Modelling and Maintenance of Very Large Database Schemata Using Meta-structures

Hui Ma[1], Klaus-Dieter Schewe[2], and Bernhard Thalheim[3]

[1] Victoria University of Wellington, School of Engineering and Computer Science, Wellington, New Zealand
hui.ma@ecs.vuw.ac.nz
[2] Information Science Research Centre, Palmerston North, New Zealand
kdschewe@acm.org
[3] Christian-Albrechts-University Kiel, Institute of Computer Science, Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Abstract. Practical experience shows that the maintenance of databases with a very large schema causes severe problems, and no systematic support is provided. In this paper we address this problem. 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. We argue that systematic use of these meta-structures will ease the modelling and maintenance of very large database schemata.

Keywords: Database modelling, schema maintenance, meta-structure.

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 [7].

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* [13] 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* [9] extends this approach by means of view-centered components with well-defined composition operators, and *hierarchy abstraction* [16] 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 [17]. 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 paper we further develop the method for systematic schema development focussing on very large schemata. We first analyse skeletons and subschemata more deeply in Section 2 and identify distinguishing dimensions [3]. 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, which determine the skeleton schema. These meta-structures are classified into three categories addressing schema construction, lifespan and context. Finally, in Section 4 we elaborate more on the application of meta-structures in data modelling, but due to space restrictions some formal details have to be outsourced. In a concurrent submission [6] we elaborate on the handling of the identified meta-structures in a more formal way.

2 Internal Dimensions of Skeletons and Subschemata

A *component* – formally defined in [9,16] – is a database schema together with import and export interfaces for connecting it to other components by standardised interface techniques. *Schema skeletons* [15] 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 [13] 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 [9] leads to a number of internal dimensions including the following imortant 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 application 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 [10] 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. Wity 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 [11].

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.

3 Meta-structures in Subschemata and Schema Skeletons

Based on an extensive study of a large number of conceptual database schemata 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. Due to space restrictions the description will only contain sufficient details for some of the meta-structures, whereas for the others it will necessarily be rather terse.

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 [9] 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.

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 by a core entity (or relationship) type used for storing basic data, and a number of subtypes of the entity type that are used to capture additional properties [16]. A typical star structure is shown in Figure 1 for the Address entity type. In the same fashion a *snowflake schema*, the one in Figure 2 – shown without attributes – represents the information structure of documented contributions of members of working groups during certain time periods.

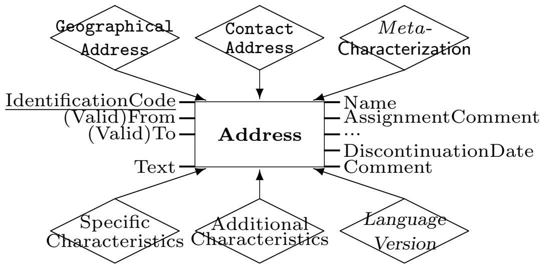

Fig. 1. The General Structure of *Addresses*

Bulk Meta-Structures. Types used in schemata in a very similar way can be clustered together on the basis of a classification.

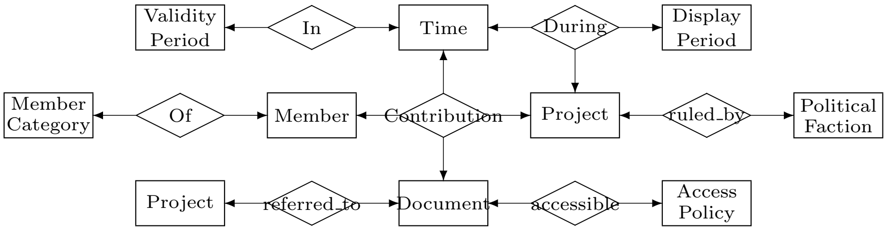

Fig. 2. Snowflake Schema on Contributions

Let us exemplify this generalisation approach for the commenting process in an e-community application. The relationship types `Made`, `Commented`, and `Reused` in Figure 3 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 4. 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.

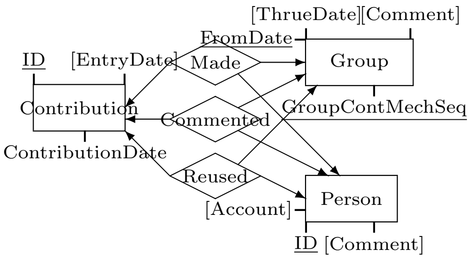

Fig. 3. E-Community Application

In general, the meta-structure can be described as follows:

Assume to be given a *central type* C and other types that are associated with C by a set of relationship types $\{A_1, ..., A_n\}$ by means of an *occurrence frame* F. The occurrence frame can be such that either all inclusion constraints $A_i[C] \subseteq A_j[C]$ for $1 \leq i, j \leq n$ must hold or another set of inclusion constraints.

