Using Meta-Structures in Database Design

Hui Ma
Victoria University of Wellington, School of Engineering and Computer Science
Wellington, New Zealand
E-mail: hui.ma@ecs.vuw.ac.nz

René Noack
Christian-Albrechts-University Kiel, Department of Computer Science
Kiel, Germany
E-mail: noack@is.informatik.uni-kiel.de

Klaus-Dieter Schewe
Software Competence Center Hagenberg, Hagenberg, Austria
E-mail: kd.schewe@scch.at

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

Keywords: database design, schema algebra, meta-structures, components, graph rewriting

Received: November 4, 2010

Practical experience shows that the design of very large database schemata causes severe problems, and no systematic support is provided. In this paper we address this problem. We define an Entity-Relationship schema algebra, which permits the representation of very large database schemata by algebraic expressions involving smaller schemata. Similar to abstraction mechanisms found in semantic data models the schema constructors can be classified into three groups for building associations and collections of subschemata, and for folding subschemata. Furthermore, based on the analysis of a large number of very large database schemata we identify twelve frequently recurring meta-structures in three categories associated with schema construction, lifespan and context. In combination with the schema algebra the meta-structures permit a component-based approach to database schema design, which can further be formalised by graph-rewriting.

Povzetek: Predstavljena je nova shema entitet in relacij za velike podatkovne baze.

1 Introduction

While data modellers learn about data modelling by means of small "toy" examples, the database schemata that are developed in practical projects tend to become very large. For instance, the relational SAP/R3 schema contains more than 21,000 tables. Moody discovered that as soon as ER schemata exceed 20 entity- and relationship types, they already become hard to read and comprehend for many developers [10].

Therefore, the common observation that very large database schemata are error-prone, hard to read and consequently difficult to maintain is not surprising at all. Common problems comprise repeated components as e.g. in the LH Cargo database schema with respect to transport data or in the SAP/R3 schema with respect to addresses.

Some remedies to the problem have already been discussed in previous work of some of the authors, and applied in some database development projects. For instance, modular techniques such as *design by units* [18] allow schemata to be drastically simplified by exploiting principles of hiding and encapsulation that are known from Software Engineering. Different subschemata are connected by bridge types. *Component engineering* [12] extends this approach by means of view-centered components with well-defined composition operators, and *hierarchy abstraction* [20] permits to model objects on various levels of detail.

In order to contribute to a systematic development of very large schemata the *co-design* approach, which integrates structure, functionality and interactivity modelling, emphasises the initial modelling of skeletons of components, which is then subject to further refinement

[21]. Thus, components representing subschemata form the building blocks, and they are integrated in skeleton schemata by means of connector types, which commonly are modelled by relationship types.

In this article we further develop the method for systematic schema development focussing on very large schemata. In Section 2 we first present an algebra for higher-order Entity-Relationship schemata [18], which permits the representation of very large schemata as algebraic expressions involving smaller and thus easier tractable schemata. Similar to abstraction mechanisms found in semantic data models [17] only three main groups of constructors are needed: *association* constructors that are used to combine schemata in a way that allows the original schemata to be regained, *folding* constructors that integrate schemata into a compact form, and *collection* constructors that deal with recurring similar subschemata. This extends our previous conference publication [7]. In particular, we permit handling schemata with constraints, and extend the description of the semantics of the operations.

In an extended theoretical study in [8] we develop a formal notion of schema morphisms, show that the corresponding category of schemata with these morphisms is finitely complete and co-complete, and also show that the algebra in this paper is well-defined and complete in the sense that all operators give rise to canonical morphisms, and all finite limits and co-limits can be expressed by the algebra. This complements our work reported in this article, which is devoted to the practical usage of the algebra for dealing with meta structures in the design of huge database schemata.

In Section 3, based on the analysis of more than 8500 database schemata, of which around 3500 should be considered very large we identify twelve frequently recurring meta-structures. These meta-structures are classified into three categories addressing schema construction, lifespan and context. This presentation polishes and extends another previous conference publication on the subject [9].

Finally, in Section 4 we address how meta-structures in combination with the schema algebra can be exploited for systematic, component-based database schema design. We analyse skeletons and subschemata more deeply and identify distinguishing dimensions [3]. Then we sketch how graph-rewriting can be used to support the design process extending and formalising existing approaches such as *design-by-units* [18], *string-bag modelling* [22], and *incremental structuring* [11].

2 An Entity-Relationship Schema Algebra

In the following we first present the gist of the Entity-Relationship model as our basis for schema design following [18]. On this basis we then describe three groups of schema constructors dealing with associations, folding, and collections of schemata. This defines a (partial) schema algebra, as constructors are only applicable, if certain preconditions are satisfied. The composition operators presented in this section will permit the construction of any schema of interest, as they mimique all set operations similar to the structural approach in [1].

2.1 Entity-Relationship Schemata

Let us briefly review the key definitions of Entity-Relationship schemata following [18]. We adopt the possibility to have higher-order relationship types and clusters, but for simplicity we disregard complex attributes, as attributes will be preserved by the schema constructors.

Thus, let $\mathcal{U}$ be a set of *attributes*. Each attribute $A \in \mathcal{U}$ is associated with a set $dom(A)$ of values called the *domain* of A.

An *entity type* (or *type of level* 0) E is defined by a finite set $attr(E) \subseteq \mathcal{U}$ of attributes and a key $k(E) \subseteq attr(E)$. The definition of an *entity* of type E is straighforward. It can be represented as a tuple $(A_1 : v_1, \ldots, A_n : v_n)$ for $attr(E) = \{A_1, \ldots, A_n\}$ and $v_i \in dom(A_i)$ for all $i = 1, \ldots, n$. An *entity set* of type E is a finite set $\{e_1, \ldots, e_m\}$ of entities of type E, such that whenever the projections $e_i[k(E)]$ and $e_j[k(E)]$ on the key coincide, then $e_i = e_j$ holds.

An *entity cluster* (or *cluster of level* 0) C is defined by a finite set $\{\ell_1 : E_1, \ldots, \ell_k : E_k\}$ with pairwise different labels ℓ_i and entity types $E_1, \ldots, E_k$ (not necessarily different). A *cluster set* of type C is defined is a labelled disjoint union $\{\ell_i : v_i \mid v_i \in \mathcal{S}(E_i)\}$ with entity sets $\mathcal{S}(E_i)$ of type E_i $(i = 1, \ldots, k)$.

A *relationship type* R *of order* $k + 1$ (or simply a *type of level* $k + 1$ with $k \geq 0$) is defined by a finite set $comp(R) = \{r_1 : R_1, \ldots, r_k : R_k\}$ with pairwise distinct role labels r_i and types or clusters R_i of level at most k, such that at least one R_i has level exactly k, a finite set $attr(R) \subseteq \mathcal{U}$ of attributes and a key $k(R) \subseteq comp(R) \cup attr(R)$. A *relationship* of type R can be represented as a tuple $(r_1 : e_1, \ldots, r_k : e_k, A_1 : v_1, \ldots, A_n : v_n)$ for $attr(R) = \{A_1, \ldots, A_n\}$ with entities or relationships e_i of type R_i, respectively, and $v_i \in dom(A_i)$.

A *cluster of level* k is defined analogously to an entity cluster with the only difference that the participating types must be of level at most k, and one of them must have level exactly k.

As an entity type E can be identified with a relationship type with an empty set of components, i.e. $comp(E) = \emptyset$, we will dispense with the separation and simply talk about types. Types of level 0 are entity types, and types of level $k > 0$ are relationship types.

An *Entity-Relationship schema* $\mathcal{S}$ (*ER-schema* for short) is a finite set of types and clusters such that, whenever $R \in \mathcal{S}$ is a relationship type, i.e. a type of level $k > 0$, and $E \in comp(R)$ is one of its components, then we must have also $E \in \mathcal{S}$, and whenever $C \in \mathcal{S}$ is a cluster, then also all types participating in C must be in $\mathcal{S}$.

If $\mathcal{S}$ is an ER-schema, a *database* over $\mathcal{S}$ is defined by

entity, relationship, and cluster sets $\mathcal{S}(R)$, respectively, for all $R \in \mathcal{S}$ such that, whenever $R \in \mathcal{S}$ is a relationship type, $r_i : R_i \in comp(R)$ is one of its components, and $(r_1 : e_1, \ldots, r_k : e_k, A_1 : v_1, \ldots, A_n : v_n) \in \mathcal{S}(R)$, then $e_i \in \mathcal{S}(R_i)$ holds, and similarly for a cluster $C = \{\ell_1 : R_1, \ldots, \ell_k : R_k\}$ we must have $\mathcal{S}(C) = \{\ell_i : e_i \mid e_i \in \mathcal{S}(R_i)\}$. Furthermore, whenever the projections $t_1[k(R)]$ and $t_2[k(R)]$ of relationships $t_1, t_2 \in \mathcal{S}(R)$ to the key $k(R)$ for a relationship type $R \in \mathcal{S}$ coincide, then already $t_1 = t_2$ holds.

In addition to the structural information that is provided by an ER-schema, a schema is usually extended by a set Σ of *integrity constraints*. These are first order formulae defined over the types and clusters in $\mathcal{S}$. In case of an extended schema $(\mathcal{S}, \Sigma)$ a database must further satisfy the constraints in Σ. In our presentation of the schema algebra constructors we will mainly deal with the schema, and the handling of integrity constraints will only mentioned briefly. If the applicability of a constructor depends on the presence of some constraints, we will mention this separately.

