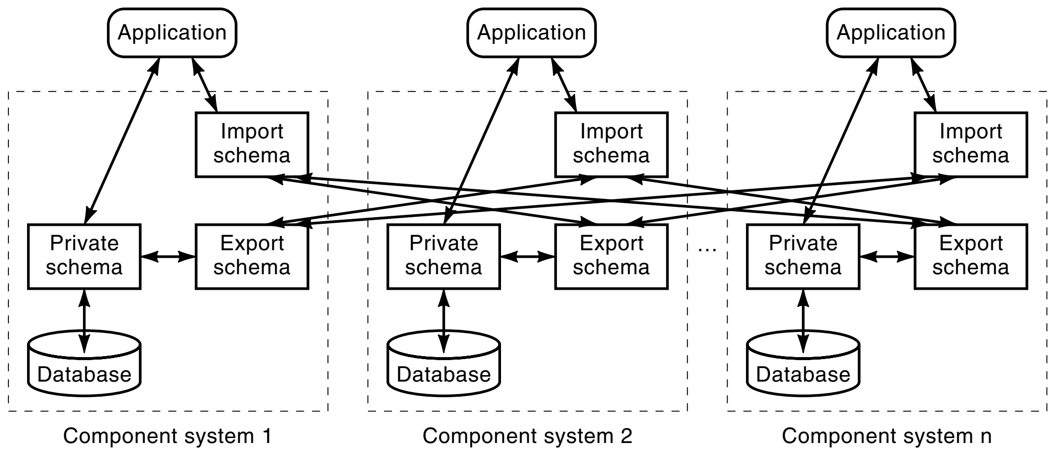

Figure 8. Import/export-schema architecture, Ref. (72).

Architectures

Among the architectures for interoperable database systems, there are three examples frequently referred to in the literature. In the following we describe the basic properties of these architectures.

Import/Export-Schema Architecture. One of the very first architectures proposed for database interoperation is the import/export-schema architecture (72). In this architecture, depicted in Figure 8, we distinguish three different kinds of schemata:

Private Schema. The usual local conceptual schema for all data stored and managed by a component system.

Export Schema. Each component system offers an export schema describing the data that may be accessed by other systems. This description includes access rights defining which other system may access which portion of the data. In this way the access to the local data can be restricted and controlled.

Import Schema. By description of the data imported from other systems a component system can give its application programs access to data stored at other component databases. An import schema gives an integrated view of the export schemata provided by other component systems. However, a real schema integration is not required in this architecture.

For this architecture it is assumed that the component systems negotiate with each other about the access to their export schemata. In this way a system can obtain the access rights to certain portions of data stored in another component database.

Applications running at a certain site can access the data offered by the corresponding component system at that site. Applications have access to two schemata of their component system, its private schema and its import schema. In using two different schemata, the integration of the data can be realized within the applications. Then the responsibility for adequacy and correctness of an integration is given to the application programmer or to the user. In general, there is no a priori integration a user or application programmer can rely on.

The import/export-schema architecture is often used as a basic architecture for loosely coupled federated database systems because it gives full autonomy to the component systems.

Multi-Database Architecture. The multi-database architecture (73) is often used for accessing several databases having the same database model, in particular, for relational databases. Nevertheless, the property that the component system have the same local data model is not a necessary requirement.

In contrast to the import/export-schema architecture, a negotiation between component systems does not take place. For accessing data from several component databases, a multi-database language is provided to users and application programs. Examples for relational multi-database languages are MSQL (73,74) and SchemaSQL (75), which extend SQL conceptually by querying multiple databases within one query.

In the multi-database architecture (see Fig. 9) we distinguish five kinds of schemata:

Physical Schema. In this architecture the physical schema is the usual internal schema of a component database.

Internal Logical Schema. This schema is the usual conceptual schema of a component database.

Conceptual Schema. This schema can be considered as an external schema of a component system defining the part of the internal logical schema accessible from the multi-database layer. The conceptual schemata are described by means of the common global data model. If a component system has a different local data model, a translation of its internal logical schema (expressed in the local data model) into the global data model is required. If no data model translation is needed and all data described by the internal logical schema are intended to be available at the multi-database layer, the conceptual schema and the internal logical schema are the same. Then we have an explicit conceptual schema.

External Schema. Superposed on the conceptual schemata of the component systems are external schemata de-

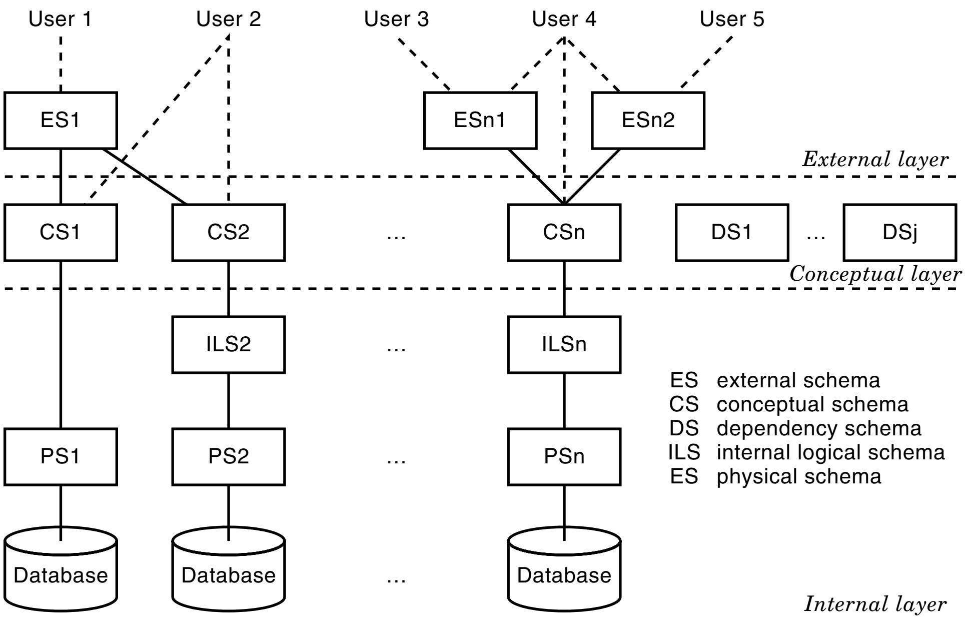

Figure 9. Multidatabase architecture, Ref. (73).

fined by the user or application programmer. The external schema usually includes one or more conceptual schemata. By means of external schemata the data stemming from the component databases can be filtered, restructured, and integrated according to the personal needs or preferences of the user. In order to define these views, a multi-database language is needed.

Dependency Schema. Interdatabase dependencies and additional global integrity constraints dependency are defined by these schemata making global integrity checking and enforcement possible.

Although this architecture is quite different from the import/export-schema architecture described before, the responsibility for integrating data stored in different component databases is given to the users and application programmers. Nevertheless, this architecture requires a common global data model and a multidatabase language as a means for users to access different databases.

Five-Level-Schema Architecture. The third architecture that must be considered is the 5-level-schema architecture (50). In this architecture (Fig. 10) we distinguish five different kinds of schemata:

Local Schema. The local conceptual schema of a component system is here called a local schema. Hence a local schema is expressed in the local data model of the corresponding component system.

Component Schema. In order to overcome the heterogeneity w.r.t. data models, the local schemata are translated into a common global data model. As a result we obtain component schemata. If a component system already uses the global data model as local data model, the local schema and component schema are the same.

Export Schema. Due to the fact that a component schema still describes all data stored in the component database, an export schema can be defined for restricting the parts of the component schema, and thereby the parts of the component database, that can be accessed by global applications. If all data are to be exported, no separate export schema is needed.

Federated Schema. The federated schema (also called *integrated schema* or *global schema*) provides an integrated view on all export schemata given for the component systems participating in a federation. The major emphasis is given to integration. This means that any redundancy found in the export schemata is removed in the federated schema. Furthermore structural differences and other conflicts between export schemata are resolved. The federated schema is the conceptual schema of the federation. It hides the distribution of data as well as all heterogeneities like different local data models and different ways of modeling or structuring data.

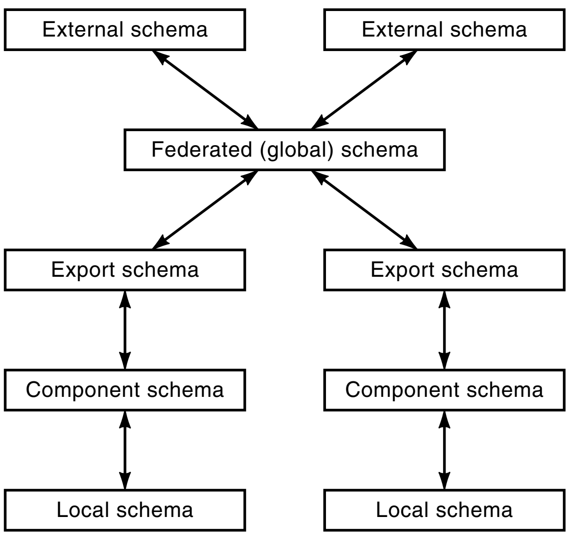

Figure 10. Five-level-schema architecture, Ref. (71).

External Schema. Like external schemata in the traditional 3-level-schema architecture, external schemata are specific views on the conceptual schema (here on the federated schema) for certain applications or users.

The main property of this architecture in comparison with the two described before is that it provides a federated schema. The users and application programmers can rely on that federated schema. The designer of the federated schema is responsible for its adequacy and correctness. A federated schema has to fulfill several criteria, such as described in the next section.

Requirements for Integrated Schemata

When we integrate the given local schemata into a federated schema, several requirements must be taken into account. Besides being important for building a federated schema in the 5-level-schema architecture, these requirements extend to the import/export-schema architecture and to multi-database architecture as well. The user or application programmer is responsible for the quality of the integration he (she) is making for himself (herself). Hence, the same criteria apply.

Following Ref. 76, there are four major criteria for schema integration:

Completeness. The integrated schema must contain all schema elements given in at least one of the local schemata. This means that there must be no loss of information contained in local schemata.