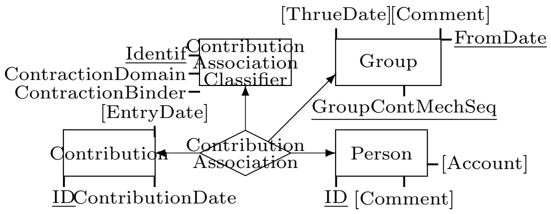

Fig. 4. Bulk Meta-Structure for E-Community

Now we combine the types $\{A_1, ..., A_n\}$ into a type `BulkType` with an additional component `ContractionAssistant`, and attributes `Identif` to identify objects of this type, `ContractionDomain` with domain $\{A_1, \ldots, A_n\}$, and `ContractionBinder` with domain F. This is shown in Figure 5.

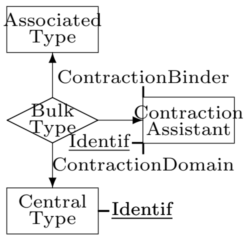

Fig. 5. General Bulk Meta-Structure

Architecture and Constructor-Based Meta-Structures. Categorisation and compartmentalization 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. Therefore, we adopt the term *waffle meta-structure* or *architecture meta-structure* for structures that arise this way. These meta-structures are especially usefulfor 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.

Star and snowflake schemata may be composed by composition operators such as *product*, *nest*, *disjoint union*, *difference* and *powerset*. These operators permit the construction of any schema of interest, as they are complete for sets. A structural approach as in [1] can be employed. Thus, all constructors known for database schemata may also be applied to meta-schema construction.

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 meta-structures* display the phases in the lifespan of objectss, *incremental meta-strucutres* permit the recording of the development, enhancement and ageing of objects, *loop meta-structures* support chaining and scaling to different perspectives 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.

Evolution Meta-Structures. 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.

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.

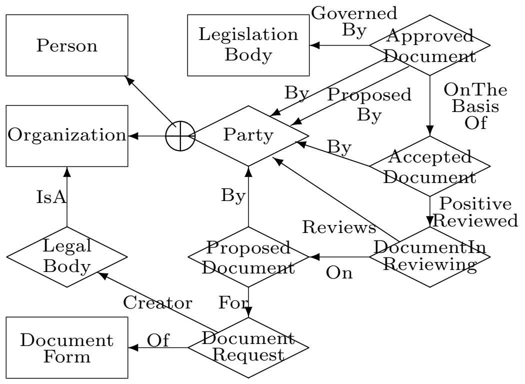

Fig. 6. Incremental Meta-Structure

Circulation meta-structures permit to display objects in different phases. For instance, legal document handling in the SeSAM system, an e-government application, is based on such phases: `DocumentForm`, `ProposedDocument`, `DocumentInReviewing`, `AcceptedDocument`, `RejectedDocument`, `FinalVersionDocument`, `ApprovedDocument`, and `ArchievedDocument`. The circulation model is supported by a phase-based dynamic semantics [13]. Alternatively, an incremental meta-structure could be chosen as shown in Figure 6.

Incremental Meta-Structures. Incremental meta-structures enable the production of new associations based on a core object. It employs 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.

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 consider the schema displayed in Figure 6 dealing with the handling of legal documents in the e-gvernance 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`.

Loop Meta-Structures. Loop 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 since 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.

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.

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 [16] with check-in and check-out facilities supports flexible network extension.

3.3 Context Meta-structures

According to [18] 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].

Meta-Characterisation Meta-Structures. Meta-characterisation is orthogonal to the structuring dimension that may have led to a schema as displayed in Figure 1. 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.

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).

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 Application of Meta-structures in Data Modelling

Let us now briefly illustrate meta-structuring for some application examples.

4.1 Design by Units

The design-by-units framework [13] provides a modular design technique exploiting the internal skeleton structure of very large schemata. For illustration let us consider the example of the Cottbus*net* website involving four main components:

- The star subschema characterising people maintains the data for types of people of interest: `Member_Of_Group`, `Representative_Of_Partner`, and `User`.
- The snowflake subschema on project information is used for the representation of information on various projects, their different stages, their results and their representation.
- The snowflake subschema on group information allows to store data on groups, their issues, leadership, obligations and results.
- The snowflake subschema on website maintenance provides data on the information that must be given through the web interface to authorized, anonymous and general users.