In the following we will commonly use the graphical representation of an ER-schema $\mathcal{S}$ by a directed graph with vertices defined by $\mathcal{S}$ and directed edges from a relationship type to all its components (labelled by the roles if necessary), as well as directed edges from a cluster type to its participating types (also labelled by the labels, if necessary). For convenience, entity types are represented by rectangles, relationship types by diamonds, and clusters by circles marked with a $+$. Attributes are usually attached to types or omitted, and keys are emphasized in some way, e.g. underlining attributes in the key and marking components in the key. We usually refer to the graphical representation of an ER-schema as an *ER-diagram*. Constraints are not indicated in ER-diagrams.

Sometimes we like to emphasize a distinguished *root* in an ER-schema. In case there is a type (or cluster) from which all other types and cluster can be reached by following the edges in the ER-diagram, this type is of course a natural choice for the root. In general, however, such a type does not exist, but there may be several types (or clusters) that cannot be reached from any other type or cluster by following component edges. Each of these types/clusters can be used as root of the schema.

2.2 Renaming

As the names of types and clusters in ER-schemata must be unique, we must avoid name clashes when applying the schema constructors. Therefore, we have to provide a renaming constructor. For this, if $R_1, \ldots, R_k$ and $R'_1, \ldots, R'_k$ are pairwise distinct sequences of names, a *renaming* is a mapping $\{R_1 \mapsto R'_1, \ldots, R_k \mapsto R'_k\}$. If $(\mathcal{S}, \Sigma)$ is an ER-schema, then replacing each occurrence of R_i in $\mathcal{S}$ and Σ by R'_i results in the schema

$$\varrho_{R_1 \mapsto R'_1, \ldots, R_k \mapsto R'_k}(\mathcal{S}, \Sigma).$$

2.3 Association Constructors

We disitinguish two kinds of association constructors: constructors that lead to schemata, into which the original schemata can be embedded as subschemata, and constructors that lead to schemata that can be projected onto the original schemata.

2.3.1 Sum and Join

The simplest form of a composition through association is by means of a direct sum, i.e. disjoint union constructor. More generally, we consider joins of two schema along input- and output-views [12]. For this let $(\mathcal{S}_i, \Sigma_i)$ be a schema with two subschemata $\mathcal{I}_i \subseteq \mathcal{S}_i$ called *input view*, and $\mathcal{O}_i \subseteq \mathcal{S}_i$ called *output-view* ($i = 1, 2$). We request that $\mathcal{I}_i$ and $\mathcal{O}_j$ for $i = 1, j = 2$ or $i = 2, j = 1$ are isomorphic in a purely graph-theoretic sense (not as in [8]), i.e. there exists a graph-isomorphism $\sigma : \mathcal{I}_i \to \mathcal{O}_j$.

The *join schema*

$$\mathcal{S} = \mathcal{S}_1 \bowtie_{\mathcal{I}_1 := \mathcal{O}_2 \| \mathcal{I}_2 := \mathcal{O}_1} \mathcal{S}_2$$

results from the two given schemata by identifying in $\mathcal{S}_1 \cup \mathcal{S}_2$ the input-view of first schema with the output-view of the second one and vice versa. That is, we rename $\mathcal{S}_i$ in a way that the subschemata $\mathcal{I}_1$ and $\mathcal{O}_2$ (and likewise $\mathcal{I}_2$ and $\mathcal{O}_1$) become identical, while all other types are different, and then build the union. Attribute sets for types that are identified are merged by using set union.

Furthermore, if a type the subschemata $\mathcal{I}_1$ and $\mathcal{O}_2$ has components outside the subschema, these components will be preserved in the join. This applies analogously to $\mathcal{I}_2$ and $\mathcal{O}_1$. This may have the effect that an entity-type in one of the views becomes a relationship type in the join schema. The set Σ of constraints on $\mathcal{S}$ is defined by the union $\Sigma_1 \cup \Sigma_2$ after the renaming. In this way the original schemata $\mathcal{S}_i$ become subschemata of the join schema $\mathcal{S}$, and consequently, each database over $\mathcal{S}$ can be mapped to a database over $\mathcal{S}_i$.

The join with empty input-and output views is the *direct sum* $\mathcal{S}_1 \oplus \mathcal{S}_2$. The join of the schemata $\mathcal{S}_1$ and $\mathcal{S}_2$ along the input- and output-views shown in Figure 1 is the schema $\mathcal{S}$ shown in the same figure. We omitted all attributes, as these are preserved by the join.

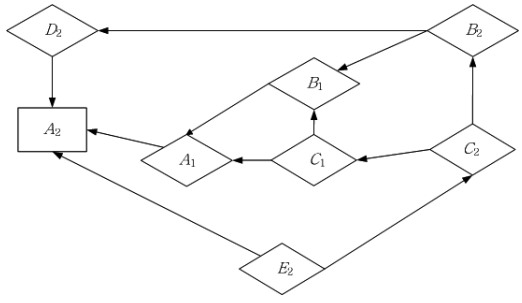

Figure 2: The reference-join on two schemata.

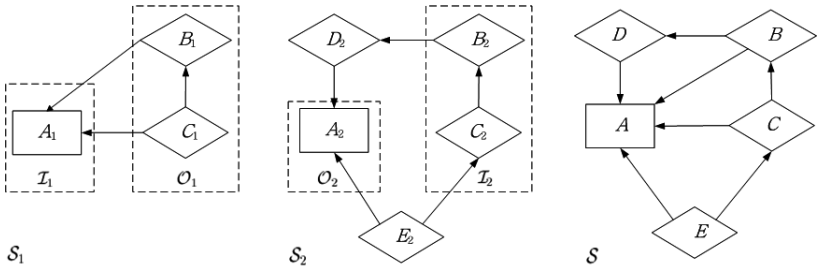

Figure 1: The join operator on two schemata.

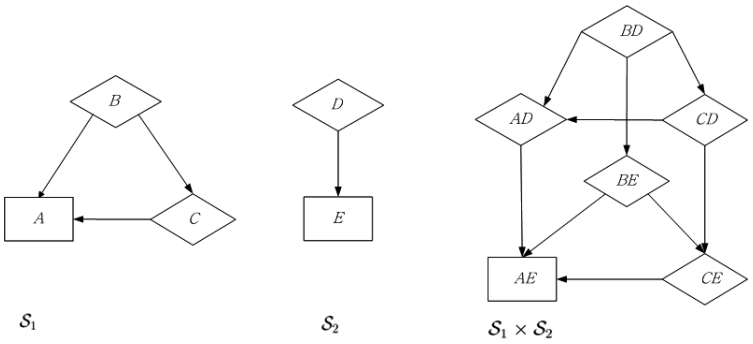

Figure 3: The product operator on two schemata.

A variant of the join operator is provided by means of a *reference-join*. The prerequisites are the same as for the join operator, only that we need that the set of type names of both schemata are disjoint. In this case, however, the output-views $\mathcal{O}_i$ $(i = 1, 2)$ of the original schemata are preserved within the resulting schema

$$\mathcal{S} = \mathcal{S}_1 \bowtie_{\mathcal{I}_1 \to \mathcal{O}_2 \| \mathcal{I}_2 \to \mathcal{O}_1} \mathcal{S}_2$$

and references from the types in $\mathcal{I}_i$ to those in $\mathcal{O}_j$ for $(i, j) = (1, 2)$ or $(2, 1)$ are added. This requires that entity-types in the input-views be turned into relationship types. The schema shown in Figure 2 shows the result of the reference-join of the schemata $\mathcal{S}_1$ and $\mathcal{S}_2$ from Figure 1. In this case, the set of constraints Σ associated with $\mathcal{S}$ is simply defined by the union $\Sigma = \Sigma_1 \cup \Sigma_2$.

Another variant can be obtained, when cooperating views [18] are employed instead of merging input- and output-views or letting the former ones reference the latter ones. In this case the data exchange has to be specified explicitly by means of operations. As we neglected operations in our model, we have to discard this alternative for the presentation here.

2.3.2 Product and Meet

Dual to the sum constructor we can define a product constructor. In this case let $(\mathcal{S}_i, \Sigma_i)$ be schemata with disjoint name sets $(i = 1, 2)$. For types $R_i \in \mathcal{S}_i$ defined as $(comp(R_i), attr(R_i), k(R_i))$ $(i = 1, 2)$ define their product $R_1 \times R_2$ by the type

$$R_{1,2} = (comp(R_1) \times \{R_2\} \cup \{R_1\} \times comp(R_2), \\ attr(R_1) \cup attr(R_2), k(R_{12})),$$

i.e. if $comp(R_i) = \{r_{i1} : R_{i1}, \ldots, r_{ik_i} : R_{ik_i}\}$, we obtain $comp(R_{12}) = \{r_{11} : R_{11,2}, \ldots, r_{1k_1} : R_{1k_1,2}, r_{21} : R_{1,21}, \ldots, r_{2k_2} : R_{1,2k_2}\}$, and the key $k(R_{12})$ is defined as $\{r_{1j} : R_{1j,2} \mid r_{1j} : R_{1j} \in k(R_1)\} \cup \{r_{2j} : R_{1,2j} \mid r_{2j} : R_{2j} \in k(R_2)\} \cup \{A \mid A \in attr(R_1) \cap k(R_1)\} \cup \{A \mid A \in attr(R_2) \cap k(R_2)\}$.

