

Database Component Ware

Bernhard Thalheim

Computer Science Institute, Brandenburg University of Technology at Cottbus,
PostBox 101344, D-03013 Cottbus
Email: thalheim@informatik.tu-cottbus.de

Abstract

Database modeling is still a job of an artisan. Due to this approach database schemata evolve by growth without any evolution plan. Finally, they cannot be examined, surveyed, consistently extended or analyzed. Querying and maintenance become very difficult. Distribution of database fragments becomes a performance bottleneck. Currently, databases evolve to huge databases. Their development must be performed with the highest care.

This paper aims in developing an approach to *systematic* schema composition based on components. The approach is based on the internal skeletal meta-structures inside the schema. We develop a theory of database components which can be composed to schemata following a architecture skeleton of the entire database.

1 Towards Information Systems Engineering

Observations.

Database modeling is usually carried out by handi-craft. Database developers know a number of methods and apply them with high craftman's skills. Monographs and database course books usually base explanations on small or 'toy' examples. Therefore database modeling courses are not the right school for people handling database applications in practice. Database applications tend to be large, carry hundreds of tables and have a very sophisticated and complex integrity support.

Database schemata tend to be large, unsurveyable, incomprehensible and partially inconsistent due to application, the database development life cycle and due to the number of team members involved at different time intervals. Thus, consistent management of the database schema might become a nightmare and may lead to legacy problems. The size of the schemata may be very large, e.g., the size of the SAP R/3 schema consisting of more than 21.000 tables. In contrast, (Moody 2001) discovered that diagrams quickly become unreadable once the number of entity and relationship types exceeds about twenty.

It is a common observation that large database schemata are error-prone, difficult to maintain and to extend and not-surveyable. Moreover, development of retrieval and operation facilities requires highest professional skills in abstraction, memorization and programming. Such schemata reach sizes of more

than 1000 attribute, entity and relationship types. Since they are not comprehensible any change to the schema is performed by extending the schema and thus making it even more complex.

This observation becomes more severe in the case of huge database. The computational facilities known and applicable so far to huge database are insufficient. Huge databases only survive and can be successfully run if their schemata have been developed with the highest care and are thus of highest quality.

Large schemata also suffer from the *deficiency of variation detection*: The same or similar content is often repeated in a schema without noticing it¹. The SAP R/3 schema is a typical example of schema evolution. During first years of exploitation the honeycomb architecture has worked nicely. The schema was easy to maintain and to the database was simple to query. Later, however, a large number of extensions has led to a situation that the system cannot be down-sized to the needs of medium companies².

The Opposite Approach: Application Engineering.

Communications of ACM devoted an entire issue to component engineering (Arsanjani 2002). This issue summarizes the approaches developed so far. A typical approach is the low level pattern-based component engineering (Crnkovic 2002). Components are units of composition. "*The most important feature of a component is the separation of its interfaces from its implementation. This separation is different from those we find in OO programming languages, where class definitions are separated from class implementation.*" (Crnkovic et al. 2002) This approach is far from being novel. Already structured programming and languages such as Modula-2 had this feature.

Software engineering is still based on *programming in the small* although a number of approaches has been tried to reason on *programming in the large*. Software development is mainly based on stepwise development from scratch. Software reuse has been considered but never reached the maturity for application engineering. Database development is also mainly *development in the small*. Schemes are developed step by step, extended type by type, and normalized locally type by type. Views are still defined type by type al-

¹The Lufthansa cargo database schema contains, for instance, several sub-schemata which store very similar information on the transport log and accounting: air transport of goods, ground transport through cooperating companies and ground transport on the airports. These three kinds of transport have been modeled and implemented by three differently located teams. The similarity of the schemata has not been detected by the teams and caused a number of redundancy and inconsistency problems.

²During my sabbatical in Spring 1999 we have analyzed the entire schema and found a large number of repeating tables and large redundancy within the data. For instance, similar address information is kept at the same time in more than 75 tables.

though more complex schemata can be easily defined by extended ER schemata (Thalheim(HERM) 2000).

Therefore, database engineering must still be considered as *handicraft* work which require the skills of an *artisan*. Engineering in other disciplines has already gained the maturity for industrial development and application.

Engineering applications have been based on the simple *separation principle*: *Separation of elements which are stable from elements which are not*.

This separation allows *standardization* and *simple integration*. An example is the specification of screws as displayed in Figure 1³. Screws have a standardized representation: basic data, data on the material, data on the manufacturing, data on specific properties such as head, etc.

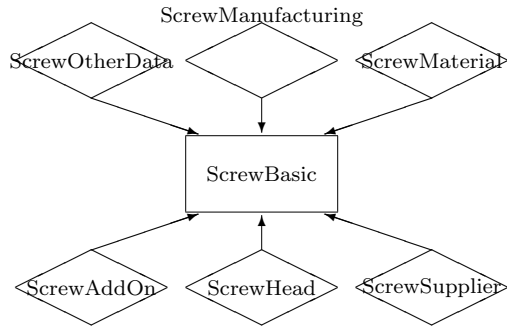


Figure 1: HERM Representation of the Star Type Screw

Hierarchy Abstraction.

Large database schemata can be drastically simplified if techniques of modular modeling such as *design by units* (Thalheim(HERM) 2000) are used. Modular modeling is an abstraction technique based on principles of hiding and encapsulation. Design by units allows to consider parts of the schema in a separate fashion. The parts are connected via types which function similar to bridges.

Hierarchy abstraction enables in considering objects in a variety of levels of detail. Hierarchy abstraction is based on a specific form of the general join operator (Thalheim 2002). It combines types which are of high adhesion and which are mainly modeled on the basis of star sub-schemata. *Specialization* is a well-known form of hierarchy abstraction. For instance, an *Address* type is specialized to the *GeographicAddress* type. Other forms are role hierarchies and category hierarchies. For instance, *Student* is a role of *Person*. *Undergraduate* is a category of *Student*. The behavior of both is the same. Specific properties have been changed. *Variations* and *versions* can be modeled on the basis of hierarchy abstraction.

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

³ We use the extended ER model (Thalheim(HERM) 2000) that allows to display subtypes on the basis of unary relationship types and thus simplifies representation.

Component Construction.

The term “component” has been around for a long time. Component-based software has become a “buzzword” since about ten years beyond classical programming paradigms such as structured programming, user-defined data types, functional programming, object-orientation, logic programming, active objects and agents, distributed systems and concurrency, and middleware and coordination. Various component technologies have been developed since then: Source-level language extensions (CORBA, JavaBeans); binary-level object models (OLE, COM, COM+, DCOM, .NET); compound documents (OLE, OpenDoc, BlackBox).

A component is considered to be a software implementation that can be autonomically executed, implements one or more interfaces, has a system-wide identity, has instances with their own identity, bundles data and procedures and hides the details of the implementation that are irrelevant to the outside.

The components usually used are considered to be small programs. In reality, a component is a basic unit which can be separated. Therefore, the size might be larger than usually considered in COM+ programming.

Codesign of Structuring, Functionality and Interactivity.

Large schemata are developed for large applications, are extended after integration of other schemata and are modified after the schema has been used. At the same time, all facets (structuring, functionality and interaction) of the application must be considered. Codesign (Thalheim(HERM) 2000) of database applications aims in consistent development of all facets of database applications: structuring of the database by schema types and static integrity constraints, behavior modeling by specification of functionality and dynamic integrity constraints and interactivity modeling by assigning views to activities of actors in the corresponding dialogue steps. First, a skeleton of components is developed. This skeleton can be refined during evolution of the schema. Then, each component is developed step by step. If this component is associated to another component then its development must be associated with the development of the other component as long as their common elements are concerned.

Therefore, structuring in codesign may be based on two constructs:

Components: Components are the main building blocks. They are used for structuring of the main data. The association among components is based on ‘connector’ types (called hinge or bridge types) that enable in associating the components in a variable fashion.

Skeleton-based construction: Components are assembled together by application of connector types. These connector types are usually relationship types.

Components are not Extensions of Objects.

Object-orientation has led to a better culture in software projects. Beside a huge number of ad-hoc approaches with an ad-hoc semantics and ad-hoc implementations, a sophisticated model object-oriented database can be developed (Schewe/Thalheim 1993). Object-orientation has led to a number of conceptions that are widely used in database applications such as object identifier, rich type systems, active objects, triggers, and polymorphism. At the same time

limitations of these concepts have been investigated, e.g., pitfalls of identifiability (Beeri/Thalheim 1998) or rule triggering (Schewe/Thalheim 1998).

