Engineering Database Component Ware

Bernhard Thalheim

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

Abstract. Large database applications often have a very complex structuring that complicate maintenance, extension, querying, programming. Due to this complexity systems become unmaintenable. We observe, however, that large database applications often use an implicit structuring into connected components. We propose to initially use this internal structuring for application development. The application architecture is based on database components. Database components can be composed to an application system. This paper shows how components may be developed, composed and applied.

1 Towards Information Systems Engineering

Component-Based Application Engineering
Software engineering is still based on *programming in the small* although a number of approaches has been proposed for *programming in the large*. Programming in the large uses strategies for programming, is based on architectures, and constructs software from components which collaborate, are embedded into each other, or are integrated for formation of new systems. Programming constructs are then pattern or high-level programming units and languages.

The next generation of programming observed nowadays is *programming in the world* within a collaboration of programmers and systems. It uses advanced scripting languages such as Groovy with dynamic integration of components into other components, standardisation of components with guarantees of service qualities, collaboration of components with communication, coordination and cooperation features, distribution of workload, and virtual communities. Therefore, component engineering will also form the kernel engineering technique for programming in the world. The next generation of software engineering envisioned is currently called as *programming by composition* or construction. In this case components also form the kernel technology for software and hardware.

Software development is mainly based on stepwise development from scratch. Software reuse has been considered but never reached the maturity for application engineering. Database development is also mainly *development in the small*. Schemes are developed step by step, extended type by type, and normalized locally type by type. Views are still defined type by type although more complex schemata can be easily defined by extended ER schemata [Tha00].

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

D. Draheim and G. Weber (Eds.): TEAA 2006, LNCS 4473, pp. 1–15, 2007.

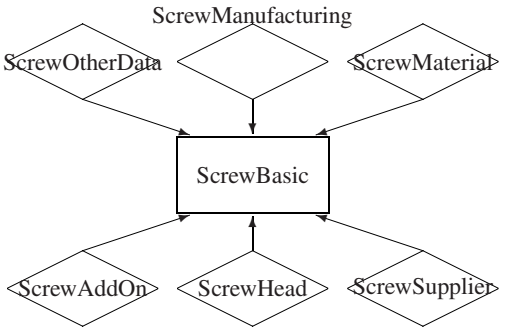


Fig. 1. HERM Representation of the Star Type *Screw*

Engineering applications have been based on the simple *separation principle*: *Separation of elements which are stable from those elements which are not.* This separation allows *standardization* and *simple integration*. An example is the specification of screws as displayed in Figure 1[1]. Screws have a standardized representation: basic data, data on the material, data on the manufacturing, data on specific properties such as head, etc.

Complex Applications Result in Large Schemata

Monographs and database course books usually base explanations on small or 'toy' examples. Reality is, however, completely different. Database schemata tend to be large, not surveyable, incomprehensible and partially inconsistent due to application, the database development life cycle and due to the number of team members involved at different time intervals. Thus, consistent management of the database schema might become a nightmare and may lead to legacy problems. The size of the schemata may be very large.

It is a common observation that large database schemata are error-prone, are difficult to maintain and to extend and are not surveyable. Moreover, development of retrieval and operation facilities requires highest professional skills in abstraction, memorization and programming. Such schemata reach sizes of more than 1000 attribute, entity and relationship types. Since they are not comprehensible any change to the schema is performed by extending the schema and thus making it even more complex. Database designers and programmers are not able to capture the schema.

Application schemata could be simpler only to a certain extent if software engineering approaches are applied. The repetition and redundancy in schemata is also caused by

- different usage of similar types of the schema,
- minor and small differences of the types structure in application views, and
- semantic differences of variants of types.

Therefore, we need approaches which allow to reason on repeating structures inside schemata, on semantic differences and differences in usage of objects.

[1] We use the extended ER model [Tha00] that allows to display subtypes on the basis of unary relationship types and thus simplifies representation.

Large schemata also suffer from the *deficiency of variation detection: The same or similar content is often repeated in a schema without noticing it.*

Techniques to Decrease Complexity in Applications

Large database schemata can be drastically simplified if techniques of *modular modelling* such as *modular design by units* [Tha00] are used. It is an abstraction technique based on principles of hiding and encapsulation. Design by units allows to consider parts of the schema in a separate fashion. The parts are connected via types which function similar to bridges.

Data warehousing and user views are often based on *snowflake or star schemata*. The intuition behind such schemata is often hidden. Star and snowflake schemata are easier to understand, to query, to survey and to maintain. At the same time, these structures are of high redundancy and restricted modelling power. For instance, the central type in a star or snowflake schema is a relationship type which has attributes that use only numerical types. We may wonder, however, why we need to apply these restrictions and why we should not use this approach in general.