The skeleton of the application schema combines these components. The internal structure of the components is either a star or a snowflake subschema. The skeleton is associates the components using connector types, e.g. `Contribution` in Figure 2 between *Person* or *Member*, *Document*, *Project* and *Time*, `PortfolioProfile` between *Person*, *Group*, and *Website*, etc. In a similar way we can extent the skeleton by a component describing the organisation of group work.

4.2 Incremental Structuring

Meta-structuring supports the incremental evolution of database systems as a specific form of database system evolution, in particular for facility management systems. Such systems use a number of phases such as *planning phase*, *construction phase*, *realization phase*, and *maintenance phase*. Based on meta-structures we developed the novel architecture shown in Figure 7, which has already been positively evaluated in a project [8], in which auxiliary databases provide help information, and information on regulations, customers, suppliers, etc.

Thus, incremental evolution is supported by meta-structuring on the basis of import forms of two kinds:

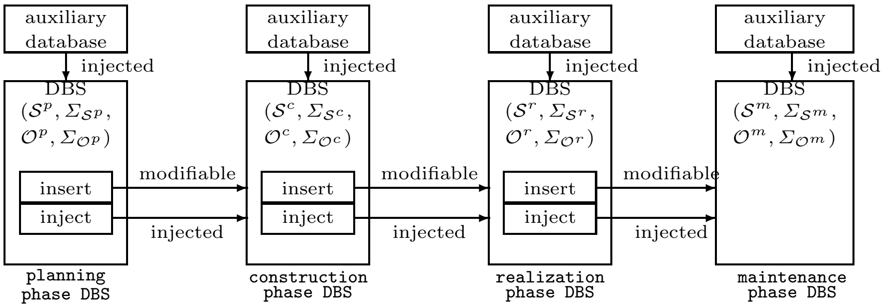
auxiliary database
injected
DBS
$(\mathcal{S}^p, \Sigma_{\mathcal{S}^p}, \mathcal{O}^p, \Sigma_{\mathcal{O}^p})$
insert
inject
modifiable
injected
planning phase DBS
auxiliary database
injected
DBS
$(\mathcal{S}^c, \Sigma_{\mathcal{S}^c}, \mathcal{O}^c, \Sigma_{\mathcal{O}^c})$
insert
inject
modifiable
injected
construction phase DBS
auxiliary database
injected
DBS
$(\mathcal{S}^r, \Sigma_{\mathcal{S}^r}, \mathcal{O}^r, \Sigma_{\mathcal{O}^r})$
insert
inject
modifiable
injected
realization phase DBS
auxiliary database
injected
DBS
$(\mathcal{S}^m, \Sigma_{\mathcal{S}^m}, \mathcal{O}^m, \Sigma_{\mathcal{O}^m})$
maintenance phase DBS

Fig. 7. The General Architecture of Incremental Evolution of Database Systems

- *Injection forms* enable the injection of data into another database. The forms are supported by cooperating views. Data injected into another database cannot be changed by the importing database system. The structure $(\mathcal{S}^{inject}, \Sigma_{\mathcal{S}})$ of the views of the exporting database system is entirely embedded into the structure $(\mathcal{S}', \Sigma_{\mathcal{S}'})$ of the importing database system. The functionality $(\mathcal{O}^{inject}, \Sigma_{\mathcal{O}})$ of the views of the exporting database system is partially embedded into the functionality $(\mathcal{O}', \Sigma_{\mathcal{O}'})$ of the importing database system by removing all modification operations on the injected data. These data can only be used for retrieval purposes.
- *Insertion forms* enable the insertion of data from an exporting database into an importing database. These data can be modified. The structure $(\mathcal{S}^{insert}, \Sigma_{\mathcal{S}})$ and the functionality $(\mathcal{O}^{insert}, \Sigma_{\mathcal{O}})$ of the views of the exporting database system are entirely embedded into the structure $(\mathcal{S}', \Sigma_{\mathcal{S}'})$ and the functionality $(\mathcal{O}', \Sigma_{\mathcal{O}'})$ of the importing database system.

4.3 The String Bag Modelling Approach

Looking from the distance at an ER schema the observer may find the diagram similar to a string bag, and indeed, there is a large number of similarities. Using the metaphor of a *string-bag* it has been observed that most database queries access types that are connected by a subbag [14], and this can be exploited to automatically derive corresponding SQL queries. For this some of the types in a query serve as critical types, while the others are used to select the appropriate paths in the ER schema.