Object-orientation has led to a large number of misconceptions ((Webster 1995) provides a list of 88 pitfalls.) which substantially reduced its usefulness. Object-orientation is not well-suited for component-based development and hinders it (Nierstrasz 1995). Object-oriented source code exposes the class hierarchy and not the interaction among objects. Therefore objects are wired instead of plugged together. The association of objects is distributed among the objects. Object-orientation is domain-driven and leads to designs based on domain objects instead of available components and standard architectures. Rich object interfaces are used instead of plug-compatible interfaces. Furthermore, compositional abstractions such as synchronization policies, coordination abstractions, wrappers and mixins (Ancona/Zucca 2002) cannot be naturally handled as objects.

Goals of the Paper.

The paper develops a methodology for *systematic development of large schemata*. Analyzing a large number of applications it has been observed in (Thalheim 2000) that large schemata have a high internal similarity. This similarity can be used to reason on the schema in various levels of detail. At the same time, similarity can be used for improvement and simplification of the schema.

At the same time, each schema has building blocks. We call these blocks or cells in the schema *component*. These components are combined with each other. At the same time, schemata have an internal many-dimensionality. Main or kernel types are associated with information facets such as meta-characterization, log, usage, rights, and quality information. The schemata have a meta-structure. This meta-structure is captured by the *skeleton* of the schema. This skeleton consists of the main modules without capturing the details within the types. The skeleton structure allows to separate parts of the schema from others. The skeleton displays the structure at a large. At the same time, schemata have an internal *meta-structure*.

2 Skeletons and Components Within Database Schemata

2.1 Dimensionality Within a Schema

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

2.2 Component Sub-Schemata

Database schema development may be performed similar to engineering approaches (Thalheim 2002).

This approach is similar to Hilbert's programme of redevelopment of Mathematics. It is based on inductive construction:

Development of building blocks: There are parts in the database schema which cannot be partitioned into smaller parts without losing their meaning. Typical such parts are kernel types together with their specialization types.

Development of composition methods: Composition of sub-schemata to larger schemata is mainly based on association types, i.e. relationship types bridge types, nest of types, introduce variations of types, etc.

Rules for application of composition methods:

Constraints for application of composition methods allow to keep track on restrictions, special application conditions and on the context of the types to be composed.

Our approach is based on principles of component construction (Broy 1997), the theory of functional systems (Kudrjavcev 1982), the theory of cooperating views (Thalheim(HERM) 2000), and the theory of structures (Malzew 1970). The last theory is the kernel for the theory of abstract state machines (Gurevich 1997). Components are composed to schemata. The composition theory of stream processing functions (Broy 1997) is combined with the interface scripting (Nierstrasz 1995).

A **component** is *database scheme that has an import and an export interface for connecting it to other components by standardized interface techniques*.

A **database component** $\mathcal{C} = (S, I^V, O^V, S^C, \Delta)$ is specified by

(static) schema S describing the database schema (and static integrity constraints),

syntactic interface providing names (structures, functions) with parameters and database structure for the database state S^C and the view schemata for the input view I^V and the output view O^V ,

behavior relating the I^V, O^V (view) channels with their input stream from M^* to the output stream from M^* and the database that has been changed according to the input and the previous database state

$$\Delta : (S^C \times (I^V \rightarrow M^*)) \rightarrow \mathcal{P}(S^C \times (O^V \rightarrow M^*)).$$

Component combinations may be based on a theory of schema constructors. We introduce these constructors in Section 4 in a general form.

Component composition operations such as merge, fork, transmission are definable via application of superposition operations:

Identification of channels: Given a component $\mathcal{C} = (S, I^V, O^V, S^C, \Delta)$ and domain-compatible channels C_1 and C_2 . A new schema $\zeta_{C_2:=C_1}(\mathcal{C})$ is obtained by identifying the channels and using the names of C_1 .

Permutation of channels: Given a component $\mathcal{C} = (S, I^V, O^V, S^C, \Delta)$ and domain-compatible channels C_1 and C_2 . A new schema $\tau_{C_1, C_2}(\mathcal{C})$ is obtained by interchanging the channels.

Renaming of channels: Given a component $\mathcal{C} = (S, I^V, O^V, S^C, \Delta)$ and a channel set C of \mathcal{C} . Given further domain-compatible channels C' which are free of name conflicts with C . A new schema $\delta_{C, C'}(\mathcal{C})$ is obtained by renaming all types in C into C' .

Introduction of fictitious channels: Given a component $\mathcal{C} = (S, I^V, O^V, S^C, \Delta)$. Given further domain-compatible channels C which are free of name conflicts with \mathcal{C} . A new schema $\iota_C(\mathcal{C})$ is obtained by adding C to the component without being associated to any channel of \mathcal{C} . The domain type $dom(C) = \emptyset^C$ is defined by using the empty set for all basic types of C .

Parallel composition with feedback: The parallel composition $\mathcal{C}_1 \otimes \mathcal{C}_2 = (S, I^V, O^V, S^C, \Delta)$ of two components without name conflicts $\mathcal{C}_1 = (S_1, I_1^V, O_1^V, S_1^C, \Delta_1)$ and $\mathcal{C}_2 = (S_2, I_2^V, O_2^V, S_2^C, \Delta_2)$ via domain-compatible channels is defined via assignment of syntactical interfaces (with unification of output channels to input channels) $I^V = (I_1^V \setminus O_2^V) \cup (I_2^V \setminus O_1^V)$, $O^V = (O_1^V \setminus I_2^V) \cup (O_2^V \setminus I_1^V)$, internal channels $Z^V = (I_1^V \cap O_2^V) \cup (I_2^V \cap O_1^V)$ and schema union $S = S_1 \cup S_2$ for $S^C = S_1^C \times S_2^C$.

The composition of the behavior function $\Delta = \Delta_1 \otimes \Delta_2$ defined by

$$\Delta((\sigma_1, \sigma_2), x) = \{ ((\sigma'_1, \sigma'_2), y|_{O^V}) \mid \begin{aligned} & y|_{I^V} = x|_{I^V} \wedge \\ & (\sigma'_1, y|_{O_1^V}) \in \Delta_1(\sigma_1, y|_{I_1^V}) \wedge \\ & (\sigma'_2, y|_{O_2^V}) \in \Delta_2(\sigma_2, y|_{I_2^V}) \} \end{aligned}$$

for $S^C = S_1^C \times S_2^C$.

Parallel composition is displayed in Figure 2.

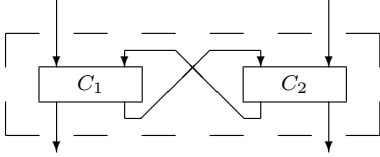


Figure 2: The Composition of Database Components

Components may form a component algebra $(\{\mathcal{C}\}, \zeta, \delta, \tau, \iota, \otimes)$. A database schema is formed from components by applying the functions of the component algebra. Components can be associated to each other. The association is restricted to domain-compatible input or output schemata which are free of name conflicts.

2.3 Skeleton and Architecture of Schemata

Skeletons form a framework and survey the general architecture or plan of an application to which details such as the types are to be added. Plans show how potential types associated to each other in general.

The skeleton is defined by units and their associations:

Units and components: The basic element of a unit is a component. A set of components forms a unit if this set can be semantically separated from all other components without losing application information. Units may contain entity, relationship and cluster types. The types have a certain affinity or adhesion to each other.

Units are graphically represented by rounded boxes.

Associations of units: Units may be associated to each other in a variety of ways. This variety reflects the general associations within an application. Associations group the relation of units by their meaning. Therefore different associations may exist between the same units.

Associations can also relate associations and units or associations. Therefore, we use an inductive structuring similar to the extended entity-relationship model (Thalheim(HERM) 2000).

Associations are graphically represented by double rounded boxes.

The skeleton is based on the *repulsion* of types. The measure for repulsion can be based on natural numbers with zero. A zero repulsion is used for relating weak types to their strong types. The repulsion measure $r(x, y)$ is a norm in the Mathematical sense, i.e. $r(x, y) \geq 0$, $r(x, x) = 0$, $r(x, y) \leq r(x, z) + r(z, y)$. The repulsion measure allows to build i-shells $\{T' \mid r(T, T') \leq i\}$ around the type T .

Repulsion is a semantic measure that depends on the application area. It allows to separate types in dependence on the application. We may enhance the repulsion measure with application points of view \mathcal{V} , i.e., introduce a measure $r_V(T, T')$ for types T, T' and views V from \mathcal{V} .

These views are forming the associations of the units. Associations are used for relating units or parts of them to each other. Associations are often representing specific facets of an application such as points of view, application areas, and workflows that can be separated from each other.

Let us consider a database schema used for recording information on production processes: the *Party* unit with components such as *Person*, *Organization*, the variety of their subtypes, address information, and data on their profiles, their relationship, etc.,

the *Work* unit with components *Product*, specializations of *Product* such as *Good* and *Service*, *Consumption* of elements in the production process, and the *WorkEffort* component which enables in describing the production process,

the *Asset* unit consisting of only one component *Asset* with subtypes such as *Property*, *Vehicle*, *Equipment*, and *Other Asset* for all other kinds of assets, and

the *Invoice* unit which combines components *Invoicing* with *InvoiceLineItem* which lists work tasks and work efforts with the cost and billing model, *Banking*, and *Tracking*.