Co-design [Tha00] of database applications aims in consistent development of all facets of database applications: structuring of the database by schema types and static integrity constraints, behavior modelling by specification of functionality and dynamic integrity constraints and interactivity modelling by assigning views to activities of actors in the corresponding dialogue steps. Co-design, thus, is based on the specification of the the database schema, functions, views and dialogue steps. At the same time, various abstraction layers are separated such as the conceptual layer, requirements acquisition layer and implementation layer.

Software becomes surveyable, extensible and maintainable if a clear separation of concerns and application parts is applied. In this case, a skeleton of the application structure is developed. This skeleton separates parts or services. Parts are connected through interfaces. Based on this *architecture*, an application can be developed part by part.

We *combine* modularity, star structuring, co-design, and architecture development to a novel framework based on components. Such combination seems to be not feasible. We discover, however, that we may integrate all these approaches by using a component-based approach. This skeleton can be refined during evolution of the schema. Then, each component is developed step by step. Structuring in component-based co-design is based on two constructs:

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

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

Goals of the Paper

The paper surveys our approach [Tha02, Tha03a, Tha05] for *systematic development of large database schemata* and applies it for database construction based on components and for collaborating component suites. The paper is based on [Fey03, FT02, ST06a,

ST04]. We introduce first the concept of database components and then discuss engineering of database applications based on components.

2 Database Components and Construction of Schemes

Database Schemes in a Nutshell

We use the extended ER model for representation of structuring and behavior generalizing the approach of [PBGG89]. The extended ER model (HERM) [Tha00] has a generic algebra and logic, i.e., the algebra of derivable operations and the fragment of (hierarchical) predicate logic may be derived from the HERM algebra whenever the structure of the database is given.

A database type $\mathfrak{S} = (S, O, \Sigma)$ is given by

- a structure S defined by a type expression defined over the set of basic types B, a set of labels L and the constructors product (tuple), set and bag, i.e. an expression defined by the recursive type equality
  $t = B \,|\, t \times ... \times t \,|\, \{t\} \,|\, [t] \,|\, l : t \quad ,$
- a set of operations defined in the ER algebra and limited to S, and
- a set of (static and dynamic) integrity constraints defined in the hierarchical predicate logic with the base predicate P_S.

Objects of the database type $\mathfrak{S}^C$ are S-structured. Classes $\mathfrak{S}^C$ are sets of objects for which the set of static integrity constraints is valid.

Operations can be classified into "retrieval" operations enabling in generating values from the class $\mathfrak{S}^C$ and "modification" operations allowing to change the objects in the class $\mathfrak{S}^C$ if static and dynamic integrity constraints are not invalidated.

A database schema $\mathfrak{D} = (\mathfrak{S}_1, ..., \mathfrak{S}_m, \Sigma_G)$ is defined by

- a list of different database types and
- a set of global integrity constraints.

The HERM algebra can be used to define (parameterized) views $\mathfrak{V} = (V, O_V)$ on a schema $\mathfrak{D}$ via

- an (parameterized) algebraic expression V on $\mathfrak{D}$ and
- a set of (parameterized) operations of the HERM algebra applicable to V.

The view operations may be classified too into retrieval operations O_V^R and modification operations O_V^M. Based on this classification we derive an *output view* O^V of $\mathfrak{V}$ and an *input view* I^V of $\mathfrak{V}$.

In a similar way (but outside the scope of this paper) we may define transactions, interfaces, interactivity, recovery, etc.

Obviously, I^V and O^V are typed based on the type system. Data warehouse design is mainly view design [Tha00].

Database Components and Component Algebra

A database component is *database scheme that has an import and an export interface for connecting it to other components by standardized interface techniques*. Components are defined in a data warehouse setting. They consist of input elements, output

elements and have a database structuring. Components may be considered as input-output machines that are extended by the set of all states S^C of the database with a set of corresponding input views $I^{\mathcal{V}}$ and a set of corresponding output views $O^{\mathcal{V}}$. Input and output of components is based on channels K. The structuring is specified by S_K. The structuring of channels is described by the function $type : C \to \mathcal{V}$ for the view schemata $\mathcal{V}$. Views are used for collaboration of components with the environment via data exchange. In general, the input and output sets may be considered as abstract words from M^* or as words on the database structuring.

A database component $\mathcal{K} = (S_K, I_K^{\mathcal{V}}, O_K^{\mathcal{V}}, S_K^C, \Delta_K)$ is specified by

(static) schema S_K describing the database schema of $\mathcal{K}$,
syntactic interface providing names (structures, functions) with parameters and database structure for S_K^C and $I_K^{\mathcal{V}}, O_K^{\mathcal{V}}$,
behavior relating the $I^{\mathcal{V}}, O^{\mathcal{V}}$ (view) channels
$\Delta_K : (S_K^C \times (I_K^{\mathcal{V}} \to M^*)) \to \mathcal{P}(S_K^C \times (O_K^{\mathcal{V}} \to M^*))$.