Correctness. For each element in the integrated schema, there must exist a corresponding (semantically equivalent) element in one of the local schemata. There must not exist invented schema elements in the integrated schema. Due to the fact that the original database schemata were isolated, there is one exception. During the integration process we may have found interschema dependencies which cannot be expressed in a single local schema. For these interschema dependencies we may add corresponding elements into the integrated schema. Of course these additions must be consistent with the information adapted from the local schemata.

Minimality. Each real-world concept modeled in several local schemata may only be represented once in the integrated schema. Redundancy on the schema level must be avoided.

Understandability. The integrated schema should be understandable to global users.

The last criterion is the most difficult one because there is obviously no way to check it formally. It is a very subjective property. For instance, global users who are used to a certain representation of their application world, because they have used one of the local systems for many years, may have problems in understanding a integrated schema if the part they already know is represented in a completely different way. For those users understandability goes along with similarity to the original local schemata.

Classification of Schematic Conflicts

A large number of classifications for schematic conflicts can be found in the literature. Here we follow the classification given in Ref. 77 where four classes of conflict are identified. Due to the fact that the class of *heterogeneity conflicts* described in Ref. 77 mainly refers to system-dependent heterogeneity (see Subsection 3.1.3) we here consider the remaining three classes:

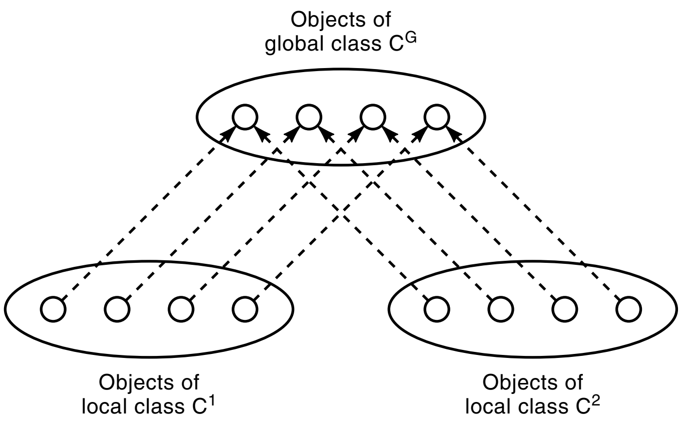

Figure 11. Semantically equivalent class extensions.

- Semantic conflicts
- Description conflicts
- Structural conflicts

There is some overlap between these classes. Combinations of different kinds of conflicts usually occur together because there are some causal relationships between them.

Semantic Conflicts. During the integration of database schemata, we have to deal with semantically overlapping universes of discourse. As a consequence there are in a local schema elements that correspond to schema elements in another local schema. In particular, there are corresponding classes (or relations). However, they often do not represent exactly the same set of real-world objects. Therefore we usually distinguish four basically different situations: There may be a semantic *equivalence, inclusion, overlapping,* or *disjointness* of class extensions (where class extension refers to the collection of objects represented by a class or relation):

Semantically Equivalent Class Extensions. The two classes (or relations) always represent exactly the same collection of real-world objects (see Fig. 11). Therefore the sets of instances stored for these two classes must represent the same real-world objects at each instant of time. Obviously this is a very strong property.

Semantic Inclusion of Class Extensions. In the case where only a subset of objects represented by one class is represented by another class, a semantic inclusion is given (Fig. 12). A semantic inclusion means that there is a subset relationship between the two sets of instances stored for these classes in the different local databases at each instant of time. In object-oriented approaches such an inclusion is modeled as a specialization between a class and its subclass.

Semantically Overlapping Class Extensions. In contrast to a semantic equivalence, the sets of instances stored do not need to completely match each other (see Fig. 13).

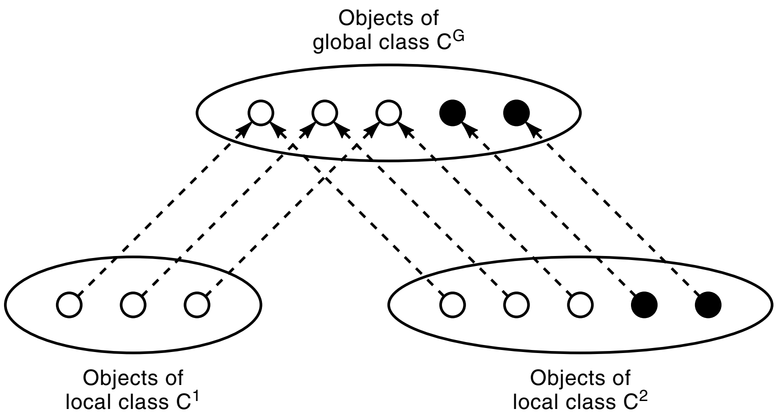

Figure 12. Semantic inclusion of class extensions.

There can be objects stored in one database without a corresponding object in the other database. A semantic overlap means that there can be an overlap of the currently stored instances, but it is not required that such an overlap occur at each instant of time.

Semantically Disjoint Class Extensions. This situation is of interest if the two disjoint class extensions (which are stored in different databases) semantically belong together (see Fig. 14). For calling two class extensions semantically disjoint, we must be sure that at no time one object can be represented in both databases.

Description Conflicts. Different approaches to describe the properties of real-world objects in the local database schemata can lead to conflicting descriptions. Due to different requirements of local applications, there can be different sets of properties (attributes) used to describe the same kind of objects. Furthermore homonyms and synonyms can occur as names of object classes, relations, and attributes, since the local schemata are designed independently and each designer makes his/her own choice of names.

Besides these basic conflicts there are a number of more subtle description conflicts. For instance, range conflicts occur when different ranges are used for corresponding attributes. In the same way we may find scaling conflicts if there are different units of measurement or different scaling of values in the local schemata. There is another type of description conflict if we have different integrity constraints for corresponding objects or class extensions.

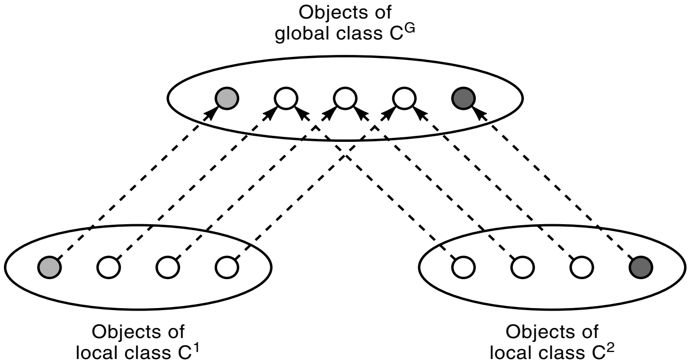

Figure 13. Semantically overlapping class extensions.

Structural Conflicts. The problem of different modeling possibilities for the same real-world fact is not limited to heterogeneous data models. Even in using the same data model there are usually several ways to express one fact. In particular, this holds for semantically rich data models (i.e., data models offering a lot of modeling concepts), but we can find different schemata describing the same universe of discourse and having the same real-world semantics for data models with only few modeling concepts like the relational model.

In Figure 6 we already gave an example where different modeling concepts were used to express the same real-world properties. Another typical example of a structural conflict is the situation where for one local schema we have an attribute that corresponds to a class or relation in another schema. This conflict can occur, for instance, if in the first schema only a single property of some real-world objects is of interest, whereas the applications using the second schema need several different properties of these objects.

INTEGRATION PROCESS

The goal of the integration process is to overcome heterogeneity in the data model and schema level. We will explain this process in relation to the 5-level-schema architecture described in Ref. 71. If necessary, the process can be adapted to the other schema architectures introduced above.

Common Data Model. The problem of different data models among the local schemata to be integrated is resolved by translating the local schemata into a common data model. Choosing the right common data model for the integration process is critical to the whole integration process. One criterion to use in choosing the right common data model is the semantic power of the modeling concepts. In the demand for completeness, the translation into a common data model must not be accompanied by a loss of semantics expressed by the local schemata (78). For this reason most approaches to schema integration prefer as a common data model a model with semantically rich concepts. Typically an object-oriented model is used. For the translation into a semantically more powerful data model, the local schemata must be semantically enriched [see (79,80)].

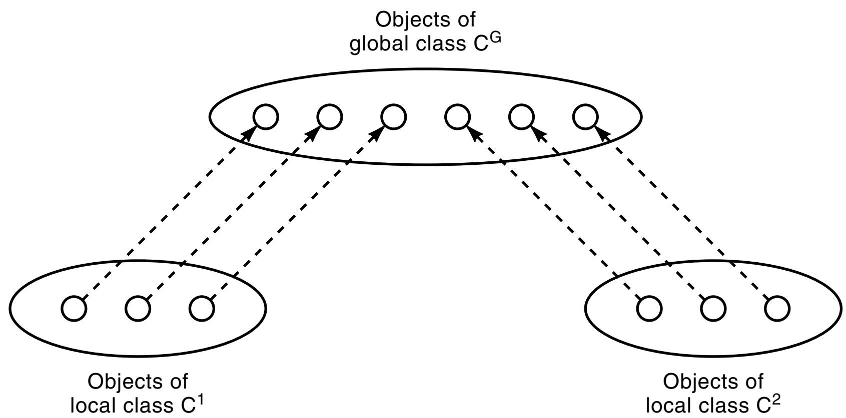

Figure 14. Semantically disjoint (but related) class extensions.

Deciding on an object-oriented data model, however, has a disadvantage that is not sufficiently considered in the literature: A semantically rich data model gives a designer freedom to model a universe of discourse in many different ways, which increases the heterogeneity on a schema level; thus more effort is needed to overcome the increased heterogeneity.

Beside the semantic richness there is another aspect that has to be considered in selecting the common data model. The common data model can be a design data model or a database model supported by commercial database management systems. The advantage of a design data model is its abstraction from implementation detail and the existence of a graphical notation for its schemata. The resulting integrated schemata, however, have to be later translated into a database model without loss of information, typically into the ODMG object model (81).