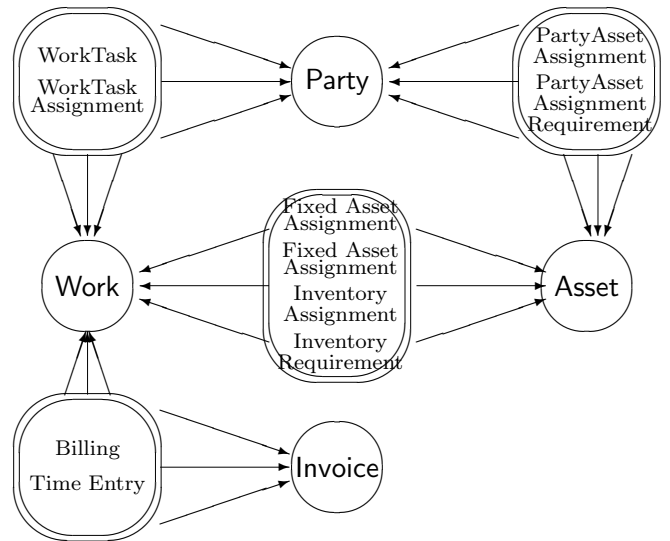


Figure 3: Skeleton of a Schema Supporting Production Applications

These units have a number of associations among them:

the *WorkTask-WorkAssignment* association combines facets of work such *PartyAllocation*, *PartyWork-TaskAssignment*, *TrackOfWorkTasks*, and *ControllingOfWorkTasks*,

the *PartyAssetAssignment* association combines the units *Party* and *Asset*,

the *Billing* association is relationship among *Work* and *Invoices* and has components such as *Tracking*, *Controlling*, and *Archiving*,

the *AssetAssignment* association allows to keep track on the utilization of fixed assets in the production process and thus associates *Work* and *Asset*. The four main units can be surveyed in a form displayed in Figure 3. We assume that billing is mainly based on the work effort. Work consumes assets and fixed assets.

The full schema for the *Production* application has been discussed in (Thalheim 2000). It generalizes schemata which have been developed in production applications and consists of more than more 150 entity and relationship types with about 650 attribute types.

In general, the skeleton forms an *architecture* of the schema. The components may be composed into larger components. The skeleton displays the meta-structure of the schema. This meta-structure enables is separating parts of the schema from other parts. This separation allows simple replacement of parts of schemata by new parts which meet new requirements in the application. As long as this replacement is free of side-effects due to the local replacement nothing than the replacement is performed. If the replacement is concerned with more than one component all components must be considered.

2.4 Meta-Characterization of Components, Units, and Associations

Utilization information is often only kept in log files. Log files are inappropriate if the utilization or historic information must be kept after the data have been changed. Database applications are often keeping track of utilization information based on archives. The same observation can be made for schema evolution. We observed that database schemata change already within the first year of database system exploitation. In this case, the schema information must be kept as well.

The skeleton information is kept by a meta-characterization information that allows to keep track on the purpose and the usage of the components, units, and associations. Meta-characterization can be specified on the basis of docket (Schmidt/Schring 1999) that provide information. The following frames follows the codesign approach (Thalheim(HERM) 2000) with the integrated design of structuring, functionality, interactivity and context. The frame is structured into general information provided by the header, application characterization, the content of the unit and *documentation* of the implementation.

- on the content (*abstracts* or *summaries*),
- on the delivery instruction,
- on the parameters of functions for treatment of the unit (opening with(out) zooming, breath, size, activation modus for multimedia components etc.)
- on the tight association to other units (versions, releases etc.),
- on the meta-information such as resources, restriction, copyright, roles, distribution policy etc.

- on the content providers, content reviewers and review evaluators with quality control policies,
- on applicable workflows and the current status of completion and
- on the log information that enable in tracing the object's life cycle.

Dockets can be extended to general descriptions of the utilization. The following definition frame is appropriate which classifies meta-information into mandatory, good practice, optional and useful information. The following table is divided into mandatory and optional information:

header	
content	name
problem area	
solution	intention
variants	application area
application	
applicability	consequences of application
usability profile	experience reports
description	
structuring: structure, static constraints	functionality: operations, dynamic constraints, enforcement procedures
implementation	
implementation	code sample
associated unit	collaboration
mandatory	good practice

The first table displays information necessary for any application. The following table concentrates on properties of components which should be part of any good documentation:

header	
developer	copyright
motivation	source
also known as	see too
application	
sample applications	known applications
DBMS	other DBMS
description	
interactivity: story space, actors, media objects, representation	context: tasks, intention, history, environment, particular
implementation	
associated framework	links to extensions
integration strategy	versions
optional	useful

3 Database Components

We restrict now the theory of database components to entity-relationship schemes. It can be developed in a similar form based on UML diagrams or object-oriented techniques. In HERM schemes we observe a number of basic structures. These basic structures can be structurally described by star or snowflake schemes introduced in the following subsections. Considering a large number of applications in (Thalheim 2000) we found star and snowflakes to be the only components. A similar observation has been made in (Levene/Loizou 2002). Components are seldom of the granularity of a singleton entity type. They are rather sub-schemata.

3.1 Star Component Schema

A star schema for a database type C_0 is defined by

- the (full) (HERM) schema $\mathcal{S} = (C_0, C_1, \dots, C_n)$ covering all types on which C_0 has been defined,

- the subset of *strong types* C_1, \dots, C_k forming a set of keys K_1, \dots, K_s for C_0 , i.e., $\cup_{i=1}^s K_i = \{C_1, \dots, C_k\}$ and $K_i \rightarrow C_0$, $C_0 \rightarrow K_i$ for $1 \leq i \leq s$ and $card(C_0, C_i) = (1, n)$ for $(1 \leq i \leq k)$.
- the extension types C_{k+1}, \dots, C_m satisfying the (general) cardinality constraint $card(C_0, C_j) = (0, 1)$ for $((k+1) \leq i \leq n)$.

The extension types may form their own (0, 1) specialization tree (hierarchical inclusion dependency set). The cardinality constraints for extension types are partial functional dependencies.

There are various variants for representation of a star schemata:

- Representation based on an entity type with attributes C_1, \dots, C_k and C_{k+1}, \dots, C_l and specializations forming a specialization tree C_{l+1}, \dots, C_n .
- Representation based on a relationship type C_0 with components C_1, \dots, C_k , with attributes C_{k+1}, \dots, C_l and specializations forming a specialization tree C_{l+1}, \dots, C_n . In this case, C_0 is a *pivot element* (Biskup/Polle 2000) in the schema.
- Representation by be based on a hybrid form combining the two above.

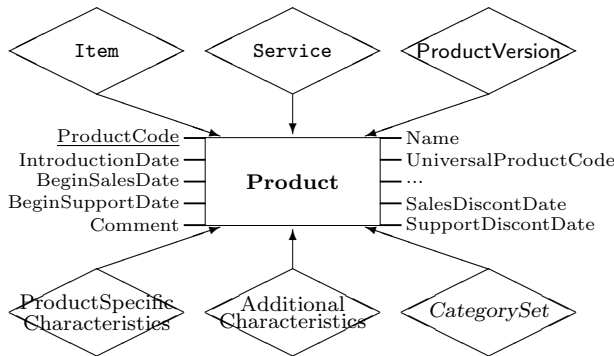


Figure 4: Star Component Schema *Product_Intext_Data* for the *Product* Application

The schema in Figure 4 is based on the first representation option. It shows the different facets of product characterizations. Thus, a *star component schema* is usually characterized by a kernel entity type used for storing basic data, by a number of dimensions that are usually based on subtypes of the entity type such as *Service* and *Item*, and on subtypes which are used for additional properties such as *AdditionalCharacteristics* and *ProductSpecificCharacteristics*. These additional properties are clustered according to their occurrence for the things under consideration. Furthermore, products are classified by a set of categories. Finally, products may have their life and usage cycle, e.g., versions. Therefore, we observe that the star schema is in our case a schema with four dimensions: subtypes, additional characterization, life cycle and categorization.

3.2 Varying Star Component Schemata for Variable Integration

Star schemata may occur in various variants within the same conceptual schema. Therefore, we need variants of the same schema for integration into the schema. We distinguish the following variants:

Integration and representation variants: For representation and for integration we can define views on the star type schema with the restriction of invariance of identifiability through one of its keys. Views define ‘context’ conditions for usage of elements of the star schema.

Versions: Objects defined on the star schema may be a replaced later by objects that display the actual use, e.g., *Products* are obtained and stored in the *Inventory*.

Variants replacing the entire type another through renaming or substitution of elements.