If R_1 is a cluster, say $R_1 = \{\ell_1 : R_{11}, \ldots, \ell_{k_1} : R_{1k_1}\}$, and R_2 is a type as before, then their product is the cluster

$$R_1 \times R_2 = \{\ell_1 : R_{11,2}, \ldots, \ell_{k_1} : R_{1k_1,2}\}.$$

The product of a type R_1 and a cluster R_2 is defined analogously. Finally, if both R_1 and R_2 are clusters, say $R_i = \{\ell_{i1} : R_{i1}, \ldots, \ell_{ik_i} : R_{ik_i}\}$ for $i = 1, 2$, then their product is the cluster

$$R_1 \times R_2 = \\ \{\ell_{1j_1,2j2} : R_{1j_1,2j_2} \mid 1 \leq j_1 \leq k_1,\ 1 \leq j_2 \leq k_2\}.$$

The *product schema* is defined as

$$\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 = \{R_1 \times R_2 \mid R_1 \in \mathcal{S}_1,\ R_2 \in \mathcal{S}_2\}.$$

Of course, in all cases we have to create new names for the new types (or clusters) $R_{i,j} = R_i \times R_j$, and also new names for labels in the clusters and roles in the components. Figure 3 shows the product $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$ of the schemata in the same figure. We omitted all attributes.

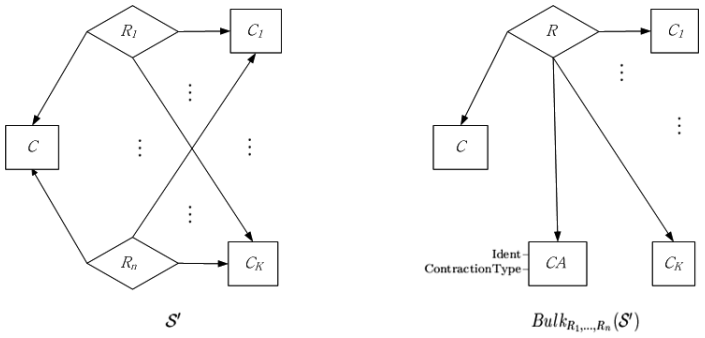

Figure 4: The bulk operator on a database schema.

Each product type (or cluster) $R_1 \times R_2 \in \mathcal{S}_1 \times \mathcal{S}_2$ contains the roles and attributes from R_1 and R_2, and thus defines projections $R_1 \times R_2[R_i]$ for $i = 1, 2$. Thus, whenever a constraint in Σ_i refers to a type R, this type may be replaced by any projection $R \times R'[R]$ (or $R' \times R[R]$, respectively). Taking all the constraints defined in this way defines the set of constraints $\hat{\Sigma}_i$, and the set of constraints on $\mathcal{S}$ is defined by the union $\Sigma = \hat{\Sigma}_1 \cup \hat{\Sigma}_2$.

In this way, similar to the case of the join-operator, a database over a product schema $\mathcal{S}_1 \times \mathcal{S}_2$ can be projected to a database over the original schemata $\mathcal{S}_i$ $(i = 1, 2)$.

We can also define a dual *meet constructor* $\bullet_\varphi$ for the join constructor. In this case we need an additional *matching condition* φ, and we define the meet schema as

$$\mathcal{S} = \mathcal{S}_1 \bullet_\varphi \mathcal{S}_2 = \{R_1 \times R_2 \mid R_1 \in \mathcal{S}_1,\ R_2 \in \mathcal{S}_2 \text{ with } \varphi(R_1, R_2)\}.$$

Matching conditions can express requirements such as common attributes or inclusion constraints.

2.4 Folding and Unfolding of Schemata

As observed in [9] similar subchemata can be integrated by replacing a number of relationship types by a new relationship type plus an additional entity type. For this assume we have a schema $\mathcal{S}'$ with a central entity (or relationship) type C, and n relationship types $R_1, \ldots, R_n$ that all relate C to a number $C_1, \ldots, C_k$ of entity or relationship types as shown in the left hand part of Figure 4.

Then we can replace $R_1, \ldots, R_n$ by a new relationship type R with a new additional component CA. This type must have an attribute "ContractionType" with domain $\{1, \ldots, n\}$ that will be used to identify the original relation. It may further be advisable to add an identifying attribute "Ident".

The schema $\mathcal{S} = Bulk_{R_1,\ldots,R_n}(\mathcal{S}')$ resulting from applying this *bulk* operator is illustrated in the right hand part of Figure 4. With respect to integrity constraints in Σ each occurrence of a type R_i has to be replaced by the projection $R[C, C_1, \ldots, C_k]$ and the condition $R.CA.\text{ContractionType} = R_i$ has to be added.

The bulk constructor $Bulk_{R_1,\ldots,R_n}$ can be refined to better handle attributes that are not common to all types $R_1, \ldots, R_n$. Such an attribute A becomes an "optional" attribute of the type CA, i.e. its domain will be defined as $dom_{\mathcal{S}}(A) = dom_{\mathcal{S}'}(A) \cup \{\mathit{undef}\}$. If A is not an attribute of the type R_i, the constraint

$$CA.\text{ContractionType} = R_i \Rightarrow CA.A = \mathit{undef}$$

has to be added to Σ.

Semantically, it is easy to see how databases over $\mathcal{S}'$ are mapped onto databases over $\mathcal{S} = Bulk_{R_1,\ldots,R_n}(\mathcal{S}')$. We get $\mathcal{S}(C) = \mathcal{S}'(C)$ and $\mathcal{S}(C_i) = \mathcal{S}'(C_i)$ for $i = 1, \ldots, k$, $\mathcal{S}(CA) = \{(\text{Ident} : i, \text{ContractionType} : R_i \mid i = 1, \ldots, n\}$, and $\mathcal{S}(R) = \bigcup_{i=1}^{n} \{\hat{t}_i \mid t_i \in \mathcal{S}'(R_i)\}$, where the tuple $\hat{t}_i$ results from t_i by adding the role $(CA : i)$.

The *expansion* constructor $Expand_{E:A}$ is inverse to the bulk constructor. In this case we need an entity type E with $k(E) = \{ident\}$, and an attribute $A \in attr(E) - k(E)$ with a finite enumeration domain $dom(A) = \{v_1, \ldots, v_n\}$. Furthermore, there must be a unique relationship type $R \in \mathcal{S}$ with a component E occurring once, i.e. $r : E \in comp(R)$, and for all r' and all R' with $r' : E \in comp(R')$ we must have $R' = R$ and $r' = r$.

In the resulting schema $Expand_{E:A}(\mathcal{S})$ the type R will be replaced by n types $R_1, \ldots, R_n$ corresponding to the values $v_1, \ldots, v_n$ of the attribute A. For each of these types we have $comp(R_i) = comp(R) - \{r : E\}$. Each attribute of R becomes an attribute of R_i, and each attribute $B \in attr(E) - \{ident,\ A\}$ is added as an attribute of R_i, unless Σ contains a constraint of the form above.

The mapping of databases over $\mathcal{S}$ to databases over $Expand_{E:A}(\mathcal{S})$ is just the inverse of the mapping for the bulk operator: "forget" $\mathcal{S}(CA)$ and split $\mathcal{S}(R)$ into n sets according to the value of the CA role.

Component nesting can be applied to a schema $\mathcal{S}_1$ to replace a component C of a type R by a complete subschema $\mathcal{S}_2$ that is rooted at a type T. Attributes, identifying components $I_1, \ldots, I_k$ and other components $C_1, \ldots, C_\ell$ of C will become components of the root type T of $\mathcal{S}_2$ within the new schema. We denote the schema $\mathcal{S}$ resulting from the application of the nesting operator by $nest_{C:\mathcal{S}_2(T)}(\mathcal{S}_1)$. Figure 5 illustrates the application of the nesting operator.

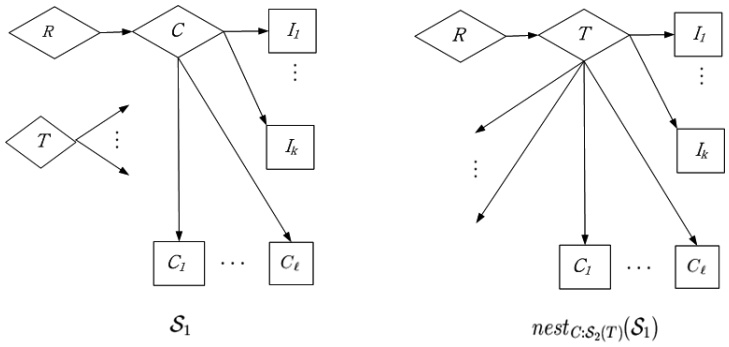

Figure 5: The nesting operator on a database schema.

If we define an input-view $\mathcal{I}_1 = \{C\}$ for $\mathcal{S}_1$, an output-view $\mathcal{O}_2 = \{T\}$ for $\mathcal{S}_2$, and let $\mathcal{O}_1 = \emptyset = \mathcal{I}_2$, then component nesting is actually a special case of a join. Component nesting is usually applied with a type C that has not yet been developed, i.e. it is an entity type in $\mathcal{S}_1$. It generalises entity model clustering, entity clustering, entity and relationship clustering, entity tree clustering in the design-by-units method [18].

2.5 Collection Constructors for Schemata

While all operators discussed so far have arity 1 or 2, the collection constructions apply to any number k of schemata. If $\mathcal{S}_1, \ldots, \mathcal{S}_k$ are schemata, we can build the *set schema* $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, provided the element schemata are pairwise distinct, the *multiset schema* $\langle \mathcal{S}_1, \ldots, \mathcal{S}_k \rangle$, the *list schema* $[\mathcal{S}_1, \ldots, \mathcal{S}_k]$, and the *tree schema* $\langle\langle \mathcal{S}_1, \ldots, \mathcal{S}_k \rangle\rangle$.