Components can be associated to each other. The association is restricted to domain-compatible input or output schemata which are free of name conflicts.

Components $\mathcal{K}_1 = (S_1, I_1^{\mathcal{V}}, O_1^{\mathcal{V}}, S_1^C, \Delta_1)$ and $\mathcal{K}_2 = (S_2, I_2^{\mathcal{V}}, O_2^{\mathcal{V}}, S_2^C, \Delta_2)$ are free of name conflicts if the set of attribute, entity and relationship type names are disjoint.

Channels C_1 and C_2 of components $\mathcal{K}_1 = (S_1, I_1^{\mathcal{V}}, O_1^{\mathcal{V}}, S_1^C, \Delta_1)$ and $\mathcal{K}_2 = (S_2, I_2^{\mathcal{V}}, O_2^{\mathcal{V}}, S_2^C, \Delta_2)$ are called *domain-compatible* if $dom(type(C_1)) = dom(type(C_2))$.

An output O_1^V of the component $\mathcal{K}_1$ is *domain-compatible* with an input I_2^V of the component $\mathcal{K}_2$ if $dom(type(O_1^V)) \subseteq dom(type(I_2^V))$

Component operations such as merge, fork, transmission are definable via application of superposition operations [Kud82, Mal70]: Identification of channels, permutation of channels, renaming of channels, introduction of fictitious channels, and parallel composition with feedback displayed in Figure 2.

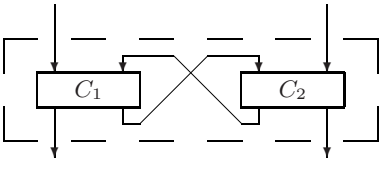


Fig. 2. The Composition of Database Components

Thus, a component schema is usually characterized by a kernel entity type used for storing basic data, by a number of dimensions that are usually based on subtypes of the entity type which are used for additional properties. These additional properties are clustered according to their occurrence for the things under consideration. Typically, the component schema uses four dimensions: subtypes, additional characterization, versions and meta-characterizations.

The star schema is the main component schema used for construction.

A star schema for a database type C_0 is defined by

- the (full) (HERM) schema $\mathcal{S} = (C_0, C_1, ..., C_n)$ covering all types on which C_0 has been defined,
- the subset of *strong types* $C_1, ..., C_k$ forming a set of keys $K_1, ..., K_s$ for C_0, i.e., $\cup_{i=1}^{s} K_i = \{C_1, ..., C_k\}$ and $K_i \rightarrow C_0\,,\ C_0 \rightarrow K_i$ for $1 \leq i \leq s$ and $card(C_0, C_i) = (1, n)$ for $(1 \leq i \leq k)$.
- the extension types $C_{k+1}, ..., C_m$ satisfying the (general) cardinality constraint $card(C_0, C_j) = (0, 1)$ for $((k + 1) \leq i \leq n)$.

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

There are various variants for representation of a star schemata:

- Representation based on an entity type with attributes $C_1, ..., C_k$ and $C_{k+1}, ..., C_l$ and specialisations forming a specialization tree $C_{l+1}, ..., C_n$.
- Representation based on a relationship type C_0 with components $C_1, ..., C_k$, with attributes $C_{k+1}, ..., C_l$ and specialisations forming a specialization tree $C_{l+1}, ..., C_n$. In this case, C_0 is a *pivot element* [BP00] in the schema.
- Representation by be based on a hybrid form combining the two above.

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

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

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

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

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

Lifespan variants of objects and their properties may be explicitly stored. The lifespan of products in the acquisition process can be based on the *Product-Quote-Request-Response-Requisition-Order-InventoryItem-StoredItem* cycle displayed in Figure 6

Meta-Characterization of Components, Units, and Associations

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

The skeleton information is kept by a meta-characterization information that allows to keep track on the purpose and the usage of the components, units, and associations. Meta-characterization can be specified on the basis of dockets [SS99] that provide information. The following frames follows the co-design approach [Tha00] with the integrated design of structuring, functionality, interactivity and context. The frame is structured into general information provided by the header, application characterization, the content of the unit and *documentation* of the implementation.