History variants: Temporality can be explicitly recorded by adding a history dimension, i.e., for recording of instantiation, run, usage at present or in the past, and archiving.

Lifespan variants of objects and their properties may be explicitly stored. The lifespan of products in the acquisition process can be based on the *Product-Quote-Request-Response-Requisition-Order* cycle.

3.3 Snowflake Component Schema

Star schemata may be extended to snowflake schemata. Database theory folklore uses star structures on the basis of α -acyclic hypergraphs (Thalheim 1990, Yuan/Osoyoglu 1992). Snowflake structuring of objects can be caused by the internal structure of functional dependencies. If for instance, the dependency graph for functional dependencies forms a tree then we may decompose the type into a snowflake using the functional dependency $X \rightarrow Y$ for binary relationship types R on X, Y with $card(R, X) = (1, 1)$ and $card(R, Y) = (1, n)$.

This observation is not surprising. The only case when decomposition algorithms are applicable and generate the same results as synthesis algorithms is the case that the dependency graph for functional dependencies is hierarchical.

Let us now generalize this observation. A snowflake schema is a

- star schema \mathcal{S} on C_0 extended or changed by
 - variations \mathcal{S}^* of star schema (with renaming)
 - with strong 1-n-composition by association (glue) types $A_S^{S'}$ associating the star schema with another star schema \mathcal{S}' either with full composition restricted by the cardinality constraint $card(A_S^{S'}, S) = (1, 1)$ or with weak, referencing composition restricted by $card(A_S^{S'}, S) = (0, 1)$,
- which structure is potentially C_0 -acyclic.

A schema \mathcal{S} with a ‘central’ type C_0 is called *potentially C_0 -acyclic* if all paths p, p' from the central type to any other type C_k are

- either entirely different on the database, i.e., the exclusion dependency $p[C_0, C_k] \parallel p'[C_0, C_k]$ is valid in the schema
- or completely identical, i.e. the pairing inclusion constraints $p[C_0, C_k] \subseteq p'[C_0, C_k]$ and $p[C_0, C_k] \supseteq p'[C_0, C_k]$ are valid.

The exclusion constraints allow to form a tree by renaming the non-identical types. In this case, the paths carry different meanings. The pairing inclusion constraints allow to cut the last association in the second path thus obtaining an equivalent schema or to introduce a mirror type C'_k for the second path. In this case, the paths carry identical meaning.

The schema in Figure 5 is similar to the relational schema in (Levene/Loizou 2002) and illustrates the properties of snowflakes. The paths *Lecture-Lecturer-WorksIn-Department* and *Lecture-Program-OfferedBy-Department* are based on pairing inclusion dependencies. The paths *Lecture-Course* and *Lecture-BasedOn-Course* are based on an exclusion dependency.

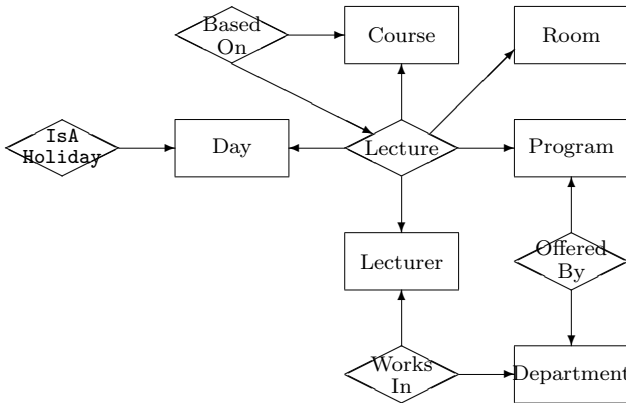


Figure 5: A Snowflake Component of a Lecture Planning Database

The constraints allow to separate the cycles in either

tight cycles which couple data in a mutually form and are expressed by pairwise path inclusion constraints such as

$$\begin{aligned} & \text{Lecture-Lecturer-WorksIn-Department} \\ & \quad [Lecture, Department] \\ \subseteq \supseteq & \text{Lecture-Program-OfferedBy-Department} \\ & \quad [Lecture, Department] \end{aligned}$$

or

data-separated cycles which can be displayed by schemata with mirrors as shown in Figure 6 similar to the approach used in hierarchical database systems. Data-separated cycles are expressed by path exclusion constraints such as $Lecture-Course[Lecture, Course] \parallel$ $Lecture-BasedOn-Course[Lecture, Course]$.

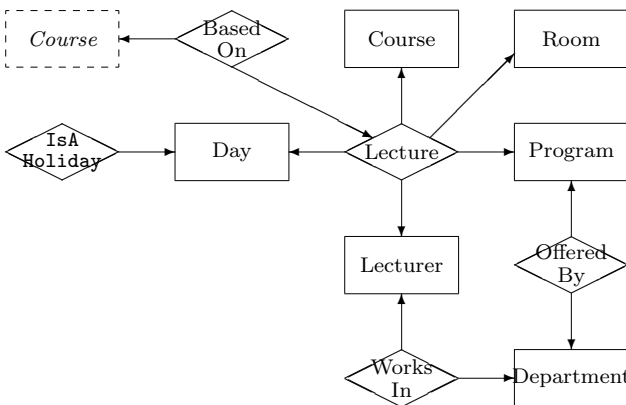


Figure 6: A Snowflake Component with Mirror Types

4 Composition of Schemata Based On Components

4.1 Composition By Constructors

We distinguish three main methods for composition of components: *construction by combination* or association on the basis of constructors, *construction by folding* or combining schemata to more general schemata and *construction by categorization*. It is not surprising that these methods are based on principles of component abstraction (Smith/Smith 1977). Composition is based on the skeleton of the application and uses a number of composition constructors.

4.1.1 Constructor-Based Composition:

Star and snowflake schemata may be composed by the composition operations such as *product*, *nest*, *disjoint union*, *difference* and *set* operators. These operators allow to construct any schema of interest since they are complete for sets. We prefer, however, a more structural approach following (Brown 2000). Therefore, all constructors known for database schemata may be applied to schema construction.

The extended entity-relationship model supports these constructor based composition on the basis of relationship types. The composition expression is either forming a relationship type as the type *Judge* relating the *CourtCase* type to the *Person* type. displayed in Figure 7 or is described by the direct composition expression, e.g. the type *Accused* is associated to the *Court Case* and to the *Party* which defined by the disjoint union of *Person* and *OtherCourtParty*. The types *CourtCase* and *Person* may be the central types of a *Court* star schema and a *Person* star schema.

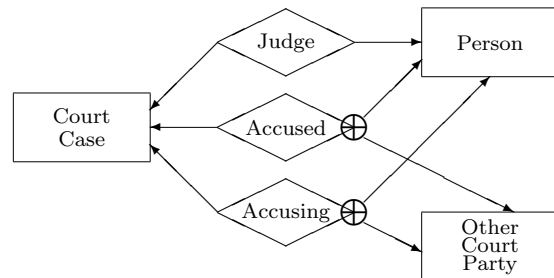


Figure 7: Application of Composition Constructors in a Court Case Application

4.1.2 Bulk Composition:

Types used in schemata in a very similar way can be clustered together on the basis of a classification. Let us exemplify this generalization approach for *Ordering processes*. The types *PlacedBy*, *TakenBy*, and *BilledTo* in Figure 8 are similar. They associate orders with both *PartyAddress* and *PartyContactMechanism*. They are used together and at the same objects, i.e. each order object is at the same time associated with one party address and one party contact mechanism.

Thus, we can combine the three relationship types into the type *OrderAssociation*. The type *OrderAssociationClassifier* allows to derive the three relationship type. The domains $dom(ContractionDomain) = \{PlacedBy, TakenBy, BilledTo\}$ and $dom(ContractionBinder) = \forall$ can be used to extract the three relationship types as displayed in Figure 9.