We explain here only the main ideas of schema integration. To make it more understandable, and since most approaches prefer the object-oriented data model, we choose the OMT object model (41) as the common data model.

Filtering Component Databases. Transforming the local schemata into the common data model produces the component schemata. Export schemata are defined on the component schemata in order to hide parts of the component databases from global applications. The restriction is mainly specified by applying *selection* and *projection* operations on the component schemata. The *selection* operation selects data records of the component database to be visible to global application in correspondence to selection conditions. The *projection* operation restricts the visible attributes of selected database records.

Problems of Schema Integration. The main focus of the following sections is a description of how to generate a federated schema from a given export schemata in the OMT object model. So-called schema integration must control heterogeneity on the schema level. Heterogeneity occurs when the same real-world aspect is modeled in different ways in parts of different schemata. Typically the integration of schemata is very complex. In practice, this process is often the bottleneck of any database integration (82).

There are many classes of conflicts contributing to heterogeneity. An ad hoc approach without considering the underlying method can thus fail largely because of the complexity involved. A big problem is just to detect conflict. Unfortunately, conflict is something that cannot be entirely and automatically detected in the schemata to be integrated. In general, additional information stemming from the designer of the component databases is required for detection. For example, knowledge of the semantic equivalence of a class `Person` and a class `People` can only come from a person knowing the semantics of the corresponding component databases. Obviously a thesaurus could help in some such instances, but each synonym must be confirmed by a human expert. Furthermore, in general, not all existing correspondences can be found by means of a thesaurus. Therefore the detection of conflicts can only proceed slowly.

Once a conflict is perceived, its resolution can be tricky. Often there is more than one way to reach a solution, so the best way must be decided. There is the additional matter that in resolving different classes of conflicts, the resolution of one conflict can cause another conflict. A clever rule giving an order to resolve conflict could in turn minimize the effort to integrate schemata. In contrast to conflict detection, schemata can be better integrated by applying rules and algorithms.

In summary, there is a need for a *design method* for schema integration. Such a method must define successive phases, classes of conflicts, and unification rules. We next give an overview of the different methods for schema integration in terms of the four phases identified in Ref. 76.

Phases of Schema Integration

1. *Preintegration.* In many practical environments more than two schemata are integrated. In this phase the designer has to decide on the strategy to use in integrating the given schemata. Answers to the following questions must be found: Do the schemata have different weights of relevance for the integration? Should the integration problem be broken down into the integration of two schemata at one time? And if so, in which order should they be integrated?
2. *Schema Comparison.* Schematic conflicts are detected in comparing the schemata. The information on correspondences among different schemata are typically captured in *correspondence assertions* (77).
3. *Schema Conforming.* Conflicts in the detected correspondences in this phase are resolved. This is done by

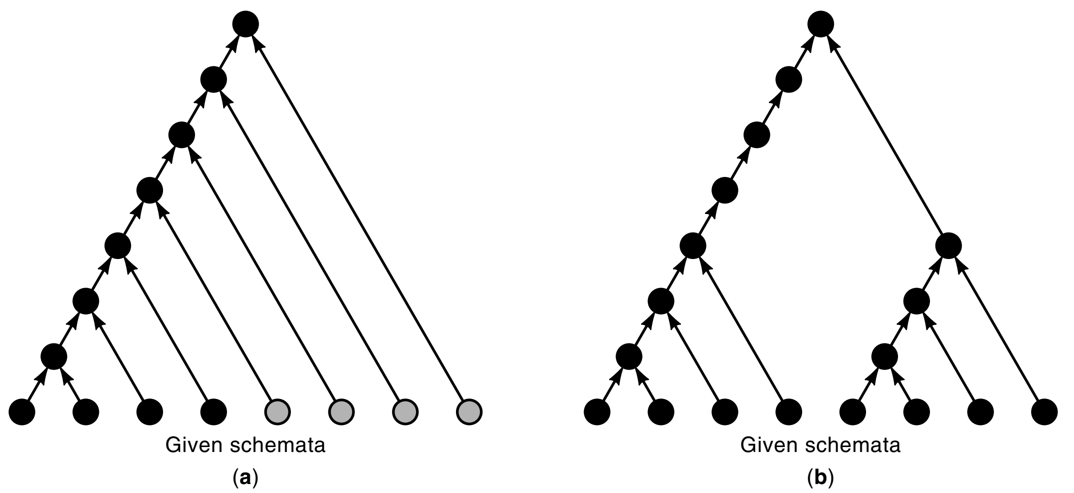

Figure 15. Weighted binary integration strategy.

schema transformations which homogenize the schemata to be integrated.

4. *Merging and Restructuring.* The homogenized schemata are merged into one federated schema. This schema, however, has to fulfill quality criteria such as minimality and understandability. Therefore additional restructuring transformations are often needed.

The phases of schema integration described in Ref. 76 do not fit to the 5-level-schema architecture. For example, external schemata are not considered. We adapt it here in a similar list of phases of schema integration:

1. *Preintegration.* This phase is the same as the preintegration phase described above.
2. *Schema Homogenization.* Schema homogenization combines the phases schema comparison and schema conforming. For each conflict class all conflicts have to be, first, detected and, second, resolved. In this way one class of conflicts is resolved before another class of conflicts is detected. This approach simplifiers the detecting of conflicts in contrast to the approach described in Ref. 76.
3. *Schema Merging.* In this integration phase the homogenized schemata are merged into one schema. Redundancy among the homogenized schemata is removed in a way that allows the federated schema to fulfill the demand for minimality.
4. *Derivation of External Schemata.* For different global applications, appropriate external schemata must be derived. This phase can also encompass a translation to another data model.

The following subsections describe the phases in more detail.

Preintegration

If more than two schemata have to be integrated, then preintegration allows us to select the right integration strategy. There are a number of integration strategies that integrate schemata to a single schema. The integration strategies differ in the number of intermediate integration steps, the number of schemata to be integrated in one intermediate integration step, and the weights associated to the given schemata.

The different integration strategies are pictured in Figs. 15–18 as different tree types. The leaf nodes denote the given schemata to be integrated, whereas the nonleaf nodes represent intermediate results of integration.

In the following we introduce four integration strategies described in Ref. 76. They can be organized into two groups. The first group contains *binary* integration strategies. These strategies integrate exactly two schemata in one integration step. Therefore the corresponding tree is a binary one. The other group contains *n-ary* integration strategies, which are not restricted to two schemata.

The advantage of a binary integration strategy is the reduced complexity of each integration step. Only two schemata have to be compared, conformed, and merged. If, however, more than two schemata have to be integrated, then the whole integration task must be broken down to various binary integration tasks. Therefore intermediate integration steps have to be performed.

We distinguish between two binary integration strategies: the *weighted* (see Fig. 15) and the *balanced* (see Fig. 16) integration strategy.

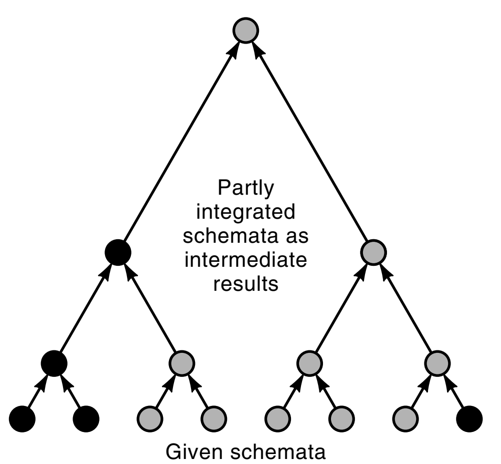

Figure 16. Balanced binary integration strategy.

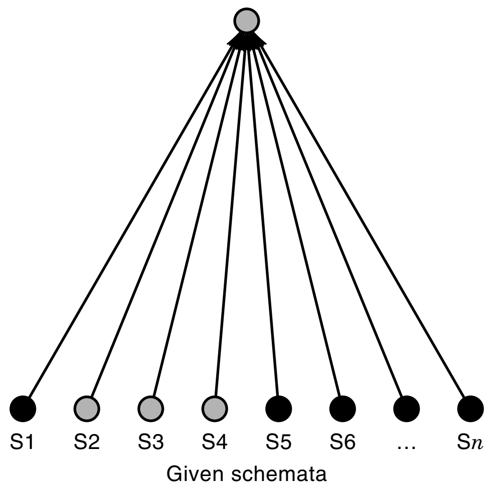

Figure 17. One shot integration strategy.

Weighted Binary Integration Strategy. The weighted binary integration strategy gives different weights to the schemata to be integrated. Some schemata are integrated in an earlier step than other schemata. The schemata considered early are analyzed and adjusted many times (as intermediate integration results) during the integration step. Of course there are many variants to a weighted integration tree construction. Figure 15 shows only two variants of weighted integration trees. A designer can influence the weight of each schema to be integrated by ordering them in this way on an unbalanced tree.

Balanced Binary Integration Strategy. The balanced binary integration strategy integrates all schemata with the same weight. No given schema is prioritized. The designer can only decide which given schemata have to be integrated in pairs in the first intermediate integration step.

In contrast to the binary integration strategies, the n-ary strategies of the second group do not restrict the number of schemata to be integrated to a single intermediate integration step. Therefore the number of intermediate steps can be fewer than of those of the binary integration strategy. We distinguish between two n-ary integration strategies: the *one-shot* (see Fig. 17) and the *iterative* integration strategy (see Fig. 18).

One-Shot Integration Strategy. A very simple integration strategy is the *one-shot* integration strategy. All schemata are integrated at the same time. The problem with this strategy is obviously its complexity. For n schemata the complexity in integrating them results from the fact that each schema can have correspondences to any number of other schemata.

Iterative Integration Strategy. In contrast to the one-shot integration strategy the *iterative* strategy does not integrate all schemata at the same time. Intermediate integration of schemata is performed. In contrast to the binary integration strategies, the iterative integration strategy is not restricted to two schemata to be integrated in one intermediate integration step. The next phases follow the binary approach whereby exactly two schemata are integrated as expressed in the OMT object model.