As schemata, the result of the first three constructions can be identified with the sum, i.e. the join with empty views, of the element schemata, while a tree schema contains an additional relationship type with k components that are root types of the element schemata. Renaming has to be applied in all cases to avoid name clashes, and for constraint sets the union operator is used. As such, the collection constructions are only a mild extension.

However, they unfold their power by means of collection operators that can be applied to a set, multiset, list or tree schema $\mathcal{S}'$:

- $all\_of(\mathcal{S}')$ denotes the schemata that contains all schemata in the collection as subschemata. The construction can be used to specify that all $\mathcal{S}_1, \ldots, \mathcal{S}_k$ (or their root types, respectively) must appear as components in some other construction, e.g. in the bulk or nesting construction we discussed above.

- Similarly, $any\_of(\mathcal{S}')$ denotes one arbitrary element schema, and $n\_of(\mathcal{S}')$ denotes an arbitrary selection of n of the element schemata. Semantically, this leads to the disjoint union of databases, i.e. the original databases are embedded in the resultung databases after applying the operator.

- The selection of subschemata in the collection using any of the constructors all_of, any_of or n_of can be refined by adding selection criteria in form of a `where`-clause. For instance, $n\_of(\{\mathcal{S}_1, \ldots, \mathcal{S}_k\})$ `where` φ would select n of the element schemata among those satisfying the condition φ.

- $n\_th(\mathcal{S}')$ for a list or tree schema denotes the n'th element schema, provided $1 \leq n \leq k$ is satisfied.

As an example consider again the schema $\mathcal{S}'$ in Figure 4. If we define schemata $\mathcal{S}_i$ for $i = 0, \ldots, k$ to contain only one type – C_i for $i \neq 0$ and C for $i = 0$ – then we could define the types R_i as

$$R_i = (all\_of(\{\mathcal{S}_0, \ldots, \mathcal{S}_k\}), \mathcal{S}, \mathcal{K}),$$

i.e. the components are the (root) types in $\mathcal{S}_i$, while the set of attributes $\mathcal{A}$ and the keys $\mathcal{K}$ are specified elsewhere. Alternatively, if $\mathcal{A} = \emptyset$, we could define R_i as the root type in the schema $Tree(\mathcal{S}_0, \ldots, \mathcal{S}_k)$.

Similarly, the type R in the schema $\mathcal{S} = Bulk_{R_1,\ldots,R_n}(\mathcal{S}')$ can be defined as

$$R = \{CA\} \cup all\_of(\{\mathcal{S}_0, \ldots, \mathcal{S}_k\}), \mathcal{S}, \mathcal{K})$$

with the entity type $CA = (\{\text{Ident, ContractionType}\}, \{\text{Ident}\})$.

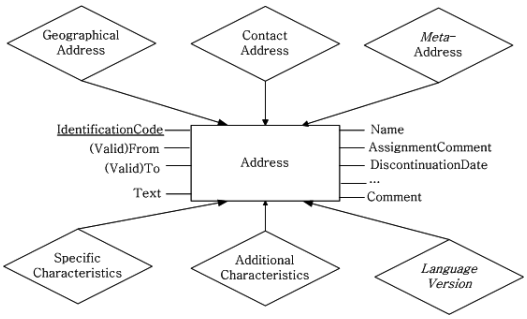

Figure 6: The General Structure of Addresses.

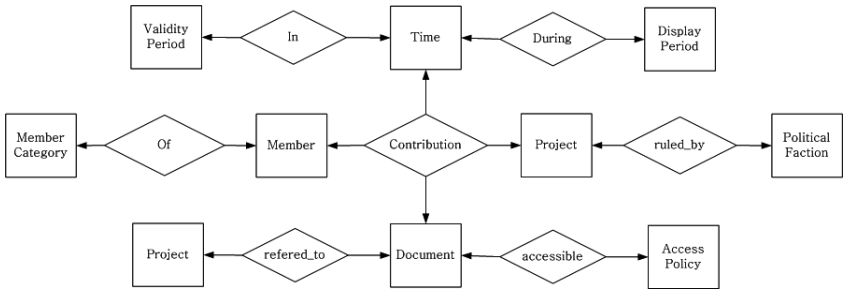

Figure 7: Snowflake Schema on Contributions.

3 Meta-Structures in Very Large Database Schemata

Based on an extensive study of a large number of conceptual database schemata – we analysed more than 8500 database schemata, of which around 3500 should be considered being very large – we identify frequently occurring meta-structures and classify them in three categories according to construction, lifespan and context. In the following we describe these meta-structures.

3.1 Construction Meta-Structures

Structures are based on building blocks such as attributes, entity types and relationship types. In order to capture also versions, variations, specialisations, application restrictions, etc. structures can become rather complex. As observed in [12, 14] complex structures can be primarily described on the basis of *star* and *snowflake meta-structures*. In addition, *bulk meta-structures* describing the similarity between things and thus enable generalisation and combination, and *architecture meta-structures* describe the internal construction by building blocks and the interfaces between them.

3.1.1 Star and Snowflake Meta-Structures

Star typing has been used already for a long time outside the database community. The star constructor permits to construct associations within systems that are characterized by complex branching, diversification and distribution alternatives. Such structures appear in a number of situations such as composition and consolidation, complex branching analysis and decision support systems.

A *star meta-structure* is characterized either by a (core) entity type E and a number of (peripheral) subtypes, i.e. unary relationship types R_i with $comp(R_i) = \{E\}$ $(i = 1, \ldots, n)$, or by a core (level 1) relationship type R together with its components, which are of course entity types. In the former case the core type is usually used for storing basic data, and the subtypes are used to capture additional properties [20]. Such a star structure is shown in Figure 6 with the entity type `Address` as its core. Taking the relationship type `Contribution` in Figure 7 as core type of a star schema, the subschema containing `Contribution`, `Member`, `Document`, `Project`, and `Time` defines another star structure.

We consider star structures as the simplest schemata, which naturally appear as subschemata of any conceptual schema. However, if the core type is an entity type, even a simple star schema can be written as the join of several schemata (in any order). For instance, the star schema in Figure 6 can be composed out of six small schemata, each consisting of the entity type `Address` and a single subtype such as `GeographicalAddress` or `ContactAddress`. For building the joins we always have to take the subschema $\{$`Address`$\}$ as input- and output-schemata, respectively.

A slighly more complicated meta-structure arises, if we take a star schema $\mathcal{S}_1$ and apply the nesting operator $nest_{C:\mathcal{S}_2(T)}$ with a type T in another star schema $\mathcal{S}_2$ to one of its peripheral types C. More generally, as nesting is a special case of the join-operator, we could apply the join $\bowtie_{\mathcal{I}_1:=\mathcal{O}_2 \| \mathcal{I}_2:=\mathcal{O}_1}$ with $\mathcal{I}_1$ containing several peripheral types of the star schema $\mathcal{S}_1$, $\mathcal{O}_2$ containing several types of another star schema $\mathcal{S}_2$, and $\mathcal{I}_2 = \mathcal{O}_1 = \emptyset$. This procedure may be applied repeatedly. In all these cases the result is called a *snowflake schema*.

For example, the snowflake schema in Figure 7 – for simplicity, attributes have been omitted – represents the information structure of documented contributions of members of working groups during certain time periods. In this case the schema result from extending the original star schema with core relationship type `Contribution` by means of nesting and join with six star schemata centred around the relationship types `In`, `During`, `Of`, `ruled_by`, `referred_to`, and `accessible`, respectively.

Star and snowflake schemata are common in data warehouses and OLAP systems [6].

3.1.2 Bulk Meta-Structures

A *bulk meta-structure* is represented by a schema that results from the application of the bulk-operator, i.e. $\mathcal{S} = Bulk_{R_1,\ldots,R_n}(\mathcal{S}')$. Thus, in a bulk structure types that are used in a very similar way are clustered together. Apply-

ing the expand-operator $Expand_{E:A}$ to the bulk structure $\mathcal{S}$ returns the original schema $\mathcal{S}'$. Thus, a bulk structure is merely a compacted representation for sructurally similar information.

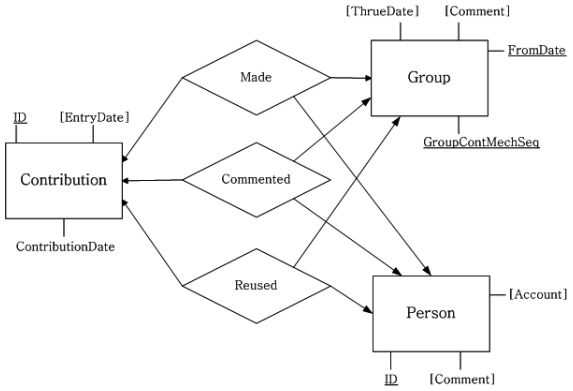

Figure 8: E-Community Application.

Let us exemplify this approach for the commenting process in an e-community application. The relationship types `Made`, `Commented`, and `Reused` in Figure 8 are all similar. They associate contributions with both `Group` and `Person`. They are used together and at the same objects, i.e. each contribution object is at the same time associated with one group and one person.

We can combine the three relationship types into the type `ContributionAssociation` as shown in Figure 8. The type `ContributionAssociationClassifier` and the domain {Made, Commented, Reused} for the attribute `ContractionDomain` can be used to reconstruct the three original relationship types. The handling of classes that are bound by the same behaviour and occurrence can be simplified by this construction.