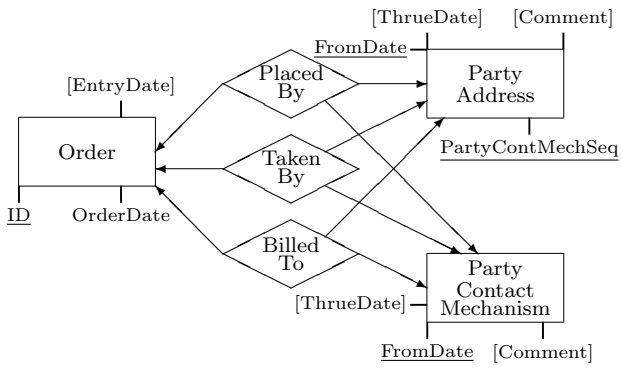


Figure 8: Similarity of Types in the *Order* Application

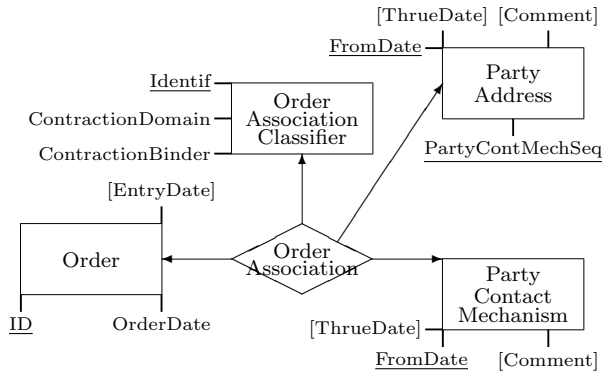


Figure 9: The Bulk Composition Within The *Order* Application

Handling of classes which are bound by the same behavior and occurrence can be simplified by this construct. In general, the composition by folding may be described as follows:

Given a *CentralType* C and associated types which are associated by a set of relationship types $\{A_1, \dots, A_n\}$ by the occurrence frame F . The occurrence frame can be \forall (if the inclusion constraints $A_i[C] \subseteq A_j[C]$ are valid for all $1 \leq i, j \leq n$) or a set of inclusion constraints. Now we combine the types $\{A_1, \dots, A_n\}$ into the type *BulkType* with the additional component *ContractionAssistant* and the attributes *Identif* (used for identification of objects in the type *ContractionAssistant* if necessary), *ContractionDomain* with $dom(ContractionDomain) = \{A_1, \dots, A_n\}$ and $dom(ContractionBinder) = F$. The general bulk composition schema is displayed in Figure 10.

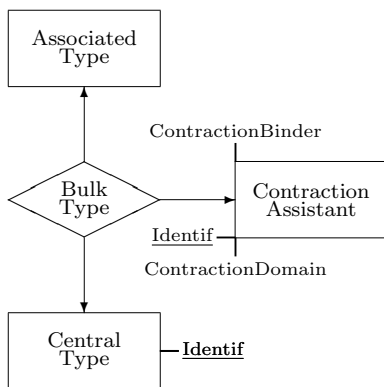


Figure 10: The General Bulk Composition

4.1.3 Architecture Composition:

Categorization-based composition have been widely used for complex structuring. The architecture of SAP R/3 often has been displayed in the form of a waffle. For this reason, we prefer to call this composition *waffle composition* or *architecture composition*.

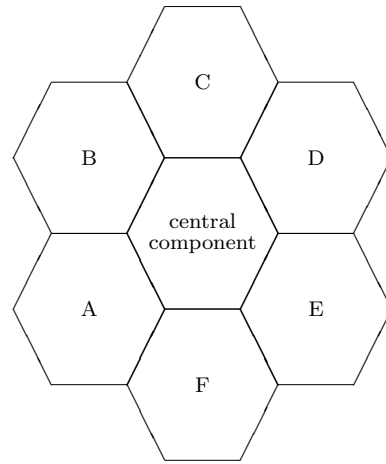


Figure 11: The Waffle Architecture Composition

The types in Figure 11 are associated with their neighbors, e.g., the schema A is associated with the central component and components B and F .

Architecture composition enables in associating through categorization and compartmentalization. This constructor is especially useful during modeling of distributed systems with local components and with local behavior. There are specific solutions for interface management, replication, encapsulation and inheritance. The cell construction is the main constructor in component applications and in data warehouse applications. Therefore, composition by categorization is the main composition approach used for in *component-based development* and in *data warehouse approaches*.

4.2 Lifespan Composition

Evolution of things in application is an orthogonal dimension which must represented in the schema from one side but which should not be mixed with constructors of the other side. We observe a number of lifespan compositions: **Evolution composition** records the *stages* of the life of things and or their corresponding objects and are closely related to *workflows*, **circulation composition** displays the phases in the lifespan of things, **incremental composition** allows to record the development and specifically the enhancement of objects, their aging and their own lifespan, **loop composition** supports nicely chaining and scaling to different perspectives of objects which seems to rotate in the workflow, and **network composition** allows the flexible treatment of objects during their evolution, support to pass objects in a variety of evolution paths and enable in multi-object collaboration.

4.2.1 Evolution Composition:

Evolution composition allows to construct a well-communicating set of types with a point-to-point data exchange among the associated types. Such evolution associations often appear in workflow applications, data flow applications, in business processes,

customer scenarios and during identification of variances. The *flow* constructor allows to construct a well-communicating set of types with a point-to-point data exchange among the associated types. Such flow associations often appear in workflow applications, data flow applications, in business processes, customer scenarios and during identification of variances.

Evolution is based on the specific treatment of stages of objects. Things are passed to teams which work on their further development. This workflow is well-specified. During evolution, things obtain a number of specific properties. The record of the evolution is based on evolution composition of object classes. Therefore, we define a specific composition in order to support the modeling, management and storage of evolution.

Evolution composition can be represented in the language of the extended entity-relationship model in two ways:

Based-On relationship types allow to represent the stepwise evolution of objects. If the lifespan is not cyclic and not too complex the representation leads to better understanding of the evolution of things to be considered in the application.

Stage-based bulk representation explicitly models the stages of the evolution process. This representation is preferable if the evolution lifespan becomes too complex.

The two approaches to evolution representation are displayed in Figure 12. For a detailed evolution model of documents we refer to (Thalheim 2001). The phases of the document evolution in an traveller setting may be distinguished by the stages *TravellerApplication*, *SupportedApplication*, *FundedApplication*, *ApprovedApplication*, and *TravelDocument*.

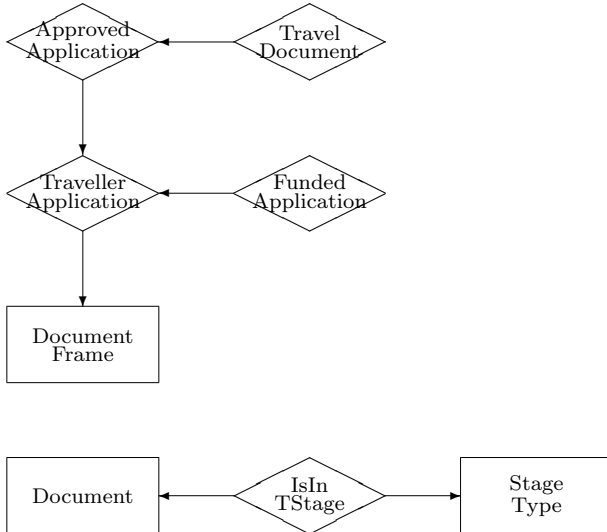


Figure 12: The Evolution of *Documents*

4.2.2 Circulation Composition:

Things may be related to each other by life cycle stages such as repetition, evolution, self-reinforcement and self-correction. Typical examples are objects representing iterative processes, recurring phenomena or time-dependent activities.

Circulation composition allows to display objects in different phases. For instance, paper handling in a conference paper submission system is based on

such phases: *PaperAbstract*, *SubmittedPaper*, *PaperInReviewing*, *AcceptedPaper*, *RejectedPaper*, *FinalVersionPaper*, *ScheduledPaperPresentation*, and *ArchivedPaperVersion*. The circulation model is supported by specific phase-based dynamic semantics (Thalheim(HERM) 2000). Circulation forms thus an iterative process.

Circulation composition can be displayed either by based-on relationship types as described above or by a type describing the circulation stage. The paper reviewing process is a process which is representation in either ways. Therefore, we may use a composition schema similar to the one in Figure 12. The stage approach is preferable however if we use dynamic integrity constraints. For instance, the type *AssignedReview* relating *SubmittedPaper* and *PCMember* has a dynamic cardinality constraint $card(AssignedReview, SubmittedPaper) = (x,5)$ with $x = 0$ in the submission phase and $x = 3$ after the submission has been closed and the paper assignment has been made.

4.2.3 Incremental Composition:

Incremental composition enables in production of new associations based on a core object. It is based on containment, sharing of common properties or resources and alternatives. Typical examples are found in applications in which processes generate multiple outcomes, collect a range of inputs, create multiple designs or manage inputs and outputs.

Incremental development enables in building layers of a system, environment, or application thus enabling in management of systems complexity. Incremental constructions may be based on intervals, may appear with a frequency and modulation. They are mainly oriented towards transport of data. Typical applications of the incremental constructor lead to the n-tier architecture and to versioning of objects. Furthermore, cooperation and synergy is supported. Typical incremental constructions appear in areas such as facility management. For instance, *incremental database schemata* are used at various stages of the architectural, building and maintenance phases in construction engineering.

A specialized incremental constructor is the *layer constructor* that is widely used in frameworks, e.g., the OSI framework for communicating processes. Incremental lifespan modeling is very common and met in almost all large applications. For instance, the schema displayed in Figure 13 uses a specific composition frame. The type *Request* is based on the the type *Quote*. Requests may be taken on their own. They are, however, coupled to quotes in order to be sent to suppliers. Thus, we have a specific variant of quote objects. The same observation can be made for types such as *Order*, *Requisition*, and *Response*.