Schema Homogenization

Many schematic conflicts can occur between two schemata. We now describe how such conflicts are handled in homogenizing the schemata. The homogenization encompasses the detection and the resolution of conflicts. For the detection of conflicts the schemata must be compared. Tools can assist in this task but only in a restricted way.

Here we focus on which semantic correspondences are needed and how they are used to homogenize the schemata. We sketch the main ideas of conflict resolutions. Furthermore only the most frequently occurring conflict classes, and those that can be resolved by separate (without schema merging) schema transformations, are considered here. The subsections explain the treatment of description conflicts and structural conflicts. (Semantic conflicts and conflicts of *different attribute sets* as a specific type of description conflict that is not resolved by separate schema transformations. The next section will describe the treatment of these conflict classes.)

Description Conflicts. Different schemata can express redundancy; namely the corresponding databases can contain semantically equivalent objects. They are often described differently in the databases. For instance, the schemata define different sets of attributes for these objects. As mentioned above, this conflict class is explained in the next section. Other types of description conflicts considered here are the following:

- *Name Conflicts.* In the schemata the names for classes and attributes can be used in two ways:

  If two semantically equivalent classes or attributes are named differently, then a *synonym* exists.

  If a class or an attribute name has a different meaning, then the given schemata represents a *homonym.*

- *Attribute Conflicts.* Two attributes stemming from different schemata can be in conflict if they express a similar property of the corresponding real-world objects in different ways. This conflict is subdivided into the following conflict classes which often occur in a combined fashion:

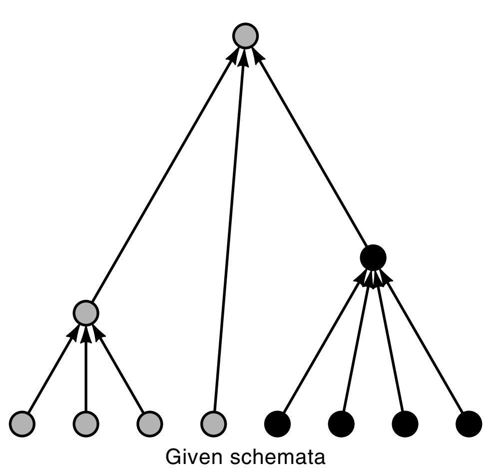

Figure 18. N-ary iterative integration strategy.

Different Values. Homonyms and synonyms can occur on a value level. For example, the strings "red" and "rot" as values of two semantically equivalent color attributes are synonyms caused by differently used languages (English and German). Another example was given previously where one price attribute includes the VAT and the corresponding price attribute excludes it.

Different Precisions. The semantically equivalent attributes can describe a property in different units of measure. For instance, one integer attribute might fix the length of a real-world object in meters whereas a corresponding integer attribute gives the length in inches. The use of different units of measurement introduces different precision levels.

- *Conflicting Integrity Constraints.* This conflict is the most difficult to assess. For corresponding classes or attributes where different integrity constraints are set, the object states or attribute values have different restrictions. Typically incomplete schema specifications are the cause of such conflicts. For example, each person of the class `person` of a first schema must be older than 30, whereas for the corresponding class the other schema do not give an age restriction.

Taking these short descriptions of conflict classes, we now turn to ideas on overcoming these conflicts.

Name Conflict. The classes and attributes of the schemata to be integrated can be compared by consulting a thesaurus. If two semantically equivalent classes (or attributes) are found to have different names, then the designer has to specify a synonym correspondence assertion of the following form:

⟨schema name⟩ . ⟨class name⟩
synonym
⟨schema name⟩ . ⟨class name⟩

For attributes the designer has to specify a synonym of the following form:

⟨schema name⟩ . ⟨class name⟩ . ⟨attribute name⟩
synonym
⟨schema name⟩ . ⟨class name⟩ . ⟨attribute name⟩

The placeholder in the brackets is replaced by the actual corresponding terms. Synonyms are easily removed by *renaming* classes and attributes. For the corresponding classes or attributes, respectively, *common* names are found.

Homonyms can be detected by comparing the class and attribute names. The designer has to declare class names to be homonyms in homonym correspondence assertions of the following form:

⟨schema name⟩ . ⟨class name⟩
homonym
⟨schema name⟩ . ⟨class name⟩

Homonym corresponding assertions for attributes have the following form:

⟨schema name⟩ . ⟨class name⟩ . ⟨attribute name⟩
homonym
⟨schema name⟩ . ⟨class name⟩ . ⟨attribute name⟩

To overcome homonym conflicts, *different* names can be introduced by the designer and the original names are changed accordingly.

Renaming classes and attributes as a schema transformation involves a very simple transformation. Since class and attribute names often occur in reference attributes and integrity constraints, the renaming operation must be performed there too.

Attribute Conflicts. If two attributes from different schemata express a similar property of the corresponding real-world objects in different ways, then an attribute conflict exists. Let us assume that we have two corresponding attributes a and b of an attribute conflict with domains $DOM(a)$ and $DOM(b)$. Different design views often cause different but semantically related attribute values of the two attributes:

Different Values. Similar to attribute names, attribute values can be synonyms or homonyms. In the homogenization, the designer must specify the mapping between the attribute domains. The function

$$f^{a\to b} \subseteq DOM(a) \times DOM(b)$$

relates a value of attribute a to a value of attribute b. There must also exist an inverse function

$$f^{a\to a} \subseteq DOM(b) \times DOM(a)$$

in order to propagate global inserts or updates to the component databases. The mapping must be therefore one-to-one. As Refs. 77, 83, 84, and 85 show, a table can be used to express the value correspondences. An example is given in Table 2, which compares English and German words for colors.

Sometimes it does not make sense to use a table in order to map attribute values. The functions $f^{a\to b}$ and $f^{b\to a}$ can be alternatively defined by arithmetic formulas or be computed algorithmically (77,83,85,86). An example is the definition of the functions $f^{a\to b}$ and $f^{b\to a}$ in mapping price values by two arithmetic formulas. In the first case the VAT is included, and in the other the VAT is excluded:

$$f^{a\to b}(a) = \frac{a}{1+\text{VAT}}$$
$$f^{b\to a}(b) = b * (1+\text{VAT})$$

For the resolution of different attribute values, one of the two representations must be selected. A schema

Table 2. Color Mapping by a Table

English Colors	German Colors
Red	Rot
Blue	Blau
Green	Grün
Black	Schwarz
White	Weiß

transformation results in one attribute being moved into the selected representation. In this instance we have the functions $f^{a \to b}$ and $f^{b \to a}$.

Since the domains of corresponding attributes can have different bounds, for some attribute values no related value of the corresponding attribute may exist. In Refs. 77 and 86 this problem is handled by uniting the domains in order to compute the domain of the transformed attribute. Additional integrity constraints are used to restrict the united domain. In this way the problem of differently bounded domains is transformed to the problem of conflicting integrity constraints.

Different Precisions. The values of two corresponding attributes a and b describe a property with different precisions. Attribute a is more precise than attribute b. In this conflict problem the specification of a mapping function $f^{a \to b}$ is needed. Due to the different precisions, however, the function is *not* injective. More than one precise value is related to one value of the less precise attribute. Therefore no inverse function exists.

These must be specified from the less precise attribute to the corresponding one by an additional function $f^{b \to a}$ that relates to each value exactly one more precise value. The existence of the functions $f^{a \to b}$ and $f^{b \to a}$ is necessary to support global read as well as update operations for mappings in both directions. The length of real-world objects, for example, can be expressed by an integer attribute in inches, whereas the corresponding integer attribute uses meters (see Fig. 3). Both functions can be defined as follows:

$$f^{a \to b}(a) = \lfloor a * 0.0254 \rfloor$$

$$f^{b \to a}(b) = \left\lceil \frac{b}{0.0254} \right\rceil$$

In Ref. 85 there are distinguished two types of conflict resolution:

- *Preference of More Precise Presentation.* The less precise attribute is transformed to a more precise attribute. If no function $f^{b \to a}$ is given, then the inverse mapping of $f^{a \to b}$ produces many precise values for one given value. As proposed by Ref. 85, we can use a value set, from which exactly one value is correct. To each value of the set an additional value of probability is computed and associated.

  If, however, the function $f^{b \to a}$ is specified, then it is used for the transformation. A problem arises because $f^{a \to b}$ and $f^{b \to a}$ are not mutually inverse. The result of this missing property is that after a global update operation on the transformed attribute, the read operation returns a value that can differ from the update value. In this way we have loss of information.
- *Preference of Less Precise Presentation.* The more precise attribute is transformed to the less precise presentation. A global value can be stored locally and read again as the same value. However, due to the less precise presentation, there is information loss during the transformation which violates the demand for completeness.

In Ref. 86 this conflict is solved by adopting both attributes separately to the merged and external schemata. The semantic relationship between those attributes is expressed by a specialization relationship. However, only few object models support the concept of attribute specialization.

Another approach to deal with different precisions is described in Ref. 87. Often the more precise attribute can be split into two attributes in such a way that between one of them and the corresponding attribute a one-to-one mapping can be specified. For example, the attribute `name` in one database contains the first and last name of persons, whereas the attribute in the other database contains only the last name. The attributes have different precisions. The conflict is resolved by splitting the first attribute into the attributes `first-name` and `last-name`.

Conflicting Integrity Constraints. The given schemata to be homogenized often specify integrity constraints on schema elements (classes and attributes). Due to semantic relationships between schema elements, there can exist correspondence assertions between them. A conflict between the integrity constraints occurs if the schemata specify different integrity constraints for corresponding schema elements and thus restrict the underlying databases differently.

For instance, two classes from different databases with the same name `person` are semantically related to each other by a correspondence assertion. The persons of the first database are restricted to persons that are younger than `50`, and the persons of the corresponding class must be older than `20` years. In this case the integrity constraints are in conflict.

Before we consider the resolution of such conflicts, we investigate some explanations for conflicting integrity constraints.