- on the content (*abstracts* or *summaries*),
- on the delivery instruction,
- on the parameters of functions for treatment of the unit (opening with(out) zooming, breath, size, activation modus for multimedia components etc.)
- on the tight association to other units (versions, releases etc.),
- on the meta-information such as resources, restriction, copyright, roles, distribution policy etc.
- on the content providers, content reviewers and review evaluators with quality control policies,
- on applicable workflows and the current status of completion and
- on the log information that enable in tracing the object's life cycle.

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

3 Non-invasive Database Component Composition

`Construction Requirements`
Component construction is based on a general component architecture or a skeleton. Each component is developed in separate. The advantage of the strict separation is an increase of modularisation, parameterisability and conformance to standards.

We derive now a none-invasive construction approach which does not change components used for construction. Due to this restriction we gain a number of properties such as *adaptivity*, *seemless gluing*, *extensibility*, *aspect separation*, *scalability*, and *metamodelling* and *abstraction*.

`Components and Harnesses`
The construction is based on harnesses and the application skeleton. The skeleton is a special form of a meta-schema architecture. It consists of a set of components and a set of harnesses for superposition operations. Harnesses are similar to wiring harnesses used in electrotechnics. A harness consists of a set of input-output channels that can be used to combine wrapped components.

Given a sets of components $\mathfrak{K} = \{\mathcal{K}_1, ..., \mathcal{K}_m\}$ and labels $\mathfrak{L} = \{L_1, ..., L_n\}$ with $n \geq m$. Given furthermore a total function $\tau : \mathfrak{L} \rightarrow \mathfrak{K}$ used for assigning roles to components in harnesses. The triple $(\mathfrak{K}, \mathfrak{L}, \tau)$ is called harness skeleton $\mathfrak{H}$. The arity of the skeleton is n.

The skeleton is graphically represented by doubly rounded boxes. Components are graphically represented by rounded boxes. The construction may lead to complex components called *units*.

The example in Figure 3 has been used in one of our projects. Parliamentarians and inhabitants are combined into a component *Users*. We may use a large variety of positions. A user may use a certain service through some devices. Appointments are based on the usage of services. Tools vary depending on services and on equipment. The final schema contains more than 2.500 attribute, entity, cluster and relationship types. The skeleton of the application is rather simple.

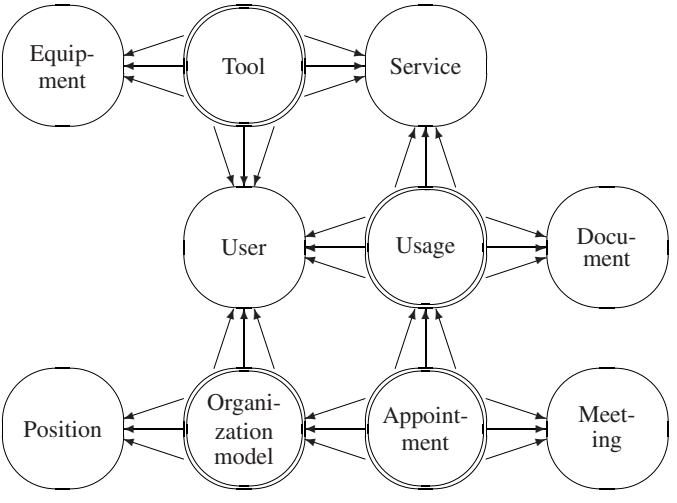


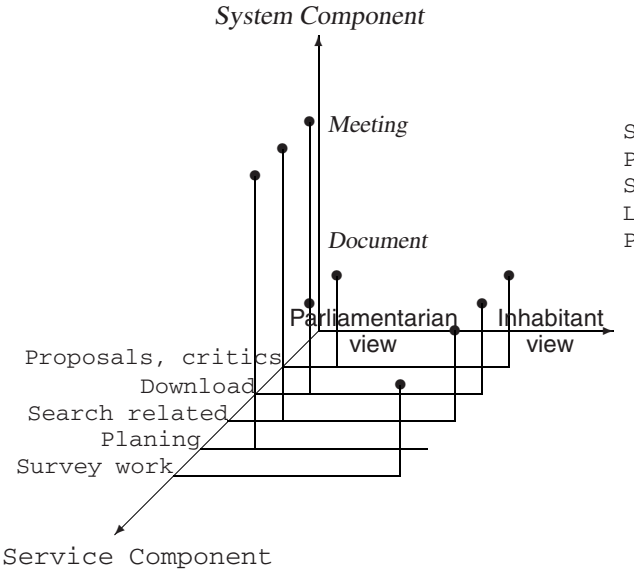
Fig. 3. Skeleton of a Schema for *e-Government Service* Applications

`Harness Filters`

Components may be associated in a variety of ways. In the application in Figure 3 the usage of services depends on the properties of parties, the tools they may use, and the services provided. Services, parties, and tools have their own dimensionality. If we use the classical approach to schema development each subtype may cause the introduction of a new usage type. The schema explodes due to the introduction of a large variety of usage type. To overcome this difficulty we introduce filters.