This observation on query behavior can be generalized to database modelling declaring some of the types to be major or core types, while the others serve for specialising the application area. This forms the basis of an abstraction principle according to which the main view schemata form the abstraction of the schema, and the application is specified by the main views of the "handles", i.e. the core types.

If we consider an application that addresses the types `Contact_Address, Party_Address, Geographical_Address`, then associations are view types on the schema relating the main type `Address` to the kind of usage, i.e. to the main use of addresses in the application.

4.4 Rigid Structuring and Principles of Schema Abstraction

Database design techniques have often aimed at finding the best possible integration on the basis of the assumption of uniformity. As already observed database schemata should be based on a balance of three principles:

Autonomy: Objects represent things that are used in a separated and potentially independent form.

Hierarchy: The main classification method is the development of hierarchies. Hierarchies may be based on generalisation/specialisation, ontologies, or concept maps.

Coordination: Objects are related to each other and are used to exchange information among objects. The cohesion of objects is supported by coordination.

These principles can be extended to principles of schema abstraction:

Extraction of real differences: Recognition of differences enable the handling of classes that are to be treated differently. It strongly promotes eventual cohesion. Differences permit building the skeleton of the application.

Cultivation of hierarchies: Things in real applications can be associated to each other by specialisation and generalisation. Modelling uses hierarchies for factoring out facets and to use them for simpler and more efficient treatment.

Contraction by similarities: Unnecessary differenciation should be avoided. Therefore, similar subschemata and types should be clustered and modelled as components.

5 Conclusion

Very large database 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. We demonstrated that meta-structures can be exploited to modularise schemata, which would ease querying, searching, reconfiguration, maintenance, integration and extension. 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. We plan to elaborate further on formal aspects of meta-structing in data modelling with the concurrent submission in [6] being a first step. This will exploit graph grammars [2] and graph rewriting [12].

References

1. Brown, L.: Integration Models – Templates for Business Transformation. SAMS Publishing (2000)
2. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformations. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
3. Feyer, T., Thalheim, B.: Many-dimensional schema modeling. In: Manolopoulos, Y., Návrat, P. (eds.) ADBIS 2002. LNCS, vol. 2435, pp. 305–318. Springer, Heidelberg (2002)
4. Kahlen, H.: Integrales Facility Management – Management des ganzheitlichen Bauens. Werner Verlag (1999)
5. Lenz, H.-J., Thalheim, B.: OLAP schemata for correct applications. In: Draheim, D., Weber, G. (eds.) TEAA 2005. LNCS, vol. 3888, pp. 99–113. Springer, Heidelberg (2006)
6. Ma, H., Schewe, K.-D., Thalheim, B.: Handling meta-structures in data modelling (submitted, 2008)
7. Moody, D.: Dealing with Complexity: A Practical Method for Representing Large Entity-Relationship Models. Ph.D thesis, University of Melbourne (2001)
8. Raak, T.: Database systems architecture for facility management systems. Master's thesis, Fachhochschule Lausitz (2002)
9. Schewe, K.-D., Thalheim, B.: Component-driven engineering of database applications. In: Conceptual Modelling – Proc. APCCM 2006. CRPIT, vol. 53, pp. 105–114. Australian Computer Society (2006)
10. Schmidt, J.W., Sehring, H.-W.: Dockets: A model for adding value to content. In: Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E. (eds.) ER 1999. LNCS, vol. 1728, pp. 248–263. Springer, Heidelberg (1999)
11. Siedersleben, J.: Moderne Softwarearchitektur. dpunkt-Verlag (2004)
12. Sleep, M.R., Plasmeijer, M.J., van Eekelen, M.C.J.D. (eds.): Term Graph Rewriting – Theory and Practice. John Wiley and Sons, Chichester (1993)
13. Thalheim, B.: Entity Relationship Modeling – Foundations of Database Technology. Springer, Heidelberg (2000)
14. Thalheim, B.: Generating database queries for web naturallanguage requests using schema information and database content. In: Applications of Natural Language to Information Systems – NLDB 2001. LNI, vol. 3, pp. 205–209. GI (2001)
15. Thalheim, B.: Component construction of database schemes. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 20–34. Springer, Heidelberg (2002)
16. Thalheim, B.: Component development and construction for database design. Data and Knowledge Engineering 54, 77–95 (2005)
17. Thalheim, B.: Engineering database component ware. In: Draheim, D., Weber, G. (eds.) TEAA 2006. LNCS, vol. 4473, pp. 1–15. Springer, Heidelberg (2007)
18. Wisse, P.: Metapattern – Context and Time in Information Models. Addison-Wesley, Reading (2001)