Reasons for Conflicting Integrity Constraints

1. *Incomplete Database Design.* It may happen that component databases were not designed completely or that not all integrity constraints are defined explicitly, though they are fulfilled by the databases due to implicit application semantics. For instance, a database may contain only persons older than `20`, but this integrity constraint is not specified. Due to this application only valid persons are inserted into the database. In other words, that integrity constraint exists implicitly in the database application.
2. *Wrong Correspondence Assertions.* In the comparison of schemata to be homogenized wrong correspondence assertions are identified. There often are different integrity constraints defined on corresponding schema elements.
3. *Different Contexts.* Corresponding classes do not exactly express the same semantics but are close semantically to each other. For example, one class contains employees of an insurance firm and the corresponding class contains persons insured by the firm. There is a correspondence between the two classes, since some persons can be employees and insured persons simultaneously.

If conflicting integrity constraints are caused by an incomplete database design, then an integrity constraint on one schema element is valid to the corresponding schema element. Therefore the schemata can be enriched by this integ-

Table 3. Mapping Between Different Length Measurements

Inches	Meters
⋮	⋮
40	1
⋮	⋮
78	1
79	2
⋮	⋮
118	2
119	3
⋮	⋮

rity constraint. In this way a schema integration helps to improve the given schemata. Conflicting integrity constraints can also help to detect wrong correspondence assertions. Wrong correspondence assertions are removed or replaced by correct ones.

Different Contexts. Conflicting integrity constraints caused by differing contexts require more complex solutions. There are two general approaches:

- *Disjunctive Combination.* For each given schema and for each schema element, the existing integrity constraints are combined *disjunctively* with the integrity constraints from the corresponding schema element. As a result the integrity constraints for those objects are weaken as the objects are stored in both databases. Such a weakening has a disadvantage. A new object may be inserted on the global level but not simultaneously stored in the component databases. In our previous example, a disjunctive combination of the integrity constraints `age < 50` and `age > 20` would eliminate the integrity constraints. Persons younger than `20` would be inserted globally but not in both databases simultaneously.
- *Conjunctive Combination.* An alternate approach is to make the integrity constraints of a schema element more restrictive so that not all locally stored objects are valid with respect to the combined integrity constraints. In other words, they are not visible on the global level. A conjunctive combination example would restrict the ages of persons to be between `20` and `50`. Persons outside this range would be stored locally and not appear in global applications.

The discussion above brings us to the problem of finding in the literature adequate coverage of integrity constraints, which is a difficult subject. Our survey below indicates the problems encountered so far in the work on conflicting integrity constraints:

- *References 88, 89.* These papers distinguish between *subjective* and *objective* integrity constraints. Subjective integrity constraints are important only in the local context. They are not considered in the integration process and are therefore not visible on global level.

  Due to the weakening of integrity constraints, this approach is similar to the disjunctive combination. Although subjective integrity constraints are specified locally, they have consequences for global applications. Not all global inserted objects can be propagated to the component databases. Their rejection is not plausible for global applications.
- *References 90, 91, 92.* These papers formally describe the problem of conflicting integrity constraints. They assume complete schema specification, and they therefore do not solve the conflict, nor as a matter of fact propose a real solution.
- *References 77, 93.* Both approaches propose to adopt the least restrictive integrity constraint from the conflicting integrity constraints. This approach is similar to the disjunctive combination. Global insertion of objects cannot always be propagated to the component databases.
- *References 94, 95.* The approach described in these papers differs from the other approaches because the specified integrity constraints are related to potential class extensions (set of possible instances of a class). In a decomposition step the classes of the schemata to be integrated are decomposed in such a way that each pair of classes ends up with either identical or disjoint extensions. Conflicting integrity constraints can now only occur between classes with identical extensions. For these classes the integrity constraints are combined conjunctively. The building of global classes in that approach is accompanied by an extensional uniting of disjoint classes. The integrity constraints of the global class are then formed by combining disjunctively the integrity constraints of the original classes. The processes of extensional decomposition and composition of the GIM-approach are explained in more detail in a later section.

Structural Conflicts. Most object models give the designer the freedom to model a real-world aspect differently and not use an identical model concept. This freedom creates structural conflicts between schema elements modeling the same real-world aspect. The most frequent type of structural conflict appears between an attribute and a class. An attribute of a class of one schema corresponds to a class of the other schema. On the instance level, there are correspondences between attribute values and objects. The integration designer has to compare both schemata to find such structural conflicts. He has to specify such conflicts as structural correspondence assertions of the following form:

⟨schema name⟩ . ⟨class name⟩ . ⟨attribute name⟩
structurally corresponds to
⟨schema name⟩ . ⟨class name⟩

For instance, the first schema has the class `Book` with the attributes `title`, `isbn`, and `publisher` whereas the second schema contains the class `Publisher` with its attributes `name` and `address` (see Fig. 19). A publisher, such as `John Wiley`, can be an attribute value in the first schema and an object in the second schema. The structural conflict is specified by the following structural correspondence assertion:

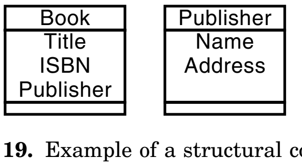

Figure 19. Example of a structural conflict.

`S1.Book.publisher`
structurally corresponds to
`S2.Publisher`

Structural conflicts are described in Refs. 77, 82, 96, and 97. For the resolution of this conflict one of the two presentations (as class or as attribute) must be preferred. Most approaches follow the strategy to prefer the less restrictive presentation. Applying this strategy to the structural conflict means to select the class presentation as the preferred variant. The class presentation enables object sharing because many references to the same object are possible. The same situation in the attribute presentation, however, will store the attribute value redundantly. Furthermore, in contrast to the attribute variant, a class presentation allows additionally characterizing attributes.

For homogenization the attribute presentation must be transformed into a class presentation. In this transformation step, a class must be created for each attribute involved in a structural conflict, whereby the attribute becomes a reference attribute directed to the new class. The newly created class then has generated for it an attribute that stores the value of a former attribute.

This transformation must consider integrity constraints, since new integrity constraints appear. For example, there is a uniqueness constraint defined for the new class on its generated attribute. Furthermore all integrity constraints that restrict the attribute of the attribute variant are adopted into the generated attribute of the new class.

On the instance level attribute values become objects. With each object a unique object identifier has to be associated. In order to have bijective database state mapping between the schemata before and after the schema transformation, bijective mapping between the attribute values and the generated object identifier must exist. Therefore an auxiliary table has to be managed.

Schema Merging

In this design step the homogenized schemata are merged into one schema. The merging concerns two types of schema elements:

- *Schema Elements without Correspondences.* Unique schema elements that have no semantic correspondence to schema elements of the second schema are merged without modifications.
- *Schema Elements with Correspondences.* Schema elements with semantic correspondences cannot be adopted in a one-to-one fashion into the merged schema because this approach would violate the demand for minimality. The schema elements with the same semantics are merged into *one* schema element of the resulting schema.

Due to conflicts not yet resolved we can have semantic correspondences between schema elements that do not express the same semantics. The next subsections will describe these conflicts and how they are resolved.

Merging schema elements with correspondences means to remove redundancy. Redundancy can also appear on the instance level. In component databases, values for the same attributes and for the same real-world objects can exist. For example, the name of a person can be stored redundantly in two databases. Such a problem can occur when values differ somewhat (data heterogeneity). The merged schema, however, must present exactly one integrated value for each attribute of two database objects representing the same real-world object. Unfortunately, there exists no general algorithm to compute the integrated values from the given values. The procedure must always be adapted to the specific situation. Two reasons for different versions of the same attribute value are the following:

- *Obsolete Values.* The values represent an attribute of a real-world object at different times before and after some change has occurred in the attribute of the real-world object. Ideally the more current value should be selected. In some cases the more current value is found in one of the databases. Then it is easy to select the right value as the integrated value. More often the decision is not so clearcut. Then an algorithm specific to the situation must be developed to compute the integrated value.
- *Wrong Values.* One or both values are wrong. The problem is to find the wrong value. In general, this is an unsolvable problem. For specific situations, however, good heuristics can often be found.

So far we have not considered the semantic conflict and the conflict of different attribute sets for corresponding classes. The next subsection deals with these conflicts and shows how they can be resolved by merging the given schemata into one schema. We will describe the classical approach which nevertheless has some disadvantages. In a subsequent subsection, we will introduce a newer approach that can be used to overcome these disadvantages.

Semantic Conflicts and Different Sets of Attributes. The problem of different semantics is the most frequent contributor to conflict in a schema integration. Conflict can also appear between semantically related classes when their extensions stand in a specific set relationship. For such related classes, different sets of attributes might be defined. In the literature these two types of conflict are often combined as in, for instance, Refs. 77, 96, 98, 99, 100, 101, and 102. Here we follow the approach of Ref. 77 which is representative of the proposed approaches.

The semantic conflict between two classes is given by a correspondence expression that fixes the semantic (extensional) relationship between them. There are five kinds of semantic relationships: $\equiv$, $\subseteq$, $\supseteq$, $\neq$, and $\cap$.

The equivalence ($\equiv$) means the equivalence of the class extensions: For each instance of the first or the second class, there exists at every instant of time an instance of the corresponding class extension that denotes the same real-world object. A subset condition ($\subseteq$ or $\supseteq$) express this implication only in one direction. That is, the class extensions are always in the specified subset relationship.

The symbol for disjointness ($\neq$) expresses that instances from two semantically related classes never denote the same real-world object. The symbol for overlapping ($\cap$) means no restriction for the class extension. The class extensions can contain semantically related and unrelated objects.

A semantic correspondence assertion is defined in the following form:

⟨schema name⟩ . ⟨class name⟩
⟨cor⟩
⟨schema name⟩ . ⟨class name⟩
with cor $\in \{\equiv, \subseteq, \supseteq, \neq, \cap\}$

For example, each person of the class `Person` of the first database is always stored in the class `Person` of the second database, and the extension of class `S2.Person` can contain more persons than the extension of the corresponding class. Therefore we have an extensional inclusion between these classes:

`S1.Person`
$\subseteq$
`S2.Person`

The classes related by a semantic correspondence assertion can have corresponding attributes. Due to the resolution of attribute conflicts related attributes have same names. The related classes, however, can have different attribute sets. The first class of two related classes has the attributes $\{a_1, \ldots, a_k, b_1, \ldots, b_l\}$, whereas the second class has the attributes $\{a_1, \ldots, a_k, c_1, \ldots, c_m\}$. That is, the attributes $\{a_1, \ldots, a_k\}$ denote the same set of attributes.