Given component schemata of an n-ary harness skeleton. A filter of an n-ary harness is an n-ary relation defined of the multi-dimensional structure of the components, i.e. on the views defined for the components.

Filters may be represented either graphically or in a tabular form. In our example, we obtain the following filter. Components are already presented in Figure 3. We develop a number of services which might be used depending on the role, rights, and positions of the users. For instance, the parliamentarian is interested in search of related documents in the role of an inhabitant and in search of related meetings.



	Parliam.	Inhab.
Sur	*Doc*	*Doc*
Pla	*Meet,Doc*	*Doc*
SRe	*Meet*	*Doc*
Load	*Meet*	-
PrCr	-	*Doc*

The implementation of filters is rather straightforward. Each harness has a filter. Since views are defined together with their identification mechanism, an n-ary harness may be represented by an $(n+1)$-ary relationship type associating the components with their roles and extended by the filter.

A harness consists of the harness skeleton $\mathfrak{H} = (\mathfrak{K}, \mathfrak{L}, \tau)$ and the harness filter $\mathfrak{F} = \{(L_i, \mathcal{V}^{L_i}) \,|\, 1 \leq i \leq n,\ L_i \in \mathfrak{L},\ \mathcal{V}^{L_i} \subseteq \mathcal{V}_{\tau(L_i)}\}$ for a set of wrapped components $(\mathcal{K}_i, \mathcal{V}_i)$.

Operators Used For Non-Invasive Schema Construction

In [Tha03b] a number of composition operators for construction of entity and relationship types has been introduced: constructor-based composition, bulk composition, lifespan composition (architecture-based composition, evolution composition, circulation composition, incremental composition, network composition, loop composition), and context composition.

We generalize now these composition operators to component-based schema construction.

Constructor harnesses are based on composition operations such as *product*, *nest*, *disjoint union*, *difference* and *set* operators.

Bulk harnesses allow to bound components, types or classes which share the same skeleton. Two harness skeletons $\mathfrak{H}_1 = (\mathfrak{K}_1, \mathfrak{L}_1\tau_1)$ and $\mathfrak{H}_2 = (\mathfrak{K}_2, \mathfrak{L}_2\tau_2)$ are called unifiable if they are defined over the same set of components, $|\,\mathfrak{L}_1\,| = |\,\mathfrak{L}_1\,| = n$, and there exists a permutation ρ on $\{1, ..., n\}$ such that $\mathcal{K}_{\tau_1(i)} = \mathcal{K}_{\tau_2(\rho(i))}$. The bulk harness of unifiable harnesses $\mathfrak{H}_1, ..., \mathfrak{H}_p$ is constructed by renaming the labels L_j of each harness $\mathfrak{H}_i$ to $L_{i,j}$ and combining the label functions τ_i.

Application-separating harnesses: An enterprize is usually split into departments or units which run their own applications and use their own data. Sharing of data is provided by specific harnesses.

Distribution-based harnesses: Data, functions and control may be distributed. The exchange is provided through specific combinations which might either be based on exchange components that are connected to the sites by harnesses or be based on combination harnesses.

Application-separation-based harnesses have been widely used for complex structuring. The architecture of SAP R/3 often has been displayed in the form of a waffle. For this reason, we prefer to call this composition *waffle composition* or architecture composition displayed in Figure 4.

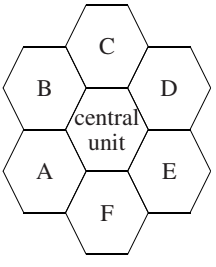


Fig. 4. The Waffle Architecture Composition

An Application of Component Composition

A typical lifespan construction is the *Order* chain displayed in Figure 6. We discover a chain in the ordering and trading process: *Quote*, *Request*, *Response*, *Requisition*, *Order*, *Delivery*, *Billing*, *Payment*. Within this chain, parameters such as people responsible in certain stages are inherited through the components. They are included into the type for the purpose of simpler maintenance. They cannot be changed within the type inheriting the component. Thus, we use an extended inheritance of structuring beyond the inheritance of identification.

At the same time, this schema can be constructed on the basis of components. We may distinguish only four basic parts. Parties are either organisations or people. Products have a number of properties that are independent on parties. The two components are associated within the ordering and trading process. The parties may play different roles within this process. The parties act based on these roles. So, the component schema is given in Figure 5.