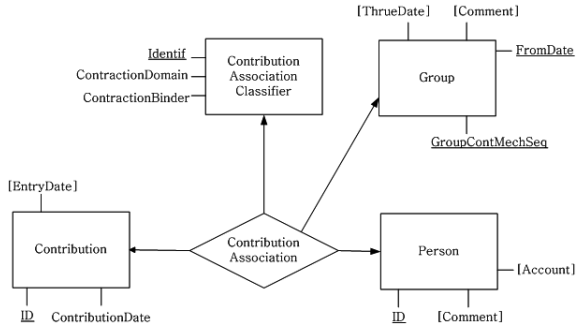

Figure 9: Bulk Meta-Structure for E-Community.

If $\mathcal{S}'$ denotes the schema in Figure 8, and $\mathcal{S}$ the one in Figure 9, we have

$$\mathcal{S} = Bulk_{\text{Made, Commented, Reused}}(\mathcal{S}') \qquad \text{and}$$
$$\mathcal{S}' = Expand_{\text{ContributionAssociationClassifier: ContractionDomain}}(\mathcal{S}) \ .$$

3.1.3 Architecture and Constructor-based Meta-Structures

Categorisation and compartment building have been widely used for modelling complex structures. For instance, the architecture of SAP R/3 has often been displayed in form of a waffle. That is, the schema is constructed out of several subschemata that are integrated by means of *bridge* or *binding schemata*. Technically, this integration is performed by means of joins involving two subschemata and their bridge schema.

We illustrate the building of a waffle structure in Figure 10. All subschemata are sketched by hexagons, and thebinding schemata are sketched as ovals.

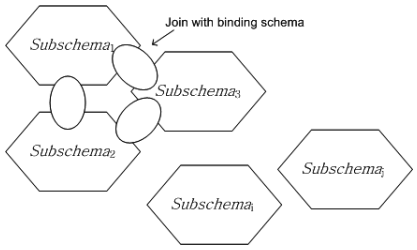

Figure 10: Waffle Meta-Structure.

Therefore, we adopt the term *waffle meta-structure* or *architecture meta-structure* for structures that arise this way. These meta-structures are especially useful for the modelling of distributed systems with local components and behaviour. They provide solutions for interface management, replication, encapsulation and inheritance, and are predominant in component-based development and data warehouse modelling.

3.2 Lifespan Meta-Structures

The evolution of an application over its lifetime is orthogonal to the construction. This leads to a number of *lifespan meta-structures*, which we describe next. *Evolution meta-structures* record life stages similar to workflows, *circulation* or *loop meta-structures* display the phases in the lifespan of objects, e.g. chaining and scaling to different perspectives of objects, *incremental meta-structures* permit the recording of the development, enhancement and ageing of objects, and *network meta-structures* permit the flexible treatment of objects during their evolution by supporting to pass objects in a variety of evolution paths and enable multi-object collaboration.

All these lifespan structures are determined by three dimensions: *expansion*, *seed*, and *feedback*. The expansion dimension captures the development of objects using a starting (entity) type that is stepwise expanded by relationship types as shown in Figure 11. Besides the added new relationship type in the i'th expansion step having the added type of the $(i-1)$'th expansion step as one of its components other types may be added to the schema and

identified with existing types. If the expansion dimension is the only one used, we obtain an incremental lifespan meta-structure as discussed below.

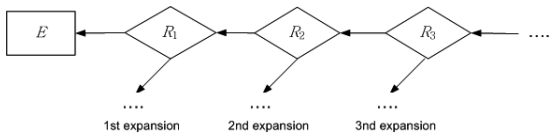

Figure 11: Expansion Dimension in Lifespan Meta-Structures.

The seed dimension captures the spreading of an objects into several related objects, thus producing a tree of types as illustrated in Figure 12. For instance, the entity type E may be `book`, and the relationship type R_1 may be `book_copy`. The technical difference to the expansion dimension is by means of participation cardinality constraints – these have been omitted in Figure 12. For expansion we have to request $card(R_i, R_{i-1}) = (0, 1)$ (with $R_0 = E$), i.e. for each entity of type E appears as a component of at most one relationship of type R_i ($i > 0$), which means that we deal with different lifespan versions of the same object. For seed the corresponding participation cardinality constraints are $card(R_i, R_{i-1}) = (1, \infty)$, i.e. each entity of type E spreads out into many relationships of type R_i ($i > 0$), which means that we do not deal with the same object, but with different levels of abstraction as book, book edition and book copy. If the seed dimension is the only one used, we obtain an evolution lifespan meta-structure as discussed below.

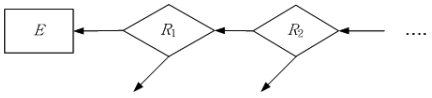

Figure 12: Seed Dimension in Lifespan Meta-Structures.

The feedback dimension captures the case of cyclic development as illustrated in Figure 13. In this case we will need relationship types linking the different stages. Alternatively, star or snowflake schemata could be used with a core entity type representing the developing object and the peripheral types modelling the various stages. If the feedback dimension is the only one used, we obtain a loop or circular lifespan meta-structure as discussed below.

For all three dimensions the basic meta-structure can be formalised by using the join-operator. This also applies, if the lifespan meta-structures is to be combined with a structural meta-structure, in which the type to be developed appears. Similarly, if several lifespan meta-structures appear together, this is reflected by the use of the product- and meet-operators. As circular (feedback) and incremental (expansion) cannot be combined, the only reasonable combined lifespan meta-structure is the network meta-structure, in which seed comes together with either incremental or loop development.

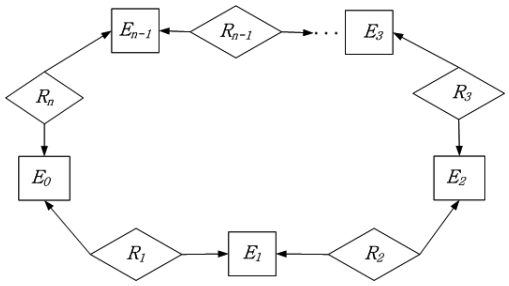

Figure 13: Feedback Dimension in Lifespan Meta-Structures.

3.2.1 Incremental Meta-Structures

Incremental meta-structures enable the production of new associations based on a core object. They employ containment, sharing of common properties or resources, and alternatives. Typical examples are found in applications, in which processes collect a range of inputs, generate multiple outcomes, or create multiple designs.

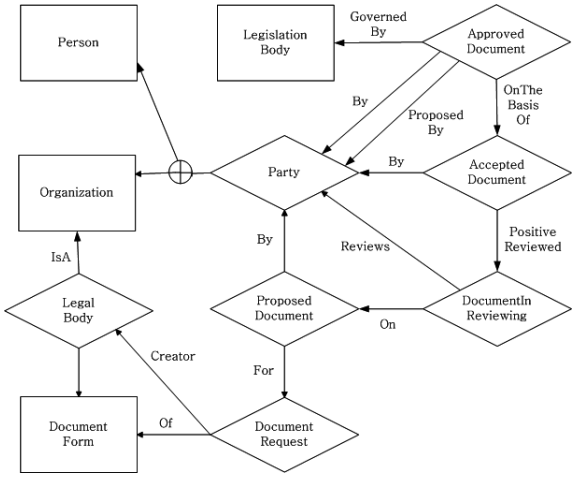

Figure 14: Incremental Meta-Structure.

Incremental development builds layers of an application with a focus on the transport of data and cooperation, thereby enabling the management of systems complexity. It is quite common that this leads to a multi-tier architecture and object versioning. Typical incremental constructions appear in areas such as facility management [4]. A special *layer constructor* is widely used in frameworks, e.g. the OSI framework for communicating processes.

As an example take the schema in Figure 14, which expands the single entity type E = DocumentForm in five steps. In the first step the relationship type R_1 = `DocumentRequest` is added, but for this also the types `LegalBody` and `Organization` are required. In the second step the relationship type

R_2 = ProposedDocument is added, which requires in addition the type Party, which is a cluster of Organization and the new type Person. In step three the added relationship type is R_3 = DocumentInReviewing, which links again to Party; in step four we have to add R_4 = AcceptedDocument, again linking to Party. In the final step the added relationship type is R_5 = ApprovedDocument with two additional roles to Party, and LegislationBody as additional fourth component.

The sequence $E, R_1, \ldots, R_5$ reflects the incremental development of a legal document in the e-governance application SeSAM. It uses a specific composition frame, i.e. the type DocumentInReviewing is based on the type ProposedDocument. Legal documents typically employ particular document patterns, which are represented by the type DocumentForm. Actors in this applications are of type Party, which generalises Person and Organisation.

Formally, an incremental lifespan meta-structures results from a sequence of join operations.

3.2.2 Evolution Meta-Structures

Objects in a database may have a number of stages. Evolution meta-structures are characterised by repetition and evolution cycles for self-correction and self-reinforcement. The core of such meta-structures is the repetition of stages of objects. We may differentiate between linear evolution models and cyclic evolution models. The former one uses non-repeatable, non-iterative specialisation schemata.

By using a *flow* constructor evolution meta-structures permit the construction of a well-communicating set of types with a P2P data exchange among the associated types. Such associations often appear in workflow applications, business processes, customer scenarios, and when when identifying variances. Evolution is based on the treatment of *stages* of objects. Objects are passed to handling agents (teams), which maintain and update their specific properties.