The schema in Figure 13 points also to the need of a more flexible schema composition similar to the form used in Figure 3. This composition is similar to the composition used in engineering on the basis of connecting components, e.g. laying of cables.

This kind of construction is supported by defining input/output views of components and associating the corresponding type by channels. In the case of the schema in Figure 13 we may use a general type *ActingParty* and *ReceivingParty* and a relationship type *ProductIncremental* associating the parties with the incremental development of products.

4.2.4 Loop Composition:

Loop composition is applied whenever the lifespan of objects is cyclic or looping. They are applied for representation of objects that store chaining of events,

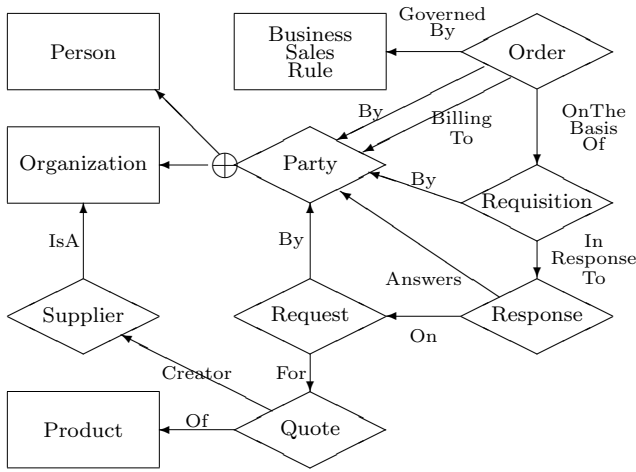


Figure 13: Incremental Composition in the *Order* Schema

people, devices or products. The loop composition is a companion of the circulation composition since it composes non-directional, non-hierarchical associations. Different modes of connectivity may be applied. Loops may be centralized and decentralized.

Loop composition models chaining and change of perspectives of events, people, devices, and other things. Loops are usually non-directional and cyclic. Temporal assignment and sharing of resources and its record, temporal association and integration, temporal rights, roles and responsibilities can be neatly represented and scaled by loop composition as long as the association is describable.

4.2.5 Network Composition:

Network or web composition enables in collecting a network of associated types to a multipoint web of associated types with specific control and data association strategies. The web has a specific data update mechanism, a specific data routing mechanism and a number of communities of users building their views on the web.

Networks are quickly evolving. They have usually an irregular growth, are built in an opportunistic manner, are rebuilt and renewed and must carry a large number of variations. Network composition enables in growth control and change management. Usually, they are supported by a multi-point center of connections, by controlled routing and replication, by change protocols, by controlled assignment and transfer, scoping and localization abstraction, and trader architecture. Further export/import converters and wrappers are supported. The database farm architecture (Thalheim 2001) with check-in and check-out facilities supports flexible network extension.

4.3 Context Composition

According to (Wisse 2001) we distinguish between the *intext* and the *context* of things which are represented by object. Intext reflects the internal structuring, associations among types and sub-schemata, the storage structuring and the representation options. Context reflects general characterizations, categorization, utilization, and general descriptions such as quality. Therefore, we distinguish between **meta-characterization composition** which is usually orthogonal to the intext structuring and can be added to each of the intext types, **utilization-recording composition** which is used to trace the running of the database engine and to restore an older state

or to reason on previous steps, and **quality composition** which allow to reason on the quality of the data provided and to apply summarization and aggregation functions in a form that is consistent with the quality of the data. The dimensionality (Feyer/Thalheim 2002) inside schemata allows to extract other context compositions. We concentrate, however, on the main compositions. All these compositions are orthogonal to the other compositions, i.e., they can be associated to any of the compositions.

4.3.1 Meta-Characterization Composition:

The meta-characterization is an orthogonal dimension applicable to a large number of types in the schema. Such characterizations in the schema in Figure 4 include language characteristics and utilization frames for presentation, printing and communication. Other orthogonal meta-characterizations are insertion/update/deletion time, keyword characterization, utilization pattern, format descriptions, utilization restrictions and rights such as copyright and costs, and technical restrictions. Meta-characterizations apply to a large number of types and should be factored out. For instance, in e-learning environments e-learning object, elements and scenes are commonly characterized by educational information such as interactivity type, learning resource type, interactivity level, age restrictions, semantic density, intended end user role, context, difficulty, utilization interval restrictions, and pedagogical and didactical parameters.

4.3.2 Utilization-Recording Composition:

Log, usage and history composition is commonly used for recording the lifespan of the database. We distinguish between **history composition** used for storing and record the log computation history in a small time slice, **usage scene composition** used to associate data to their use in business processes at a certain stage, workflow step, or scenes in an application story, and structures used to record the **actual usage**.

Data in the database may depend directly on one or more aspects of time. We distinguish three orthogonal concepts of time: temporal data types such as instants, intervals or periods, kinds of time, and temporal statements such as current (now), sequenced (at each instant of time) and nonsequenced (ignoring time). Kinds of time are: transaction time, user-defined time, validity time, and availability time.

4.3.3 Quality Composition:

Data quality is modeled by a variety of compositions. Data quality is essential whenever we need to distinguish versions of data based on their quality and reliability: **source dimension**(data source, user responsible for the data, business process, source restrictions), **intrinsic data quality** parameters (accuracy, objectivity, believability, reputation), **accessibility data quality** (accessibility, access security, contextual data quality (relevancy, value-added, timeliness, completeness, amount of information), and **representational data quality** (interpretability, ease of understanding, concise representation, consistent representation, ease of manipulation).

5 Theoretical Framework for ER Component Construction and Composition

The general component framework introduced in Section 2 must be adopted to conceptual modeling and entity-relationship models. The separation into snowflake or star schemata can be based on

general decomposition approaches. The association of schemata can be based on view cooperation approaches. The existence of an integrating mapping may be computed on the basis of equality theories. ER models rely on diagram representation. We use graph grammar rules for construction of schemata based on the ER model.

5.1 Implicit Snowflakes Within A Schema

The theoretical basis for snowflake schemata is the *structural meaning* of multi-valued and join dependencies hidden in the dependency and representable in ER-schemata:

The multi-valued dependency $X \twoheadrightarrow Y|Z$ represents a *relative structural independence*, i.e. Y -values are relatively to X -values independent on Z -values or more formally:

Given a type R and a class R^C consisting of $\text{comp}(R) = X \cup Y \cup Z$. Y is X -relatively independent from Z in R^C if the property

$\pi_Y(\sigma_{X=x}(R^C)) = \pi_Y(\sigma_{X=x, Z=z}(R^C))$
is valid for all X -values x and all Z -values z .

Therefore, a type consisting of $X \cup Y \cup Z$ can be decomposed into a schema with a type consisting of X and two relationship types relating X with Y and Z , respectively. The representation in the extended ER model is displayed in Figure 14.

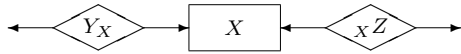


Figure 14: Multivalued Dependencies Separate Concerns

The structural independence can be extended to the dependency basis (Beeri/Kifer 1987) of the set X

$$\text{Dep}(X, \Sigma) = \{X^+ \setminus X\} \cup \{Y \mid \Sigma \models X \twoheadrightarrow Y \wedge (\forall Y' \subseteq Y (Y \neq Y' \Rightarrow \Sigma \not\models X \twoheadrightarrow Y')) \wedge \Sigma \not\models X \rightarrow Y\}$$

for the Σ -cover X^+ of X .

Proposition 1 *The dependency basis entails the decomposition of a type into the kernel type X , the extension $X^+ \setminus X$ and the relationship types relating X with the remaining elements of $\text{Dep}(X, \Sigma)$.*

In general, we can use Proposition 1 for decomposition of schemata into types which are relatively independent. This decomposition procedure can be represented by a term rewriting system TS_Σ . Obviously, the system TS_Σ terminates. Furthermore, we observe a nice correlation to properties of dependencies:

Proposition 2 *The system TS_Σ has the Church-Rosser property for a type T iff the property $\Sigma \models X \twoheadrightarrow Z, Y \twoheadrightarrow Z$ implies the property $\Sigma \models (X \cap Y) \twoheadrightarrow Z$ for all subsets X, Y, Z of the set $\text{comp}(T)$ of elements of T .*

If TS_Σ is not Church-Rosser then the type T has a variety of snowflake decompositions or viewpoints. If the system is Church-Rosser then we observe a *unique-flavor* structuring.