For example, the classes `publication` and `book` from different library databases can have overlapping extensions ($\cap$). The class `publication` has the attributes {`title`, `author`, `year`, `type`}, whereas for the class book the attributes {`title`, `author`, `isbn`} are defined. The classes overlap intentionally.

The conflicts of semantics and the different sets of attributes are resolved by applying the specialization concept in the federated schema. For this reason most approaches to schema integration suggest using an object-oriented data model as the common data model. For the resolution of these conflicts two additional classes are generated: a common superclass generated by a process of generalization and a common subclass by application of a specialization step. The extension of the superclass is defined by the union of the extensions of the given classes, and the extension of the subclass is computed by the intersection. The subclass inherits all attributes from the superclasses, whereas the common superclass contains the common attributes $\{a_1, \ldots, a_k\}$. Obviously the designer has to assign useful names to the new classes. The four resulting classes are illustrated in Fig. 20. There we show the inherited attributes explicitly. The application of this approach to the example concerning the overlapping classes `publication` and `book` is illustrated in Fig. 21.

The four classes are essential when the related classes have overlapping extensions and sets of attributes. Otherwise, some of the four classes can be omitted. For such a reduction the extensions of the four classes must be compared. If the extensions of some of the four classes are equivalent, then the highest superclass survives, whereas the other classes are omitted. The set of attributes of the topmost class is formed by the union of the attribute sets from the classes with equivalent extensions. Besides this reduction, classes with empty extensions can be removed.

The reduction can be demonstrated for the classes `S1.Person` and `S2.Person`. The extension of the class `S1.Person` is always a subset of the extension of class `S2.Person`. Since the union of a set with its superset equals the superset and their intersection equals the subset, two classes can be omitted. The result is illustrated in Fig. 22.

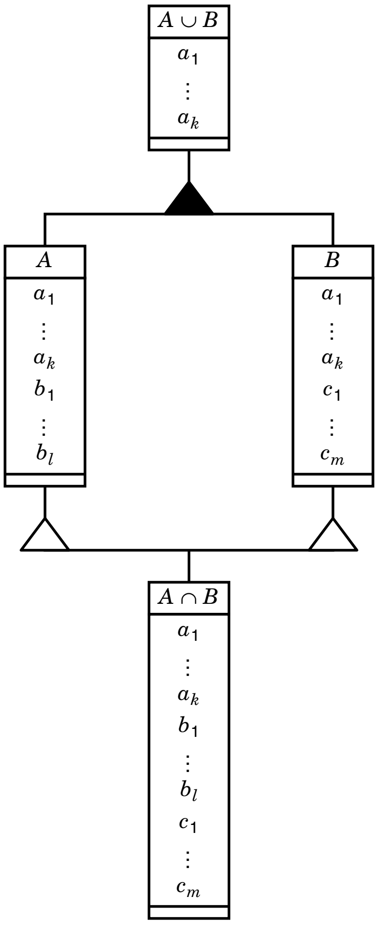

Figure 20. Resolution of a semantic conflict with different sets of attributes.

GIM-Approach. The presented approach of resolving semantic conflicts and different sets of attributes has some disadvantages. It is based on the existence of *binary* semantic correspondence assertions. However, often more than two classes are extensionally related, as is the case, for instance, if two specialization hierarchies are to be integrated. The extensional relations between more than two classes cannot be exactly expressed by binary semantic correspondence assertions. Therefore we need another formalism. As Ref. 103 proposes, we can use base extensions. In the following example we demonstrate the use of base extensions.

Figure 23 shows two example schemata. The union of the extensions of the classes `Employee` and `People` always equals the extension of the class `Person`. This information cannot be expressed using binary correspondence assertions. Table 4 specifies exactly this extensional relationship. The extensions of the example classes are decomposed into three disjoint base extensions. Each class extension is represented as the union of the corresponding base extensions. In this way the extensions of the classes and their extensional relationships can be specified correctly. Of course a base extension refers to potential class instances, since the extensional relationships are independent of a concrete database state. The

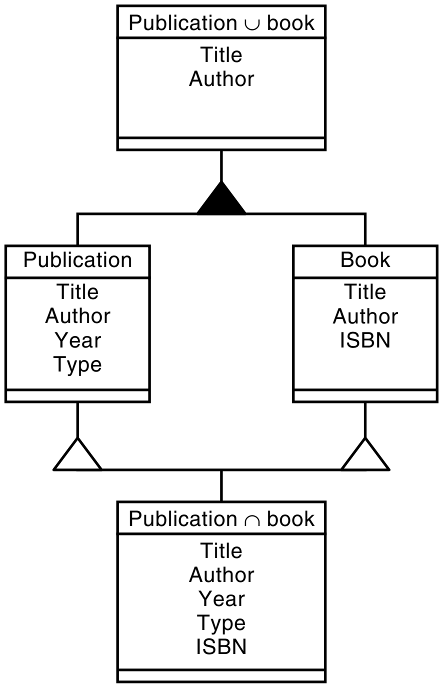

Figure 21. Resolution for the overlapping classes `publication` and `book`.

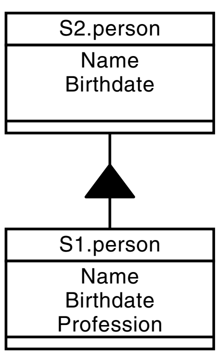

Figure 22. Resolution for the classes `S1.Person` and `S2.Person`.

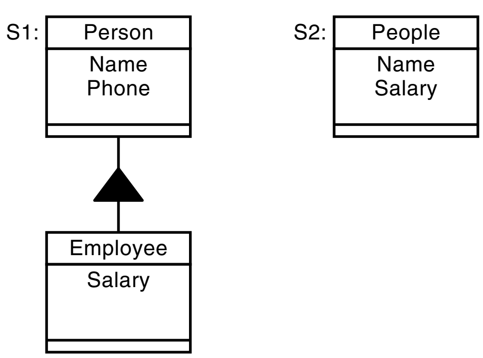

Figure 23. Two example schemata.

Table 4. Extensional Relationships of the Example

Base Extension	1	2	3
S1.Person	√	√	√
S1.Employee	√	√	
S2.People		√	√

following three semantic correspondence assertions, however, do not completely define the extensional relationships:

$$\texttt{S1.Person} \supseteq \texttt{S1.Employee}$$
$$\texttt{S1.Person} \supseteq \texttt{S2.People}$$
$$\texttt{S1.Employee} \cap \texttt{S2.People}$$

The semantic correspondence assertions cannot express that each person of the class `Person` is simultaneously stored as an object in the class `Employee` or in the class `People`.

Due to the incomplete information about extensional relationships, the approach introduced in the previous subsection cannot produce an adequate merged schema. Furthermore this approach can result in a merged hierarchy with a lot of generated subclasses and superclasses. As we will show in the next subsection, the GIM-approach [see (87)] produces in general relatively simple schemata.

Assume that we specified extensional relationships using base extensions. Following the GIM-approach, each base extension is now interpreted as a class. Such a class has an attribute if at least one of the corresponding original classes defines that attribute. In this way the schemata are merged into one merged schema considering base extensions as classes. This merged schema can be regarded as a table relating base extensions to attributes. Of course this presentation is simplified because integrity constraints, data types, and reference attributes are not considered. It is, however, sufficient to explain the main idea of the GIM-approach. Table 5 presents the merged schema of our example. Here all three attributes are defined for all base extensions.

Since the merged schema cannot serve as a schema for applications, the merged schema has the function of an intermediate representation. An additional step is necessary to produce an understandable schema. This step can also be used to derive external schemata and is therefore described in the next section.

Derivation of External Schemata

In general, more than one application runs on the global level of a database integration. The applications often have different views on the integrated data. Analogously to views in relational databases, external schemata have to fit to the view of the applications and provide logical data independence. Due to the similarity with the views of traditional databases, their mechanisms can be applied to derive external schemata.

Table 5. Merged GIM-Schema of the Example

Base Extension	1	2	3
Name	√	√	√
Phone	√	√	√
Salary	√	√	√

Merging schemata produces an object-oriented, merged schema. Therefore the external schemata have to be derived from an object-oriented schema. In general, this process is more complex than deriving views from relational schemata. In Ref. 104, we have an overview of view mechanisms for object-oriented databases.

The GIM-approach results in a merged schema where we have a lot of disjoint classes. The merged schema is represented by a table that assigns attributes to base extensions. Due to the disjointness of the classes, this schema contains too many classes to be understandable for global applications. To derive external schemata, however, GIM-classes can be related to classes of the external schemata. Each correct class corresponds to a rectangle in the GIM-schema. To be more exact, for each class of the external schemata, there is a sequence of base extensions and attributes in the table so that the ticks form a rectangle. Therefore, in order to derive external classes, the designer has to find rectangles in the GIM-schema. For finding a minimal number of external classes the found rectangles must be maximal. Maximal rectangles cannot be extended by attributes or base extensions.

Different classes stemming from maximal rectangles can be in a specialization relationship. A class is a subclass of another class if its set of base extensions is a subset of the base extension set of the other class. In this way a whole specialization hierarchy can be generated. To find maximal rectangles and specialization relations between them the theory of formal concept analysis can be applied. In Ref. 105 is introduced a theory of formal concept analysis. Applying this theory, however, has an exponential computational complexity. Furthermore it produces a lattice that contains removable classes. In Ref. 106 is described an algorithm to compute an external schema in correspondence to an application view in polynomial complexity.

The GIM-schema of our example has exactly one maximal rectangle. This example demonstrates that the schema integration can result in a very simple external schema, whereas the other approach produces an unnecessarily complex merged schema.