The roles of parties in the ordering and trading process can be unfolded. We observe a role of a supplier, of a requestor, of a responding party, of a requisition party and finally the role of the orderer. At the same time, the final order has a history or a lifespan. We may apply the lifespan constructor as well. The application can be either based on collaborating components are can be condensed to the schema given in Figure 6. This schema combines components and unfolds roles and expands the ordering and trading activities. We notice that this schema is not necessarily the solution for the ordering and

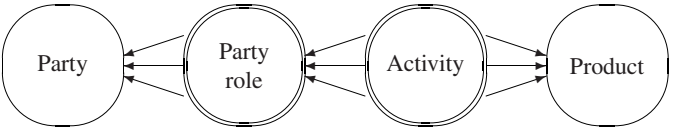


Fig. 5. Component Schema for Product Acquisition Activities

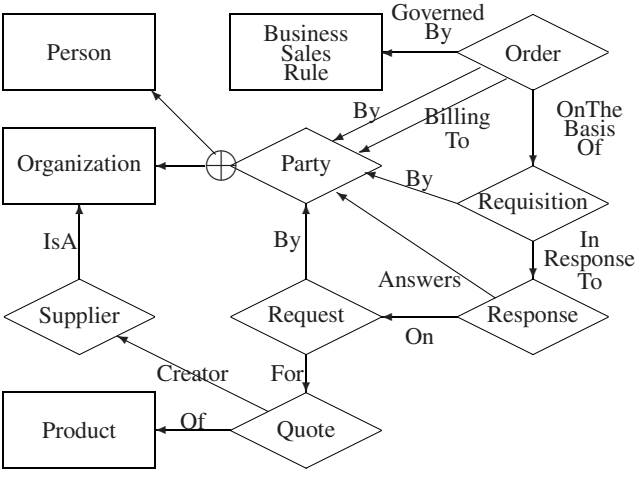


Fig. 6. The Database Schema of the Ordering and Trading Process After Composition

trading process. We may use the components instead and explicitly model component collaboration. In this case the components may stay non-integrated.

4 Collaborating Database Component Suites

Services Provided By Components For Loosely Coupled Suites

A service consists of a wrapped component $(\mathcal{K}_i, \mathcal{V}_i)$, the competencies $\Sigma_{(\mathcal{K}_i,\mathcal{V}_i)}$ provided and properties $\Psi_{(\mathcal{K}_i,\mathcal{V}_i)}$ guaranteeing service quality. Wrapped components offer their own data and functions through their views. The *competence of a service* manifests itself in the set of tasks $\mathcal{T}$ that may be performed and in the guarantees for their quality.

Database Component Collaboration

Instead of expanding and unfolding the component schema in Figure 5 we may follow a different paradigm. The four basic parts are loosely associated by a collaboration, are supported by component databases and communicate for task resolution. This approach has already been tried for distributed databases. Our approach is far more general and provides a satisfying solution.

A collaborating database component suite $\mathfrak{S} = (\mathfrak{K}, \mathfrak{H}, \mathfrak{F}, \Sigma)$ consists of

- an set $\mathfrak{K}$ of wrapped database components $(\mathcal{K}_i, \mathcal{V}_i)$
- a harness consisting of the harness skeleton $\mathfrak{H} = (\mathfrak{K}, \mathfrak{L}, \tau)$ and the harness filter $\mathfrak{F}$,
- an collaboration schema $\mathfrak{F}$ among these components based on the harness, and
- obligations Σ requiring maintenance of the collaboration.

The *collaboration schema* explicitly models collaboration among components. We distinguish three basic processes of component collaboration:

Communication is defined via exchange of messages and information or simply defined via services and protocols [Kön03]. It depends on the choice of media, transmission modes, meta-information, conversation structure and paths, and on the restriction policy. Communication must be based on harnesses.

Coordination is specified via management of components, their activities and resources. It rules collaboration. The specification is based on the pre-/post-articulation of tasks and on the description management of tasks, objects, and time. Coordination may be based on loosely or tightly integrated activities, may be enabled, forced, or blocked. Coordination is often specified through contracts and refines coordination policies.

Cooperation is the production of work products taking place on a shared space. It can be considered as the workflow or life case perspective. We may use a specification based on storyboard-based interaction that is mapped to (generic and structured) workflows. The information exchange is based on component services [ST06a] for production, manipulation, organization of contributions.

This understanding has become now a folklore model for collaboration but has not yet been defined in an explicit form. We use the separation of concern for the specification of component collaboration.

Collaboration obligations are specified through the collaboration style and the collaboration pattern.

The *collaboration style* is based on four components describing

supporting programs of the connected component including collaboration management;

data access pattern for data *release* through the net, e.g., broadcast or P2P, for *sharing* of resources either based on transaction, consensus, and recovery models or based on replication with fault management, and for *remote access* including scheduling of access;

the style of collaboration on the basis of component models which restrict possible communication;

and the coordination workflows describing the interplay among parties, discourse types, name space mappings, and rules for collaboration.