We can combine the decomposition of a type with the adhesion/cohesion functions (Thalheim(HERM) 2000) expressing the separability of pages of a website. Given a dependency basis $\text{Dep}(X, \Sigma) = \{(X^+ \setminus X), Y_1, \dots, Y_n\}$ and a partition of $X^+ \setminus X$ into X_1, \dots, X_m . Given an *adhesion function* ξ that assigns natural values from \mathcal{N}_0 (including 0) to each $X_1, \dots, X_m, Y_1, \dots, Y_n$. The adhesion $\xi(Z)$ is used for representing the separability or ‘distance’ of Z from X . We assume $\xi(X) = 0$.

Now we can define the *i-shell* of X as $H_i(X) = \{X_j \mid \xi(X_j) \leq i\} \cup \{Y_j \mid \xi(Y_j) \leq i\}$.

Classical normalization theory is based on a minimal non-redundant cover Σ_0 of a set of functional dependencies. All attributes A with $X \rightarrow \{A\} \in \Sigma_0$ are usually grouped into one schema. Consider, for instance, the dependencies $\text{PersonID} \rightarrow \text{Name}, \text{Address}$ and $\text{PersonID} \rightarrow \text{Account}$. It can be useful in applications to separate the two facts that a person has a name and an address and that a person has a bank account. Using *i-shells* we can separate attributes according to their distance to X .

If X does not have the unique-flavor structuring for T then we can base unique representation on shells which have unique-flavor structuring.

So far we have shown how snowflake schemata can be separated. Next we find a way to associate a schema with another schema.

5.2 View Cooperation

View integration is undecidable. It is often not desired. We can use the view cooperation concept (Thalheim(HERM) 2000) instead of that. The view cooperation concept is generalizes the concept of morphisms developed for algebraic systems (Malzew 1970). Given two schemata \mathcal{S}_1 and \mathcal{S}_2 and their corresponding database states \mathcal{S}_1^C and \mathcal{S}_2^C .

A partial schema morphism from \mathcal{S}_1 and \mathcal{S}_2 is given by a pair (f_{12}, f_{12}^C) of functions such that f_{12} is a partial embedding of types from \mathcal{S}_1 and \mathcal{S}_2 and f_{12}^C is a partial embedding of corresponding classes from \mathcal{S}_1^C into those of \mathcal{S}_2^C , i.e. $f_{12} : \mathcal{S}_1 \dashrightarrow \mathcal{S}_2$ and $f_{12}^C : \mathcal{S}_1^C \dashrightarrow \mathcal{S}_2^C$ with the property $f_{12}^C(\mathcal{S}_1^C) \models_T \mathcal{S}_2 \mid_{f_{12}(T)}$ if f_{12} is defined for T .

The diagram in Figure 15 thus commutes.

The partiality of (f_{12}, f_{12}^C) defines a view O_1^V in \mathcal{S}_1 and a view $f_{12}(O_1^V)$ in \mathcal{S}_2 .

Given further a partial schema morphism (f_{21}, f_{21}^C) from \mathcal{S}_1 and \mathcal{S}_2 .

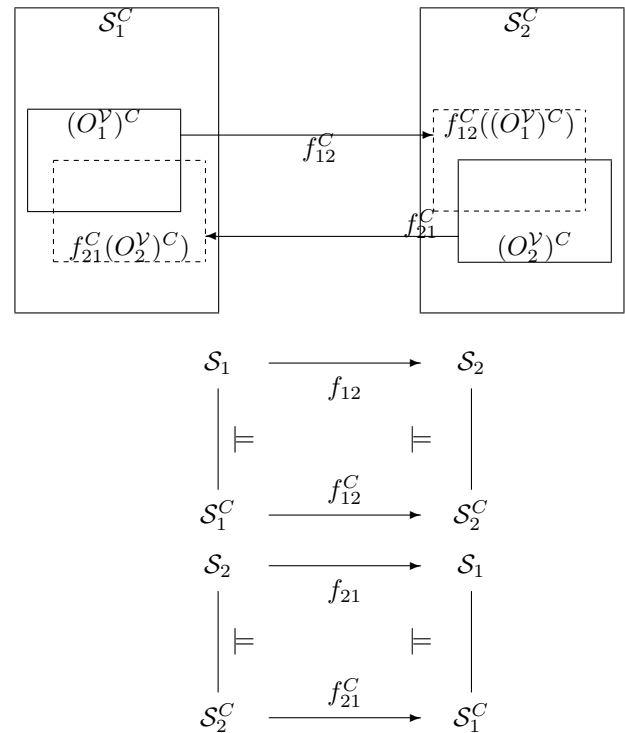


Figure 15: Partial Schema Morphisms

The two schema morphisms (f_{12}, f_{12}^C) and (f_{21}, f_{21}^C) cooperate and define a *view cooperation* if for each $T_1 \in O_1^Y \cap f_{21}(O_2^Y)$ and each $T_2 \in O_2^Y \cap f_{12}(O_1^Y)$, for each pair of corresponding classes $T_1^C \in \mathcal{S}_1^C$, $T_2^C \in \mathcal{S}_2^C$

- the functions $f_{12}^C(T_1^C), f_{21}^C(f_{12}^C(T_1^C)), f_{21}^C(T_2^C), f_{12}^C(f_{21}^C(T_2^C))$ are defined and
- the equalities $f_{21}^C(f_{12}^C(T_1^C)) = T_1^C$, $f_{12}^C(f_{21}^C(T_2^C)) = T_2^C$ are valid.

View cooperation enables in associating an input of a schema with an output of another schema. This association allows to apply composition of components as displayed in Figure 2 if we know how the functions f_{12} and f_{21} can be constructed. The next paragraph introduces a theory for associating types in one schema to types in another schema.

5.3 Term Rewriting and Equality Theories

The association of types in two schemata is restricted by the domains of the types: As introduced above two types T_1 in \mathcal{C}_1 and T_2 in \mathcal{C}_2 are *domain-compatible* if $dom(T_1) = dom(T_2)$. A schema morphism is non-trivial if O_1^Y is non-empty.

Corollary 1 *If the partial schema morphism (f_{12}, f_{12}^C) is non-trivial then the types T in O_1^Y and $f_{12}(T)$ in $f_{12}(O_1^Y)$ are domain-compatible.*

The equality of domains cannot be reduced to a subset property for types in $O_1^Y \cap f_{21}(O_2^Y)$ and in $O_2^Y \cap f_{12}(O_1^Y)$ for the view cooperation introduced above. It can be reduced to a subset property if additionally horizontal dependencies (Thalheim(HERM) 2000) are used.

Any two schema morphisms (f_{12}, f_{12}^C) and (f_{21}, f_{21}^C) allow to introduce an equality theory. We assume that the type names used in the two schemata to be integrated are disjoint. For all types T_1 in O_1^Y we add an equality $T_1 = f_{12}(T_1)$ and for all types T_2 in O_2^Y we add an equality $T_2 = f_{21}(T_2)$. We denote by \mathcal{E} the imposed equality theory.

If we are interested in a full integration the equalities may be replaced by *term rewriting rules* such as $T \rightsquigarrow f_{ij}(T)$ or $f_{ij}(T) \rightsquigarrow T$. Types have their structuring, i.e. are represented by terms of the form $T(T_1, \dots, T_m)$. Therefore, we need now rules for derivation of implied equalities or term rewritings. The following inference rules may be used:

$$\frac{\mathcal{E} \cup \{T(T_1, \dots, T_m) = S(S_1, \dots, S_m)\}}{\mathcal{E} \cup \{T_1 = S_1, \dots, T_m = S_m\}}$$

$$\frac{\mathcal{E} \cup \{T = T\}}{\mathcal{E}} \quad \frac{\mathcal{E} \cup \{T = S\}}{\mathcal{E} \cup \{S = T\}}$$

$$\frac{\mathcal{E} \cup \{T = S\}}{\vartheta_{T \rightsquigarrow S}(\mathcal{E}) \cup \{T = S\}}$$

for the substitution application $\vartheta_{T \rightsquigarrow S}$ to \mathcal{E} .

Two components are unifiable via schema morphisms if a substitution exists that relates all types associated with the views in the two morphisms. Now, the *component unification problem* is to decide whether there exist a substitution and in the positive case to compute the substitution for the schemata.

Proposition 3 *The set of inference rules is complete and consistent for the ER component unification problem.*

The component unification can be complex. In order to relate schemata with each other we can use the i-shells introduced above.

Snowflakes with the kernel types X and Y may be associated with each other. The association is based on the corresponding shells $H_i(X)$ and $H_j(Y)$. The explicit association of the elements of the covers can be based on explicit statement of synonym relationships. Those relationships can be based on equality theories.

Now, the last task in order to achieve a sound theory for ER components is the derivation of a graph replacement theory.