CONCLUSION

The traditional view of the database design process assumes that a database is designed and developed from scratch. In practice, we have to deal with existing databases quite frequently. Due to the fact that a redesign of an existing database is expensive, database integration is often required. The design of a global database is also restricted by the existing databases and the requirement that their design not be changed so that local applications can be continued without modifications.

The research in the database field shows that database design and database integration are still highly developing topics. Current as well as future application areas (data warehouses, databases for OLAP, etc.) will likely need supporting work in design and integration.

BIBLIOGRAPHY

1. H. Mannila and K.-J. Räihä, *The Design of Relational Databases,* Reading, MA: Addison-Wesley, 1992.
2. C. Batini, S. Ceri, and S B. Navathe, *Conceptual Database Design—An Entity-Relationship Approach,* Redwood City, CA: Benjamin/Cummings, 1992.
3. T J. Teorey, *Database Modeling and Design: The Fundamental Principles,* San Francisco: Morgan Kaufmann, 1994.
4. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* Redwood City, CA: Benjamin/Cummings, 1994.
5. R. J. Wieringa, *Requirements Engineering: Frameworks for Understanding,* Chichester: Wiley, 1996.
6. G. Saake, Conceptual Modeling of Database Applications, in D. Karagiannis (ed.), *Proc. 1st IS/KI Workshop, Ulm,* Berlin: Springer-Verlag, 1991, pp. 213–232.
7. G. Saake, Descriptive specification of database object behaviour, *Data Knowl. Eng.,* **6**: 47–74, 1991.
8. U. Schiel et al., Towards multi-level and modular conceptual schema specifications, *Inf. Syst.,* **9**: 43–57, 1984.
9. J. Carmo and A. Sernadas, A temporal logic framework for a layered approach to systems specification and verification, in C. Rolland et al. (eds.), *Proc. IFIP Working Conf. Temp. Aspects Info. Syst.,* Amsterdam: North-Holland, 1988, pp. 31–46.
10. G. Engels et al., Conceptual modelling of database applications using an extended ER model, *Data Knowl. Eng.,* **9**: 157–204, 1992.
11. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics,* Berlin: Springer-Verlag, 1985.
12. P. P. Chen, The entity-relationship model—Towards a unified view of data, *ACM Trans. Database Syst.,* **1**: 9–36, 1976.
13. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* Redwood City, CA: Benjamin/Cummings, 1994.
14. R. A. Elmasri, J. Weeldreyer, and A. Hevner, The category concept: an extension to the entity-relationship model, *Data & Knowledge Engineering,* **1** (1): 75–116, 1985.
15. R. A. Elmasri, J. Weeldreyer, and A. Hevner, The category concept: An extension to the entity-relationship model, *Data Knowl. Eng.,* **1**: 75–116, 1985.
16. M. Atkinson et al., The object-oriented database system manifesto, in W. Kim, J.-M. Nicolas, and S. Nishio (eds.), *Proc. 1st Int. Conf., DOOD'89, Kyoto,* Amsterdam: North-Holland, 1990, pp. 223–240.
17. A. Sernadas, Temporal aspects of logical procedure definition, *Info. Systs.,* **5** (3): 167–187, 1980.
18. U. W. Lipeck, *Dynamic Integrity of Databases* (in German), Berlin: Springer-Verlag, 1989.
19. J. Chomicki, Real-time integrity constraints, *Proc. 11th ACM SIGACT-SIGMOD-SIGART Symp. Prin. Database Syst.,* San Diego, 1992, pp. 274–281.
20. J. Chomicki and D. Toman, Temporal logic in information systems, in J. Chomicki and G. Saake (eds.), *Logics for Databases and Information Systems,* Boston: Kluwer, 1998, pp. 31–70.
21. J. Eder et al., BIER: The behaviour integrated entity relationship approach, in S. Spaccapietra (ed.), *Proc. 5th Int. Conf. Entity-Relationship Approach (ER'86),* Dijon, 1987, pp. 147–166.
22. U. W. Lipeck and G. Saake, Monitoring dynamic integrity constraints based on temporal logic, *Inf. Syst.,* **12**: 255–269, 1987.
23. R. J. Wieringa, J.-J. Ch. Meyer, and H. Weigand, Specifying dynamic and deontic integrity constraints, *Data Knowl. Eng.,* **4**: 157–189, 1989.
24. S. Khoshafian and G. P. Copeland, Object identity, in N. Meyrowitz (ed.), *Proc. 1st Int. Conf. OOPSLA'86,* Portland, Oregon, ACM Press, 1986, pp. 406–416.
25. K. Hülsmann and G. Saake, Representation of the historical information necessary for temporal integrity monitoring, in F. Banchilhon, D. Thanos, and D. Tsichritzis (eds.), *Advances*

Database Technol.—EDBT'90, Proc. 2nd Int. Conf. Extending Database Technol., Venice, Berlin: Springer-Verlag, 1990, pp. 378–392.

26. J. Rumbaugh et al., *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
27. G. Booch, I. Jacobson, and J. Rumbaugh, *Unified Modeling Language (version 1.0),* Rational Software Corp., Santa Clara, CA, 1997.
28. R. Jungclairs et al., Troll—a language for objected-oriented specification of information systems, *ACM Trans. Info. Systs.,* **14** (2): 175–211, 1996.
29. U. W. Lipeck, Transformation of Dynamic Integrity Constraints into Transaction Specifications, *Theor. Comput. Sci.,* **76**: 115–142, 1990.
30. S. Khosla, T. Maibaum, and M. Sadler, Database specification, in T. Steel and R. A. Meersman (eds.), *Proc. IFIP WG 2.6 Working Conf. Data Semantics (DS-1),* Hasselt, Belgium, Amsterdam: North-Holland, 1985, pp. 141–158.
31. J. Fiadeiro and A. Sernadas, Specification and verification of database dynamics, *Acta Info.,* **25**: 625–661, 1988.
32. H. Wächter and A. Reuter, The ConTract model, in A. K. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992, pp. 219–263.
33. M. Atkinson et al., The object-oriented database system manifesto, in W. Kim, J.-M. Nicolas, and S. Nishio (eds.), *Proc. 1st Int. Conf., DOOD'89, Kyoto,* Amsterdam: North-Holland, 1990, pp. 223–240.
34. C. Beeri, Formal Models for Object-Oriented Databases, in W. Kim, J.-M. Nicolas, and S. Nishio, eds., *Proc. 1st Int. Conf., DOOD'89, Kyoto,* Amsterdam: North-Holland, 1990, pp. 405–430.
35. S. Abiteboul and R. Hull, IFO—A formal semantic database model, *ACM Trans. Database Syst.,* **12**: 525–565, 1987.
36. D. S. Batory and W. Kim, Modeling Concepts for VLSI CAD Objects, *ACM Trans. Database Syst.,* **5**: 322–346, 1985.
37. W. Kim and F. H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications,* New York: ACM Press, 1989.
38. M. M. Hammer and D. J. McLeod, Database description with SDM: A semantic database model, *ACM Trans. Database Syst.,* **6**: 351–386, 1981.
39. R. Hull and R. King, Semantic database modelling: Survey, applications, and research issues, *ACM Comput. Surveys,* **19**: 201–260, 1987.
40. S. D. Urban and L. Delcambre, An analysis of the structural, dynamic, and temporal aspects of semantic data models, *Proc. Int. Conf. Data Eng.,* 1986, pp. 382–387.
41. J. Rumbaugh et al., *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
42. G. Booch, *Object-Oriented Design with Applications,* Redwood City, CA: Benjamin/Cummings, 1991.
43. K. G. Kulkarni and P. Atkinson, EFDM: Extended functional data model, *Comput. J.,* **29**: 38–46, 1986.
44. D. Shipman, The functional data model and the data language DAPLEX, *ACM Trans. Database Syst.,* **6**: 140–173, 1981.
45. J. J. V R. Wintraecken, *The NIAM information Analysis Method—Method and Practice.* Dordrecht: Kluwer, 1990.
46. A. Heuer and G. Saake, *Databases—Concepts and Languages, 1. Correction* (in German), Bonn: International Thomson, 1997.
47. D. Maier, The Theory of Relational Databases, Rockville, MD: Computer Science Press, 1983.
48. C. J. Date and H. Darwen, *A Guide to the SQL Standard,* Reading, MA: Addison-Wesley, 1993.
49. T. J. Teory, *Database Modeling and Design: The Fundamental Principles,* San Francisco, CA: Morgan Kaufmann, 1994.
50. D. Heimbigner and D. McLeod, A federated architecture for information management, *ACM Trans. Office Info. Systs.,* **3** (3): 253–278, 1985.
51. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* Redwood City, CA: Benjamin/Cummings, 1994.
52. R. Ramakrishnan, *Database Management Systems,* Boston, MA: WCB/McGraw-Hill, 1998.
53. A. L. Tharp, *File Organization and Processing,* New York: Wiley, 1988.
54. D. E. Shasha, *Database Tuning: A Principled Approach,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
55. M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
56. J. Banerjee et al., Semantics and implementation of schema evolution in object-oriented databases. In U. Dayal and I. Traiger (eds.), *Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, CA,* 311–322, ACM SIGMOD Record, **16** (3): ACM Press, 1987.
57. G. T. Nguyen and D. Rieu, Schema evolution in object-oriented database systems, *Data & Knowledge Engineering,* **4** (1): 43–67, 1989.
58. S. L. Osborn, The role of polymorphism in schema evolution in an object-oriented database, *IEEE Trans. Knowl. Data Eng.,* **1** (3): 310–317, 1989.
59. J. Andany, M. Leonard, and C. Palisser, Management of schema evolution in databases. In G. M. Lohmann et al. (eds.), *Proc. of the 17th Int. Conf. on Very Large Data Bases (VLDB'91), Barcelona, Spain,* 161–170, San Mateo, CA: Morgan Kaufmann, 1991.
60. R. Zicari, A framework for schema updates in an object-oriented database system. In N. Cercone and M. Tsuchiya (eds.), *Proc. of the 7th IEEE Int. Conf. on Data Engineering, ICDE'91, Kobe, Japan,* 2–13, IEEE Computer Society Press, 1991.
61. E. Sciore, Versioning and configuration management in an object-oriented data model, *VLDB J.,* **3** (1): 77–107, 1994.
62. E. Bertino and L. Martino, *Object-Oriented Database Systems—Concepts and Architectures.* Wokingham, England: Addison-Wesley, 1994.
63. A. Kemper and G. Moerkotte, *Object-Oriented Database Management.* Englewood Cliffs, NJ: Prentice-Hall, 1994.
64. H. Frank and J. Eder, Integration of behaviour models. In S. W. Liddle (ed.), *Proceedings ER'97 Workshop on Behavioural Models and Design Transformations: Issues and Opportunities in Conceptual Modeling (6–7 November 1997, UCLA, Los Angeles, CA),* 1997.
65. G. Preuner and M. Schrefl, Observation consistent integration of business processes. In C. McDonald (ed.), *Database Systems: Proceedings of the 9th Australian Database Conference, Perth, Australia, Feb. 1998 (ADC'98),* **2** (20); Springer-Verlag: Australian Computer Science Communications, 1998.
66. M. T. Özsu and P. Valduriez, Distributed database systems: Where are we now? *IEEE Comput.,* **24** (8): 68–78, 1991.
67. M. T. Özsu and P. Valduriez, Distributed data management: Unsolved problems and new issues, in T. Casavant and M. Singhal (eds.), *Readings in Distributed Computing Systems,* Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 512–514.
68. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems,* New York: McGraw-Hill, 1985.
69. M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
70. D. Bell and J. Grimson, *Distributed Database Systems,* Reading, MA: Addison-Wesley, 1992.