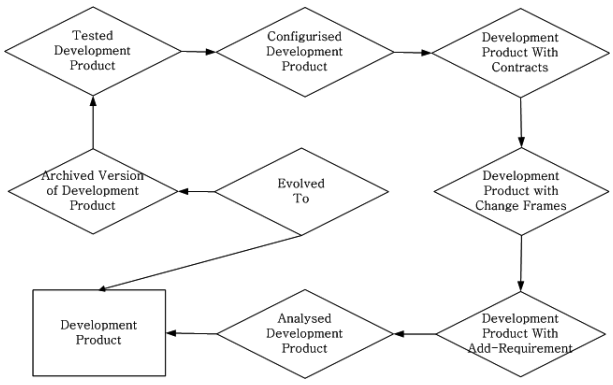

Figure 15: Evolution Meta-Structure for Software Project Management.

As an example consider the schema in Figure 15, which illustrates cyclic evolution for the support of software project management. Software development processes involve a number of actors or stakeholders, which are typically repeatable, defined, managed and optimised. Processes follow an internal work organisation, and products are analysed before requirements for development are applied. The company has developed frames or templates for changes within a product, and the requested changes are contracted to sub-divisions and sub-contractors. Finally, the next product is stored after testing and integration has been conducted. The relationship type EvolvedTo has been introduced for an explicit separation of generations or versions of development products.

3.2.3 Loop or Circulation Meta-Structures

These meta-structures appear whenever the lifespan of objects contains cycles. They are used for the representation of objects that store chains of events, people, devices, products, etc. Similar to the circulation meta-structure it employs non-directional, non-hierarchical associations with different modes of connectivity being applicable. In this way temporal assignment and sharing of resources, association and integration, rights and responsibilities can be neatly represented and scaled.

In circulation meta-structures objects may be related to each other by life-cycle stages such as repetition, self-reinforcement and self-correction. Typical examples are objects representing iterative processes, recurring phenomena or time-dependent activities. A circulation meta-structure supports primarily iterative processes.

Circulation meta-structures permit to display objects in different phases. For instance, legal document handling in the SeSAM e-government system is based on such phases, and a loop meta-structure provides an alternative to the incremental meta-structure in Figure 14.

As an example consider the schema sketch in Figure 16 dealing with document handling in a very general way. Though document handling may vary in various ways, we may assume an *inductive construction*, i.e. each document is constructed on the basis of simpler documents and base documents. The lower part of the snowflake schema addresses aspects of raw documents such as legal aspects, format, encoding, associations and contract involvement. These capture static aspects of a document. The upper part of the schema captures dynamic aspects that evolve over time. In particular, the operational document captures data entry into the document in relation to a rather complex workflow with several stages, different associated actors, various responsibilities and different stages of preparations. This leads to the almost completed blueprint and the completed submission document. Documents that are no longer subject to change are stored in an archive together with a summary or docket [13].

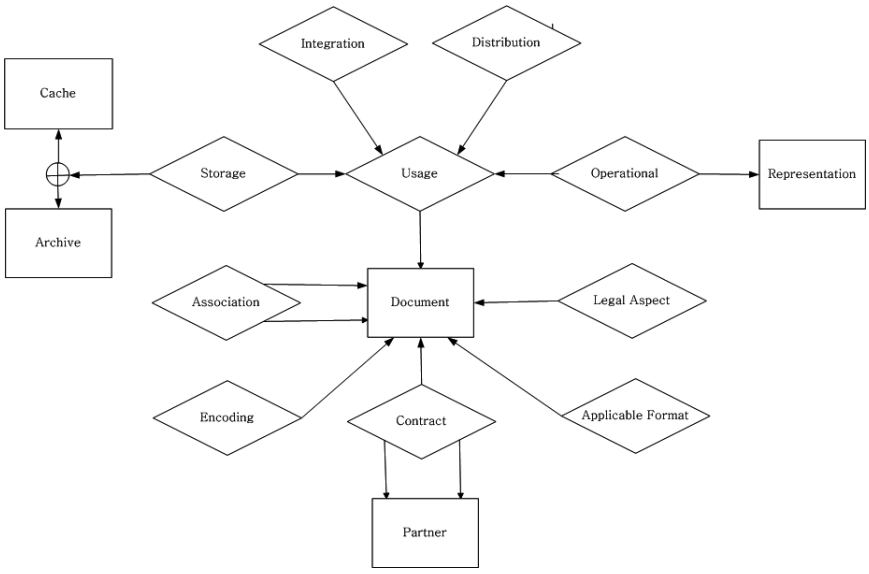

Figure 16: Loop or Circulation Meta-Structure.

3.2.4 Network Meta-Structures

Network or web meta-structures enable the collection of a network of associated types, and the creation of a multi-point 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.

Network meta-structures offer a unique opportunity to overcome the exploding type number problems in many applications, where relationships among objects are flexible, constantly changing, and reflect partial views or variants of other relationships. System configuration and configuration database management applications may however use the separation of types into categorised associations.

Network meta-structures are used for modern web services. Online resources for learning communities, special interest groups, and other shared information sources use a large number of associations among objects.

For instance, financial services are based on portfolios, combined on the fly, and provided, supported and used by institutions. Instead of representing the complex web of portfolio management we may split portfolios into basic portfolios, which relate portfolio providers and users. Such basic portfolios are combined and provided to customers or partner institutions. Whether a combination is considered to be a service depends on the customer's point of view.

As another example, railway management systems may be tightly bind to the application. The terminology varies from country to country. For instance, the notions of tunnel, path, segment, track, train or movement is different for most railway companies in Europe. Since trains also run between different regions, their scheduling, logging and reporting must combine all different systems. Moreover, identities for tracks and trains are derived from local databases and are not integrated resulting in a variety of schemata based on geographical separation. Thus, the development of a network meta-structure schema must be based on a common understanding of basic units and their disparate utilisation in the applications.

Another typical network meta-structure application is the support of legal documents as illustrated in Figure 17. They constitute a network of constantly renewed and deconstructed links among objects. In addition, local variations and specific portfolio for treatment and support are derived. Classical modelling approaches typically lead to schemata with document classes that either use chaotic sets of integrity constraints or use a confusing set of relationship types among the types in a schema. Figure 17 also illustrates the transformation of network meta-structures to abstract multi-layer structures, in which documents are interwoven with a large variety of links. This variety reflects the hierarchical structuring, usage and the evolution during the document lifespan.

As networks evolve quickly and irregularly, i.e. they grow fast and then are rebuilt and renewed, a network meta-structure must take care of a large number of variations to enable growth control and change management. Usually, they are supported by a multi-point center of connections, controlled routing and replication, change protocols, controlled assignment and transfer, scoping and localisation abstraction, and trader architectures. Furthermore, export/import converters and wrappers are supported. The database farm architecture [20] with check-in and check-out facilities supports flexible network extension.

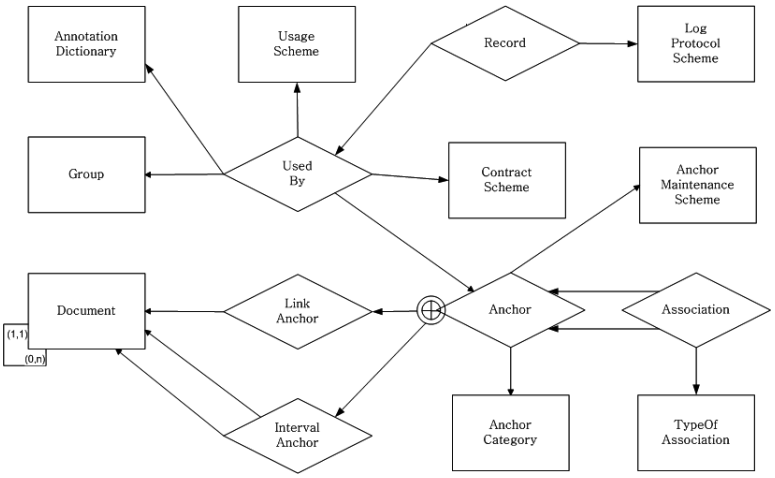

Figure 17: Network Meta-Structure.

3.3 Context Meta-Structures

According to [23] we distinguish between the *intext* and the *context* of things that are represented as objects. Intext reflects the internal structuring, associations among types and subschemata, the storage structuring, and the representation options. Context reflects general characterisations, categorisation, utilisation, and general descriptions such as quality. Therefore, we distinguish between *meta-characterisation meta-structures* that are usually orthogonal to the intext structuring and can be added to each of the intext types, *utilisation-recording meta-structures* that are used to trace the running, resetting and reasoning of the database engine, and *quality meta-structures* that permit to reason on the quality of the data provided and to apply summarisation and aggregation functions in a form that is consistent with the quality of the data. The dimensionality of a schema permits the extraction of other context meta-structures [3].

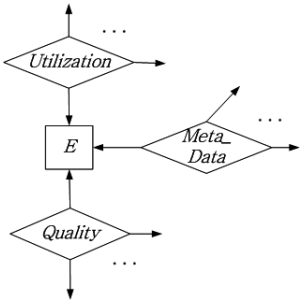

Figure 18: Context Meta-Structures.

Context meta-structures arise from joining in additional schemata – or using nesting – that capture meta information, e.g. for a document how it is used, by whom for which purpose, etc. (utilization meta-data), how accurate, complete or consistent it is (quality meta-data), or any other meta-data including technical and formating restrictions. This is illustrated in Figure 18. The three different classes of context meta-structures refer to a classification of context information.

3.3.1 Meta-Characterisation Meta-Structures

Meta-characterisation is orthogonal to the structuring dimension that may have led to a schema as displayed in Figure 6. They may refer to insertion/update/deletion time, keyword characterisation, utilisation pattern, format descriptions, utilisation restrictions and rights such as copyright and costs, and technical restrictions.