Collaboration pattern generalize protocols and their specification [Kön03]. They include the description of components, their responsibilities, roles and rights. We know a number of collaboration pattern supporting *access and configuration* (wrapper, facade, component configuration, interceptor, extension interface), *event processing* (reactor, proactor, asynchronous completion token, accept connector), *synchronization*

(scoped locking, strategized locking, thread-safe interface, double-checked locking optimization) and *parallel execution* (active object, monitor object, half-sync/half-async, leader/followers, thread-specific storage).

Exchange frames combine the collaboration schema with the collaboration obligations. The collaboration schema can be considered to be an exchange architecture that may also include the workplace of the client using the component suite.

Supporting Collaboration Schemata By Service Managers

The abstraction layer model [Tha00, ST06b] distinguishes between the application domain description, the requirements prescription, the system specification, and the logical or physical coding. The specification layer typically uses schemata for specification. These schemata may be mapped to logical codings. The mapping of services to logical database components is already given by classical database textbooks. We map collaboration schemata to service managers. This mapping provides also a framework for characterisation of competencies and quality.

The *service manager* $\mathcal{M}an$ supports functionality and quality of services and manages sets of wrapped components. The manager supports a number of features for collaboration. The architecture of the services manager follow the separation of concern into communication, coordination, and cooperation. We may thus envision the architecture in Figure 7.

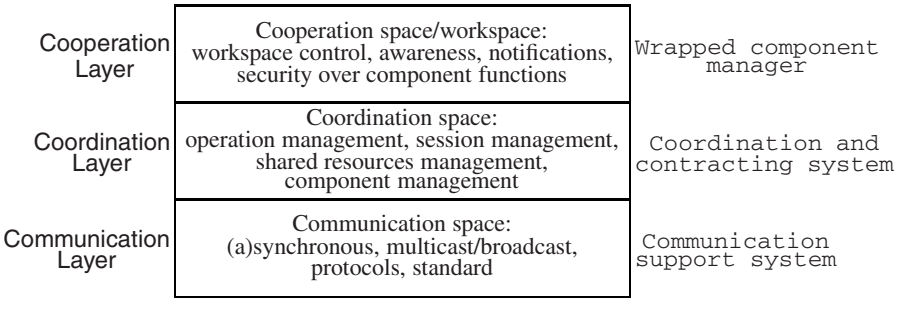


Fig. 7. Layers of a services manager for typical collaborating components

Collaborating services are defined by the quadruple $\mathcal{S} = (\mathfrak{S}, \mathcal{M}an, \Sigma_{\mathcal{S}}, \Psi_{\mathcal{S}})$ describing (Collaborating Suite, Service Manager, Competence, Characteristics). The competence is derived from the competence of the services. The quality of collaborating services may also be derived from the quality properties of components in the suite based on the properties of the harnesses, their collaboration schema, and the corresponding obligations. Typically, quality heavily depends on the suite properties. For instance, reliability of a suite may be less than the reliability of its components.

Concluding by Demonstrating the Potential of Privacy Supporting Suites

Let us show the potential of loosely coupled database component suites for privacy workbenchs. Privacy research is becoming the “poor cousin” among the mainstream research. Novel applications such as Web2.0 have created a new rush towards social

networking and collaborative applications. This enables new possibilities, but also is a threat to users' privacy and data. On the surface, many people seem to like giving away their data to others in exchange for building communities or like to get bribes from companies in exchange of privacy. A number of hidden privacy implications of some Web2.0 and Identity2.0 services, standards and applications can be observed here. At the same time, it is often stated that there is no way to properly preserve privacy.

We show the potential of collaborating databases based on the infon model of [AFFT05]. An infon is a discrete item of information of an individual and may be parametric. The *parameters* are objects, and the so-called *anchors* assign these objects such as agents to parameters.

We may distinguish four relationships between infons and individuals (people), institutions, agencies, or companies: An infon may be possessed by an individual, institution, agency, or company. For example, an individual may possess private information of another individual or, a company may have in its database, private information of someone. Individuals know that an infon is in possession of somebody else. Infons may belong to individuals. Finally, an infon is owned by an individual. The ownership is the basis for the specification of privacy.

The owner sovereignty principle restrains the right or sovereignty of people over their owned infons. A *policy* supporting the owner sovereignty principle restrains the possessor in the role of 'content and topic observer' and preserves the owner in the role of 'informed owner' and 'refresher'. The contract between owner and possessor restricts the possibilities and rights of the possessor for using content and topics on an ongoing basis by additional actions such as

- to monitor activities of the possessor,
- to collect information (about conditions of possession),
- to give a warning to the owner, and
- to take actions such as use, security, welfare, accuracy, correctness, and maintenance of infons to the owner.