71. A. P. Sheth and J. A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, *ACM Comput. Surveys,* **22**: 183–236, 1990.

72. D. Heimbigner and D. McLeod, A federated architecture for information management, *ACM Trans. Off. Info. Syst.,* **3**: 253–278, 1985.

73. W. Litwin, L. Mark, and N. Roussopoulos, Interoperability of multiple autonomous databases, *ACM Comput. Surveys,* **22**: 267–293, 1990.

74. J. Grant et al., Query languages for relational multidatabases, *VLDB J.,* **2**: 153–171, 1993.

75. L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian, SchemaSQL—A language for interoperability in relational multi-database systems, in T. M. Vijayaraman et al. (eds.), *Proc. 22nd Int. Conf. Very Large Data Bases (VLDB'96), Bombay,* San Francisco: Morgan Kaufmann, 1996, pp. 239–250.

76. C. Batini, M. Lenzerini, and S. B. Navathe, A comparative analysis of methodologies for database schema integration, *ACM Comput. Surveys,* **18**: 323–364, 1986.

77. S. Spaccapietra, C. Parent, and Y. Dupont, Model independent assertions for integration of heterogeneous schemas, *VLDB J.,* **1**: 81–126, 1992.

78. F. Saltor, M. Castellanos, and M. Garcia-Solaco, Suitability of data models as canonical models for federated databases, *ACM SIGMOD Record,* **20**: 44–48, 1991.

79. M. Castellanos, Semantic enrichment of interoperable databases, in H.-J. Schek, A. P. Sheth, and B. D. Czejdo (eds.), *Proc. 3rd Int. Workshop on RIDE-IMS'93, Vienna,* Los Alamitos, CA: IEEE Computer Society Press, April 1993, pp. 126–129.

80. U. Hohenstein, Using Semantic Enrichment to Provide Interoperability between Relational and ODMG Databases, in J. Fong and B. Siu (eds.), *Multimedia, Knowledge-Based and Object-Oriented Databases,* Berlin: Springer-Verlag, 1996, pp. 210–232.

81. R. G. G. Cattell and D. K. Barry, eds., *The Object Database Standard: ODMG-93, Release 2.0,* San Francisco, CA: Morgan Kaufmann, 1997.

82. S. Navathe and A. Savasere, A schema integration facility using object-oriented data model, in O. A. Bukhres and A. K. Elmagarmid (eds.), *Object-Oriented Multidatabase Systems—A Solution for Advanced Applications,* Upper Saddle River, NJ: Prentice-Hall, 1996, pp. 105–128.

83. U. Dayal and H. Y. Hwang, View definition and generalization for database integration in a multidatabase system, *IEEE Trans. Softw. Eng.,* **10**: 628–644, 1984.

84. L. DeMichiel, Resolving database incompatibility: An approach to performing relational operations over mismatched domains, *IEEE Trans. Knowl. Data Eng.,* **1**: 485–493, 1989.

85. A. L. P. Chen, P. S. M. Tsai, and J.-L. Koh, Identifying object isomerism in multidatabase systems, *Distributed and Parallel Databases,* **4**: 143–168, 1996.

86. J. A. Larson, S. B. Navathe, and R. Elmasri, A theory of attribute equivalence in databases with application to schema integration, *IEEE Trans. Softw. Eng.,* **15**: 449–463, 1989.

87. I. Schmitt, *Schema Integration for the Design of Federated Databases,* (in German), Dissertationen zu Datenbanken und Informationssystemen, vol. 43. PhD thesis. Sankt Augustin: Infix-Verlag, 1998.

88. M. W. W. Vermeer and P. M. G. Apers, The role of integrity constraints in database interoperation, in T. M. Vijayaraman et al. (eds.), *Proc. 22nd Int. Conf. Very Large Data Bases (VLDB'96), Bombay,* San Francisco: Morgan Kaufmann, 1996, pp. 425–435.

89. M. W. W. Vermeer, *Semantic Interoperability for Legacy Databases,* CTIT PhD thesis 97-11, Enschede, The Netherlands: Centre for Telematics and Information Technology, 1997.

90. J. Biskup and B. Convent, A formal view integration method, in C. Zaniolo (ed.), *Proc. 1986 ACM SIGMOD Int. Conf. Manage. of Data, Washington, DC,* pp. 398–407. *ACM SIGMOD Rec.,* **15**: New York: ACM Press, 1986.

91. B. Convent, Unsolvable problems related to the view integration approach, in G. Ausiello and P. Atzeni (eds.), *Proc. 1st Int. Conf. Database Theory (ICDT'86), Rome,* Berlin: Springer-Verlag, 1986, pp. 141–156.

92. L. Ekenberg and P. Johannesson, Conflictfreeness as a basis for schema integration, in S. Bhalla (ed.), *Information Systems and Data Management, Proc. 6th Conf. CIS-MOD'95, Bombay,* Berlin: Springer-Verlag, 1995, pp. 1–13.

93. M. P. Reddy, B. E. Prasad, and A. Gupta, Formulating global integrity constraints during derivation of global schema, *Data Knowl. Eng.,* **16**: 241–268, 1995.

94. S. Conrad, I. Schmitt, and C. Türker, Dealing with integrity constraints during schema integration, in *Engineering Federated Database Systems EFDBS'97—Proc. Int. CAiSE'97 Workshop, Barcelona,* 1997, pp. 13–22. Fakultät für Informatik, Universität Magdeburg, 1997.

95. S. Conrad, I. Schmitt, and C. Türker, Considering integrity constraints during federated database design, in *Advances in Databases, 16th British Nat. Conf. Databases, BN-COD 16, Cardiff, Wales, 1998,* Berlin: Springer-Verlag, 1998.

96. A. Motro, Superviews: Virtual integration of multiple databases, *IEEE Trans. Softw. Eng.,* **13**: 785–798, 1987.

97. S. Spaccapietra and C. Parent, View integration: A step forward in solving structural conflicts, *IEEE Trans. Knowl. Data Eng.,* **6**: 258–274, 1994.

98. M. T. Özsu and P. Valduriez, Distributed database systems: Where are we now? *IEEE Comput.,* **24** (8): 68–78, 1991.

99. M. V. Mannino, B. N. Navathe, and W. Effelsberg, A rule-based approach for merging generalization hierarchies, *Inf. Syst.,* **13**: 257–272, 1988.

100. A. P. Sheth, S. K. Gala, and S. B. Navathe, On automatic reasoning for schema integration, *Int. J. Intell. Coop. Inf. Syst.,* **2**: 23–50, 1993.

101. M. Garcia-Solaco, M. Castellanos, and F. Saltor, A semantic-discriminated approach to integration in federated databases, in S. Laufmann, S. Spaccapietra, and T. Yokoi (eds.), *Proc. 3rd Int. Conf. Coop. Inf. Syst. (CoopIS'95), Vienna,* 1995, pp. 19–31.

102. Y. Dupont and S. Spaccapietra, Schema integration engineering in cooperative databases systems, in K. Yetongnon and S. Hariri (eds.), *Proc. 9th ISCA Int. Conf. PDCS'96, Dijon,* 1996, pp. 759–765.

103. I. Schmitt and G. Saake, Integration of inheritance trees as part of view generation for database federations, in B. Thalheim (ed.), *Conceptual Modelling—ER'96, Proc. 15th Int. Conf., Cottbus, Germany,* Berlin: Springer-Verlag, 1996, pp. 195–210.

104. R. Motschnig-Pitrik, Requirements and comparison of view mechanisms for object-oriented databases, *Info. Syst.,* **21**: 229–252, 1996.

105. B. Ganter and R. Wille, *Formal Concept Analysis,* Berlin: Springer-Verlag, 1998.

106. I. Schmitt and G. Saake, Merging inheritance hierarchies for schema integration based on concept lattices, Preprint no. 2, Fakultät für Informatik, Universität Magdeburg, 1997. Also available online via http://wwwiti.cs.uni-magdeburg.de/publikationen/97/SS97.ps.gz

GUNTER SAAKE
STEFAN CONRAD
INGO SCHMITT
Otto-von-Guericke-Universität Magdeburg

DATABASE FOR RADAR SIGNATURE ANALYSIS.

See OBJECT ORIENTED DATABASE FOR RADAR SIGNATURE ANALYSIS.