Meta-characterisations apply to a large number of types and should therefore be factored out. For instance, in an e-learning application learning objects, elements and scenes are commonly characterised by educational information such as interactivity type, learning resource type, interactivity level, age restrictions, semantic density, intended end user role, context, difficulty, utilisation interval restrictions, and pedagogical and didactical parameters.

3.3.2 Utilisation-Recording Meta-Structures

Logging, usage and history information is commonly used for recording the lifespan of the database. Therefore, we can distinguish between *history meta-structures* that are used for storing and recording the computation history within a small time slice, *usage-scene meta-structures* that are used to associate data to their use in a business process at a certain stage, a workflow step, or a scene in an application story, and record the actual usage.

Such meta-structures are related to one or more aspects of time, e.g. transaction time, user-defined time, validity time, or availability time, and associated with concepts such as temporal data types (instants, intervals, periods), and temporal statements such as current (now), sequenced (at each instant of time) and nonsequenced (ignoring time).

3.3.3 Quality Meta-Structures

Data quality is modelled by a variety of meta-structures capturing the sources (data source, responsible user, business process, source restrictions, etc.), intrinsic quality parameters (accuracy, objectivity, trustability, reputation, etc.), accessibility and security, contextual quality (relevance, value, timelineness, completeness, amount of information, etc.), and representation quality (ambiguity, ease of understanding, concise representation, consistent representation, ease of manipulation). Data quality is essential whenever versions of data have to be distinguished according to their quality and reliability.

4 Component-Based Schema Design

In this section we want to show how meta-structures and the associated schema algebra can be exploited to support component-based engineering as proposed in [12, 20]. We briefly review the rationale behind component-based development leading to the guiding principle of skeleton schemata that combine several components. We then extend the amalgamation approach from [12] in the light of the meta-structures discussed in the previous section and the new constructs introduced in this paper. In particular, we will emphasise that amalgamation and thus schema design can be based on graph rewriting.

4.1 Rationale for Component-Driven Development

Large database schemata can be drastically simplified, if techniques of modular design such as *design by units* [18] are used. Modular design is an abstraction technique based on principles of hiding and encapsulation that are known from Software Engineering. Different subschemata are connected by bridge types. *Component engineering* [12] extends this approach by means of view-centered components with well-defined composition operators, exploiting the observation that large subschemata often have the structure of star- or snowflake-schemata known from data warehousing. *Hierarchy abstraction* [20] permits to model objects on various levels of detail.

The co-design approach to database applications [18] aims at a consistent development of all facets of database applications: structuring of the database by schema types that are controlled by static integrity constraints, behaviour modelling by specification of functionality and dynamic integrity constraints, and interactivity modelling by assigning views to activities of actors in corresponding dialogue steps. Thus, co-design integrates the specification of the static database schema, functions, views and dialogues, which is facilitated by the use of view-extended schemata. At the same time, various abstraction layers are separated such as the conceptual layer, requirements acquisition layer and implementation layer, which has now become popular under the "model-driven architecture" theme.

Understandably, co-design is a rather complex procedure. However, if combined with the component-based approach it becomes simpler. In doing so first a skeleton of components is developed. This skeleton is then subject to stepwise refinement during further development of the view-extended schema. In particular, each component is refined thereby taking care of component interaction. In summary, co-design can be based on two principles:

Use of components: Components are the main building blocks for structuring the core data. In order to capture functionality components are modelled by view-extended schemata, in which each view contains also dialogue operations [12].

Skeleton-based construction: Components are assembled and amalgamated by applying connector types, which are usually relationship types.

4.2 Dimensions of Skeletons and Subschemata

A *component* – formally defined in [12, 20] – is a database schema together with import and export interfaces for connecting it to other components by standardised interface techniques. *Schema skeletons* [19] provide a framework for the general architecture of an application, to which details such as types are to be added. They are composed of *units*, which are defined by sets of components provided this set can be semantically separated from all other components without losing application information. Units may contain entity, relationship and cluster types, and the types in it should have a certain affinity or adhesion to each other.

In addition, units may be associated with each other in a variety of ways reflecting 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 with each other. Therefore, structuring mechanisms as provided by the higher-order entity-relationship model [18] may be used to describe skeletons.

The usage of types in a database schema differs in many aspects. In order to support the maintenance of very large schemata this diversity of usage should be made explicit. Following an analysis of usage patterns [12] leads to a number of dimensions including the following important ones:

- Types may be specialized on the basis of roles objects play or categories into which objects are separated. This *specialization dimension* usually leads to subtype, role, and categorisation hierarchies, and to versions for development, representation or measures.

- As objects in the application domain hardly ever occur in isolation, we are interested in representing their associations by bridging related types, and adding meta-characterisation on data quality. This *association dimension* often addresses specific facets of an appli-

cation such as points of view, application areas, and workflows that can be separated from each other.

- Data may be integrated into complex objects at runtime, and links to business steps and rules as well as log, history and usage information may be stored. Furthermore, meta-properties may be associated with objects such as category, source and quality information. This defines the *usage, meta-characterisation* or *log dimension*. Dockets [13] may be used for tracking processing information, superimposed schemata for explicit log of the treatment of the objects, and provenance schemata for the injection of meta-schemata.
- As data usage is often restricted to some user roles, there is a *rights and obligations dimension*, which entails that the characterisation of user activities is often enfolded into the schema.
- As data varies over time and different facets are needed at different moments, there is a *data quality, lifespan and history dimension* for modelling data history and quality , e.g. source data, and data referring to the business process, source restrictions, quality parameters etc. With respect to time the dimension distinguishes between transaction time, user-defined time, validity time, and availability time.
- The *meta-data dimension* refers to temporal, spatial, ownership, representation or context data that is often associated with core data. These meta-data are typically added after the core data has been obtained.

We often observe that very large database schemata incorporate some or all of these dimensions, which explains the difficulty for reading and comprehension. For instance, various architectures such as technical and application architecture may co-appear within a schema [15].

Furthermore, during its lifetime a database schema, which may originally have captured just the normalised structure of the application domain, is subjected to performance considerations and extended in various ways by views. A typical example for a complete schema full of derived data is given by OLAP applications [5]. Thus, at each stage the full schema is in fact the result of folding extensions by means of a so-called *grounding schema* into the core database schema.

4.3 Graph-Grammar Composition for Schemata

As emphasised in [9], a structural approach to schema construction as in [1] is possible. All constructors known for database schemata may also be applied to meta-structures. Therefore, we can base a theory of schema composition on constructors for generalised Entity-Relationship schemata as in [18].

A general composition theory for such schemata can be based on the theory of graph grammars [2, 16], which has been already exploited for the CASE tool RADD [18]. The composition of graphs can be formalised by two pushouts in the category of directed graphs. However, we will avoid using category-theoretical terminology. Furthermore, instead of general graph homomorphisms we only consider subgraphs.

A *graph production rule* takes the form

$$\varrho\colon\ L\ \supseteq\ K\ \subseteq\ R$$

with graphs L, R called the left-hand side and the right-hand side of the production rule ϱ, respectively, and a common subgraph K with $L \cap R = K$, which is called the *gluing graph* of ϱ.

The intuitively clear meaning of a graph production rule is to replace the left-hand side L by the right-hand side R, whenever L appears as a subgraph of any graph G. Naturally, as the *gluing graph* K of the rule is the intersection of the left- and right-hand sides, it will be invariant under the replacement.

However, the context of L within the graph G has to be taken into account as well, i.e. it has to be specified how edges connecting vertices in $G - L$ to vertices in L are handled. This leads to the exact definition of a rule application. Such an application of a graph production rule must be conflict-free in the sense that no name clashes occur between the graphs R and $G - L$. In order to avoid name clashes, vertices and edges in $R - L$ are to be renamed.

Let $\varrho : L \supset K \subset R$ be a graph production rule, and let G be a graph. Furthermore, in order to apply ϱ to G, we assume to be given

- a *renaming function* m defined on $L \cup R$ such that $m(L)$ becomes a subgraph of G and $m(L \cup R) \cap G = m(L)$ holds, and
- a subgraph C of G called *context graph* with $C \cap m(L) = m(K)$ and $G = C \cup m(L)$.

The graph H resulting from *applying the graph production rule ϱ to G* is defined by

$$H = (G - m(L)) \cup m(R).$$

We denote the graph transformation defined by ϱ and m – C is defined implicitly – by $G \overset{\varrho,m}{\Rightarrow} H$.

In graph rewriting the used set of graph production rules must satisfy the substitution rule, i.e. none of the transformations may have side effects. If vertices and edges outside $L - K$ are not affected, graph production rules can be composed to form derived graph production rules.

If $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are schemata and $\mathcal{O}$ is an n-ary operator applicable to them, the resulting schema $\mathcal{S}$ defines the equation $\mathcal{S} = \mathcal{O}(\mathcal{S}_1, \ldots, \mathcal{S}_n)$. Using these equations in a directed way defines a graph-rewriting system GRS. In view of the previous section, the graph production rules in GRS are rather simple, but more complex and presumably more convenient rules can be derived by rule composition.