The collaboration is *faithful* if the portfolio and profile of contracting possessor do not include any forbidden action or ability, all reporting obligations are observed, and the proprietor is able to observe obligations applied to the possessor.

The private database is called *information wallet* if it is a component service with the following additional function enhancements for owners o, possessors p, infons i, infon requests r_i, time stamps t, delivered infon streams identifiers s_i, public keys $puk(r_i, o, p, t)$ for p, private keys $prik(i, o, p, t)$ for o, records of delivered infons by the owner $store(o, i, p, s_i)$, and encoding and decoding relations $encrypt(i, prik, s_i)$, $decrypt(p, r_i, s_i, puk, t)$ extended by steganographic watermarking $mark(i, o, p)$ for infons:

- $satisfy(request(r_i, o, p, t)) \Rightarrow$
  $encrypt(i, prik(i, o, p, t), s_i)) \wedge deliver(p, o, s_i)) \wedge store(o, i, p, s_i))$
- $decrypt(p, r_i, s_i, puk(s_i, o, p, t), t') \Rightarrow$
  $inform(o, Act(p, s_i, decrypt), t') \wedge mark(i, o, p)$
- $read(p, mark(i, o, p), t') \Rightarrow inform(o, Act(p, s_i, read), t')$
- $send(p, mark(i, o, p), p', t') \Rightarrow inform(o, Act(p, s_i, send(p, p')), t') \wedge$
  $\neg send(p, mark(i, o, p), p', t') \wedge send(p, r_i, p', t')$

– $satisfy(request(puk(s_i, o, p, t), o, p, t')) \Rightarrow$
$deliver(p, o, puk(s_i, o, p, t)) \wedge store(o, i, p, puk(s_i, o, p, t))$.

We assume that watermarked infons cannot be changed by anybody. We can show now that information wallets preserve the owner sovereignty principle.

References

[AFFT05] Al-Fedaghi, S.S., Fiedler, G., Thalheim, B.: Privacy enhanced information systems. In: Proc. EJC'05. Informaton Modelling and Knowledge Bases, Tallinn. Series Frontiers in Arificial Intelligence, vol. XVII, IOS Press, Amsterdam (2005)

[BP00] Biskup, J., Polle, T.: Decomposition of database classes under path functional dependencies and onto contraints. In: Schewe, K.-D., Thalheim, B. (eds.) FoIKS 2000. LNCS, vol. 1762, pp. 31–49. Springer, Heidelberg (2000)

[Fey03] Feyer, T.: A Component-Based Approach to Human-Computer Interaction - Specification, Composition, and Application to Information Services. PhD thesis, BTU Cottbus, Computer Science Institute, Cottbus (Dezember 2003)

[FT02] 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)

[Kön03] König, H.: Protocol Engineering: Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen. Teubner, Stuttgart (2003)

[Kud82] Kudrjavcev, V.B.: Functional systems (in Russian). Moscov Lomonossov University Press, Moscov (1982)

[Mal70] Malzew, A.I.: Algebraic systems. Nauka, Moscow (1970)

[PBGG89] Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: The structure of the relational database model. Springer, Heidelberg (1989)

[SS99] Schmidt, J.W., Schering, H.-W.: Dockets: a model for adding vaulue to content. In: Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E. (eds.) ER 1999. LNCS, vol. 1728, pp. 248–262. Springer, Heidelberg (1999)

[ST04] Schmidt, P., Thalheim, B.: Component-based modeling of huge databases. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 113–128. Springer, Heidelberg (2004)

[ST06a] Schewe, K.-D., Thalheim, B.: Component-driven engineering of database applications. In: APCCM'06, vol. CRPIT 49, pp. 105–114 (2006)

[ST06b] Schewe, K.-D., Thalheim, B.: Usage-based storyboarding for web information systems. Technical Report 2006-13, Christian Albrechts University Kiel, Institute of Computer Science and Applied Mathematics, Kiel (2006)

[Tha00] Thalheim, B.: Entity-relationship modeling – Foundations of database technology. Springer, Heidelberg (2000)

[Tha02] 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)

[Tha03a] Thalheim, B.: Database component ware. ADC'2003, Australian Computer Science Communications 25(2), 13–26 (2003)

[Tha03b] Thalheim, B.: Database component ware. Proc. ADC'2003, Journal on Research and Practice in Information Technology 17, 1–13 (2003)

[Tha05] Thalheim, B.: Component development and construction for database design. Data and Knowledge Engineering 54, 77–95 (2005)