5.4 Graph Grammar Composition for ER Schemata

The general composition theory for ER schemata may be based on the theory of graph grammars (Ehrig/Engels/Kreowski/Rozenberg 1999, Sleep/Plasmejer/Eekelen 1993) which has been already used for the CASE tool RADD (Thalheim(HERM) 2000). In general, composition of graphs can be formulated as a pushout in the category of graphs. A graph production rule

$$\rho : L \supset K \subset L$$

consist of three components for $K \subseteq L$,
 $K \subseteq R, L \cap R = K$:

left side L which contains the sub-graph to be replaced;

right side R which is the replacement for the left side, and a so-called

'Klebegraph' (gluing graph) K which glues the parts to be replaced.

An application of a graph production rule ρ to a graph G (transformation of G) is called conflict-free if no name clashes occur. Renaming of nodes and edges in $R \setminus L$ allows to omit conflicts⁴.

The application of the graph production rule ρ to a given graph G is based on

a renaming m with $m(L \cup R) \subseteq G$ and $m(L \cup R) \cap G = m(L)$,

a context graph C which is a sub-graph of the given graph, and

an enlargement $e : K \rightarrow C$ which is bijective and allows to glue in G the sub-graph L and C along K .

The resulting graph H is the gluing of R and C along K , i.e. $H = G \setminus e(m(L \setminus R)) \cup m(R)$.

The graph transformation is denoted by $G \xrightarrow{\rho, e} H$.

There are some useful results obtained for the ER graph grammar composition using the theory of graph transformations (Ehrig et al. 1999):

Proposition 4 *The context graph C is unique up to isomorphism if $K \rightarrow L$ is injective.*

In this case, if the gluing condition is satisfied, we can remove $m(L - K)$ from G leading to the context graph. The resulting graph is constructed by gluing of R and C along K . The redex morphism $m : L \rightarrow G$ allows to embed L into G .

The set of graph production rule must satisfy the substitution theorem, i.e. none of the transformations has side effects. Any local transformation does not

⁴In the other case name conflicts do not allow to define union and intersection of graphs. In such cases we use supergraphs of the two graphs for which the relative union and intersection may be defined.

effect elements outside $L \setminus K$. In this case, graph production rules can be combined to derived graph production rules.

The graph production rule set should be confluent, satisfy the Church-Rosser property and must be acyclic. The approach considered in this paper allows to construct a confluent, terminating and acyclic system. In order to do this we must replace all equalities in \mathcal{E} by term rewriting rules.

6 Conclusion

Huge databases are usually developed over years, have schemata that carry hundreds of types and which require high abilities in reading such diagrams or schemata. We observe, however, that a large number of similarities, repetitions, and - last but not least - of similar structuring in such schemata. This paper is aiming in extraction of the main meta-similarities in schemata. These similarities are based on on **components** which are either kernel components such as star and snowflake structures, or are build by application of **compositions** such as association, architecture, bulk or constructor composition, or **lifespan composition** such as evolution, circulation, incremental, loop, and network compositions, or are **context compositions** such as meta-characterization, utilization or deployment and quality compositions.

Therefore, we can use these meta-structuring of schemata for *modularization* of schemata. Modularization eases querying, searching, reconfiguration, maintenance, integration and extension. Further, re-engineering and reuse become feasible.

Modeling based on meta-structures enables in systematic schema development, systematic extension, systematic implementation and thus allows to keep consistency in a much simpler and more comprehensible form. We claim that such structures already can be observed in small schemata. They are however very common in large schemata due to the reuse of design ideas, due to the design skills and to the inherent similarity in applications.

Meta-structuring enable also in **component-based development**. Schemata can be developed step-by-step on the basis of the skeleton of the meta-structuring. The skeleton consists of units and associations of units. Such associations combine relationship types among the types of different units. Units form components within the schema.

Component-based development enables in **industrial development** of database applications instead of handicraft developments. Handicraft developments cause later infeasible integration problems and lead to unbendable, intractable and incomprehensible database schemata of overwhelming complexity which cannot be consistently maintained. We observe, however, that the computer game industry is producing games in a manufactured, component-based fashion. This paper shows that database systems can be produced in a similar form.

References

Ancona, D. & Zucca, E. (1998), 'A theory of mixin modules: Basic and derived operators', *Mathematical Structures in Computer Science*, **8**, 4, 401-446.

Arsanjani, A. (2002), 'Developing and integrating enterprise components and services', *Communications of ACM*, **45**, 10, 31-34.

Beeri, C. & Kifer, M. (1987), 'A theory of intersection anomalies in relational database schemes', *Journal of the ACM*, **34**, 3, 544-577.

Beeri, C. & Thalheim, B. (1998), 'Identification as a primitive of database models', in Proc. FoM-LaDO'98, Kluwer, 19-36.

Biskup, J. & Polle, T. (2000), 'Decomposition of database classes under path functional dependencies and onto constraints', in Proc. FoIKS'2000, LNCS 1762, Springer, 2000, 31-49.

Broy, M. (1997), 'Compositional refinement of interactive systems', *Journal of the ACM*, **44**, 6, 850-891.

Brown, L. (2000), *Integration models - Templates for business transformation*, SAMS Publishing.

Danoch, R., Shoval, P. & Balaban, M. (2002), 'Hierarchical ER diagrams - The method and experimental evaluation', in Proc. IWCMQ'02.

Crnkovic, I., Hnich, B., Jonsson, T. & Kiziltan, Z. (2002), 'Specification, implementation and deployment of components', *Communications of ACM*, **45**, 10, 35-40.

Ehrig, H., Engels, G., Kreowski, H.-J. & Rozenberg G. (eds.) (1999), *Handbook of graph grammars and computing by graph transformations. Vol. 2: Applications, languages and tools*, World Scientific.

Feyer T. & Thalheim, B. (2002), 'Many-dimensional schema modeling', in Proc. ADBIS'02, LNCS 2435, Springer 305 - 318.

Gurevich, Y., (May 1997), 'Draft of the ASM Guide' Technical Report, Univ. of Michigan EECS Department, CSE-TR-336-97. Available from the ASM website via <http://www.eecs.umich.edu/gasm/>

Hay, D.C., (1995), *Data model pattern: Conventions of thought*, Dorset House, 1995.

Kudrjavcev, V.B. (1982), *Functional systems*, Moscov Lomonossov University Press (in Russian).

Levene, M. & Loizou, G. (2002), 'Why is the snowflake schema a good data warehouse design?', accepted for publication in *Information Systems*, 2002.

Malzew, A.I. (1970), *Algebraic systems*, Nauka (in Russian).

Moody, D.L. (2001), *Dealing with complexity: A practical method for representing large entity-relationship models*, PhD., Dept. of Information Systems, University of Melbourne.

Nierstrasz, O. & Meijler, T.D. (1995), 'Research directions in software composition', *ACM Computing Surveys*, **27**, 2, 262-264.

Schewe, K.D. & Thalheim, B. (1993), 'Fundamental concepts of object oriented databases', *Acta Cybernetica*, **11**, 4, 49 - 81.

Schewe, K.D. & Thalheim, B. (1998), 'Limitations of rule triggering systems for integrity maintenance in the context of transition specification', *Acta Cybernetica*, **13**, 277-304.

Schewe K.-D. & Thalheim, B. (1999), 'Towards a theory of consistency enforcement', *Acta Informatica*, **36**, 2, 97-141.

Schmidt, J.W. & Schring, H.-W., (1999), 'Dockets: a model for adding value to content', in Proc. ER'99, LNCS 1728, Springer, 248-262.

- Sleep, M.R., Plasmejer, M.J. & van Eekelen, M.C.J.D. (eds.) (1999), *Term graph rewriting - Theory and practice*, Wiley.
- Smith, J.M. & Smith, D.C.W. (1977), 'Data base abstractions: Aggregation and generalization', *ACM Transactions of Database Systems* **2**, 2.
- Thalheim, B. (1990), *Dependencies in relational databases*, Teubner Verlag.
- Thalheim, B. (2000), *Entity-relationship modeling - Foundations of database technology*, Springer. See also <http://www.informatik.tu-cottbus.de/~thalheim/HERM.htm>
- Thalheim, B. (2000), *The person, organization, product, production, ordering, delivery, invoice, accounting, budgeting and human resources pattern in database design*, Preprint I-07-2000, Computer Science Institute, Brandenburg University of Technology at Cottbus.
- Thalheim, B. (2001), *Abstraction layers in database structuring: The star, snowflake and hierarchical structuring*, Preprint I-13-2001, Computer Science Institute, Brandenburg University of Technology at Cottbus.
- Thalheim, B. (2002), 'Component construction of database schemes', in Proc. ER'02, LNCS 2503,20-34.
- Webster, B.F. (1995), *Pitfalls of object-oriented development: a guide for the wary and enthusiastic*, M&T books.
- Wisse, B. (2001), *Metapattern - Context and time in information models*, Addison-Wesley.
- Yuan, L.-Y. & Ozsoyoglu, Z.M. (1992), 'Design of desirable relational database schemes', *Journal of Computer and System Sciences*, **45**, 3, 435-470.