Table 1: Rewrite rules for component amalgamation

join:	$\varrho_{\bowtie_{:=}} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \mathcal{S}_1 \cap \mathcal{S}_2 \subseteq \mathcal{S}_1 \bowtie_{\mathcal{I}_1:=\mathcal{O}_2 \| \mathcal{I}_2:=\mathcal{O}_1} \mathcal{S}_2$
sum:	$\varrho_{\oplus} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \emptyset \subseteq \mathcal{S}_1 \oplus \mathcal{S}_2$
reference-join:	$\varrho_{\bowtie_{\rightarrow}} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \mathcal{S}_1 \cup \mathcal{S}_2 \subseteq \mathcal{S}_1 \bowtie_{\mathcal{I}_1 \rightarrow \mathcal{O}_2 \| \mathcal{I}_2 \rightarrow \mathcal{O}_1} \mathcal{S}_2$
product:	$\varrho_{\times} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \emptyset \subseteq \mathcal{S}_1 \times \mathcal{S}_2$
meet:	$\varrho_{\bullet_\varphi} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \emptyset \subseteq \mathcal{S}_1 \bullet_\varphi \mathcal{S}_2$
nesting:	$\varrho_{Nest} : \mathcal{S}_1 \cup \mathcal{S}_2 \supseteq \mathcal{S}_1 - \{C\} \cup \mathcal{S}_2 \subseteq Nest_{C:\mathcal{S}_2(T)}(\mathcal{S}_1)$

4.4 Rewriting-Based Component Amalgamation

According to the decomposition theorem in [12] each behaviour-extended schema is the amalgamation of snowflake components. This naturally extends to all twelve types of meta-structures identified in [9]. Formally, this means that component sub-schemata can be written as algebraic expressions involving

- the renaming operator $\varrho_{R-1 \mapsto R'_1,\ldots,R_k \mapsto R'_k}$,
- the join operator $\bowtie_{\mathcal{I}_1:=\mathcal{O}_2 \| \mathcal{I}_2:=\mathcal{O}_1}$, and as a special case the direct sum operator $\oplus$,
- the reference-join operator $\bowtie_{\mathcal{I}_1 \rightarrow \mathcal{O}_2 \| \mathcal{I}_2 \rightarrow \mathcal{O}_1}$,
- the product operator $\times$ and more generally, the meet operator $\bullet_\varphi$,
- the bulk operator $Bulk_{R_1,\ldots,R_n}$ and its inverse expand operator $Expand_{E:A}$,
- the nesting operator $Nest_{C:\mathcal{S}'(T)}$, and
- the collection operators $\{\cdot\}$, $[\cdot]$, $\langle\cdot\rangle$, and $\langle\!\langle\cdot\rangle\!\rangle$, and the related operators all_of, $any\_of\ n\_of$, and n_th.

This defines the formal underpinnings for the following pragmatic steps in view-extended schema design:

1. We start from behaviour-extended schemata for certain tasks of the application, as they may arise from cutting up a development project and then working independently. These schemata may not be snowflake components. However, they can be represented as amalgams. Then the definition of views that connect these components defines an amalgam for the whole application.
2. Each component resulting from step one can be decomposed into snowflake components by the decomposition theorem. So we need algorithms for detecting components and checking, whether they are almost hierarchical or not. Together with phase one this amounts to an amalgam with a larger number of components, but these components are now snowflakes.
3. In the third phase we consider the overlap between components aiming at minimising them as much as possible. The result will still be an amalgam with snowflake components, but these components do not overlap excessively any more.
4. Finally, we reconsider the components resulting from phase three and recombine some of them, if this the result is still a snowflake component and the application considers the initial components as belonging together to one task of the application.

Naturally, amalgamation itself will exploit the algebra operators above. Thus the pragmatic approach can be supported by formal graph rewriting. If two component schemata $\mathcal{S}_1$ and $\mathcal{S}_2$ are given, we may define the amalgam by exploiting one of the rewrite rules in Table 1.

When applying these rules, suitable views $\mathcal{I}_j$ and $\mathcal{O}_j$, and types C and T have to be selected.

5 Conclusions

In this article we addressed the unsatisfactory situation that the design of very large database schemata is not well supported. Such schemata with hundreds or thousands of types are usually developed over years, and then require sophisticated skills to read and comprehend them. However, lots of similarities, repetitions, and similar structuring elements appear in such schemata. In this paper we highlighted the frequently occurring meta-structures in such schemata, and classified them according to structure, lifespan and context. Furthermore, we presented an algebra for handling these meta-structures, which permits large schemata to be composed out of smaller ones. In this way a component-based approach to schema design is enabled, in which the application of the schema algebra constructors can be formalised by graph rewriting.

Practically speaking, meta-structures can be exploited to modularise schemata, which would ease querying, searching, reconfiguration, maintenance, integration and extension. From a development perspective different aspects dealing with structures, lifespan and context could be separated. Thus, an easier integration of development subpro-

jects would be possible. Also reengineering and reuse are enabled.

In this way data modelling using meta-structures enables systematic schema development, extension and implementation, and thus contributes to overcome the maintenance problems arising in practice from very large schemata. Furthermore, the use of meta-structures also enables component-based schema development, in which schemata are developed step-by-step on the basis of the skeleton of the meta-structure, and thus contributes to the development of industrial-scale database applications.

However, in our presentation in this article we concentrated on schemata with constraints, thus ignoring additional aspects such as views and operations. In the component-model in [12] these were also considered as part of component-based information systems engineering. Consequently, our approach requires additional investigation of the interaction aspect. The question is, whether frequently occurring patterns can also be discovered for views and operations. For the classical application area of decision support this question has already been addressed and answered positively by means of standard OLAP operations [6].

References

[1] Brown, L. *Integration Models – Templates for Business Transformation*. SAMS Publishing, 2000.

[2] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., Eds. *Handbook of Graph Grammars and Computing by Graph Transformations – Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[3] Feyer, T., and Thalheim, B. Many-dimensional schema modeling. In *Advances in Databases and Information Systems – Proc. ADBIS 2002*, Y. Manolopoulos and P. Návrat, Eds., vol. 2435 of *LNCS*. Springer-Verlag, 2002, pp. 305–318.

[4] Kahlen, H. *Integrales Facility Management – Management des ganzheitlichen Bauens*. Werner Verlag, 1999.

[5] Lenz, H.-J., and Thalheim, B. OLAP schemata for correct applications. In *Trends in Enterprise Application Architecture*, vol. 3888 of *LNCS*. Springer-Verlag, 2005, pp. 99–113.

[6] Lenz, H.-J., and Thalheim, B. A formal framework of aggregation for the OLAP-OLTP model. *Journal of Universal Computer Science 15*, 1 (2009), 273–303.

[7] Ma, H., Noack, R., and Schewe, K.-D. Algebraic meta-structure handling of huge database schemata. In *Advances in Conceptual Modeling – Challenging Perspectives*, C. Heuser and G. Pernul, Eds., vol. 5833 of *LNCS*. Springer-Verlag, 2009, pp. 23–32.

[8] Ma, H., Noack, R., Schewe, K.-D., Thalheim, B., and Wang, Q. Complete conceptual schema algebras. submitted for publication, 2009.

[9] Ma, H., Schewe, K.-D., and Thalheim, B. Modelling and maintenance of very large database schemata using meta-structures. In *Information Systems and e-Business Technologies – 3rd International Conference UNISCON 2009, Proceedings*, J. Yang et al., Eds., vol. 20 of *LNBIP*. Springer-Verlag, 2009, pp. 17–28.

[10] Moody, D. *Dealing with Complexity: A Practical Method for Representing Large Entity-Relationship Models*. PhD thesis, University of Melbourne, 2001.

[11] Raak, T. Database systems architecture for facility management systems. Master's thesis, Fachhochschule Lausitz, 2002.

[12] Schewe, K.-D., and Thalheim, B. Component-driven engineering of database applications. In *Conceptual Modelling – Proc. APCCM 2006*, vol. 53 of *CRPIT*. Australian Computer Society, 2006, pp. 105–114.

[13] Schmidt, J. W., and Sehring, H.-W. Dockets: A model for adding value to content. In *Conceptual Modeling – ER '99*, vol. 1728 of *LNCS*. Springer-Verlag, 1999, pp. 248–262.

[14] Shoval, P., Danoch, R., and Balaban, M. Hierarchical ER diagrams (HERD) – the method and experimental evaluation. In *Advanced Conceptual Modeling Techniques*, vol. 2784 of *LNCS*. Springer-Verlag, 2002, pp. 264–274.

[15] Siedersleben, J. *Moderne Softwarearchitektur*. dpunkt-Verlag, 2004.

[16] Sleep, M. R., Plasmeijer, M. J., and van Eekelen, M. C. J. D., Eds. *Term Graph Rewriting – Theory and Practice*. John Wiley and Sons, 1993.

[17] Smith, J. M., and Smith, D. C. P. Database abstractions: Aggregation and generalization. *ACM ToDS 2*, 2 (1977), 105–133.

[18] Thalheim, B. *Entity Relationship Modeling – Foundations of Database Technology*. Springer-Verlag, 2000.

[19] Thalheim, B. Component construction of database schemes. In *Conceptual Modeling – ER 2002*, vol. 2503 of *LNCS*. Springer-Verlag, 2002, pp. 20–34.

[20] Thalheim, B. Component development and construction for database design. *Data and Knowledge Engineering 54* (2005), 77–95.

[21] Thalheim, B. Engineering database component ware. In *Trends in Enterprise Application Architecture*, vol. 4473 of *LNCS*. Springer-Verlag, 2007, pp. 1–15.

[22] Thalheim, B., and Kobienia, T. Generating database queries for web natural language requests using schema information and database content. In *Applications of Natural Language to Information Systems – NLDB 2001*, vol. 3 of *LNI*. GI, 2001, pp. 205–209.

[23] Wisse, P. *Metapattern – Context and Time in Information Models*. Addison-Wesley, 2001.