Co-Design of Structuring, Functionality, Distribution, and Interactivity for Information Systems

Bernhard Thalheim

Computer Science and Applied Mathematics Institute,
University Kiel, Olshausenstrasse 40, 24098 Kiel, Germany
Email: thalheim@is.informatik.uni-kiel.de

Keywords: Co-design, database structure, information system functionality, interaction specification, distribution specification, entity-relationship model

Abstract

Database development has mainly be considered as development of database structuring. Functionality and interactivity specification has been neglected in the past. The derivability of functionality has been a reason for this restriction of the database design approach. At the same time, applications and the required functionality became more complex. Functionality specification may be based on workflow engines. Interaction support is often not specified but hidden within the interfaces. It may, however, be specified through the story space and the supporting media type suite. Distributed applications are based on an explicit specification of import/export views, on services provided and on exchange frames. The integration of all these parts has not yet been performed. The co-design approach aims in bridging all these different aspects of applications and provides additionally a sound methodology for development of all these aspects.

1 Introduction

1.1 Information Systems Design and Development

The problem of information system design can be stated as follows:
Design the logical and physical structure of an information system in a given database management system (or for a database paradigm), so that it contains all the information required by the user and required for the efficient behavior of the whole information system for all users. Furthermore, specify the database application processes and the user interaction.

The implicit goals of database design are:

- to meet all the information (contextual) requirements of the entire spectrum of users in a given application area;
- to provide a "natural" and easy-to-understand structuring of the information content;
- to preserve the designers entire semantic information for a later redesign;
- to achieve all the processing requirements and also a high degree of efficiency in processing;
- to achieve logical independence of query and transaction formulation on this level;
- to provide a simple and easily to comprehend user interface family.

Over the last years database structures have extensively been discussed. Almost all open questions have been satisfactorily solved. Modeling includes, however, more aspects:

Structuring of a database application is concerned with representing the database structure and the corresponding static integrity constraints.

Functionality of a database application is specified on the basis of processes and dynamic integrity constraints.

Distribution of information system components is specified through explicit specification of services and exchange frames.

Interactivity is provided by the system on the basis of foreseen stories for a number of envisioned actors and is based on media objects which are used to deliver the content of the database to users or to receive new content.

This understanding has led to the **co-design approach** to modeling by specification **structuring**, **functionality**, **distribution**, and **interactivity**. These four aspects of modeling have both syntactic and semantic elements.

1.2 Information System Models in General

Database design is based on one or more database models. Often, design is restricted to structural aspects. Static semantics which is based on static integrity constraints is sparsely used. Processes are then specified after implementing structures. Behavior of processes can be specified by dynamic integrity constraints. In a late stage, interfaces are developed. Due to this orientation the depth of the theoretical basis is different as shown in the following table displaying the state of the art in the 90ies:

	Used in practice	Theoretical background
Structures	well done	well developed
Static semantics	partially used	well developed
Processes	somehow done	parts and pieces
Dynamic semantics	some parts	parts and glimpses
Services	implementations	ad-hoc
Exchange frames	intentionally done	nothing
Interfaces	intuitive	nothing
Stories	intuitive	nothing

Database design requires consistent and well-integrated development of structures, processes, distribution, *and* interfaces. We will demonstrate below that extended entity-relationship models allow to handle all four aspects.

2 Specification of Structuring

2.1 Languages for Structure Specification

Structuring of databases is based on three interleaved and dependent parts:

Syntactics: Inductive specification of structures uses a set of base types, a collection of constructors and an theory of construction limiting the application of constructors by rules or by formulas in deontic logics. In most cases, the theory may be dismissed. Structural recursion is the main specification vehicle.

Semantics: Specification of admissible databases on the basis of static integrity constraints describes those database states which are considered to be legal. If structural recursion is used then a variant of hierarchical first-order predicate logics may be used for description of integrity constraints.

Pragmatics: Description of context and intension is based either on explicit reference to the enterprize model, to enterprize tasks, to enterprize policy, and environments or on intensional logics used for relating the interpretation and meaning to users depending on time, location, and common sense.

The inductive specification of structuring is based on base types and type constructors. A base type is an algebraic structure $B = (Dom(B), Op(B), Pred(B))$ with a name, a set of values in a domain, a set of operations and a set of predicates. A class B^C on the base type is a collection of elements form $dom(B)$. Usually, B^C is required to be set. It can be a list, a multi-set, a tree etc. Classes may be changed by applying operations. Elements of a class may be classified by the predicates.

A *type constructor* is a function from types to a new type. The constructor can be supplemented with a *selector* for retrieval (such as *Select*) and *update functions* (such as *Insert, Delete*, and *Update*) for value mapping from the new type to the component types or to the new type, with correctness criteria and rules for validation, with default rules, with one or more user representations, and with a physical representation or properties of the physical representation.

Typical constructors used for database definition are the *set*, *tuple*, *list* and *multiset* constructors. For instance, the set type is based on another type and uses algebra of operations such as union, intersection and complement. The retrieval function can be viewed in a straightforward manner as having a predicate parameter. The update functions such as *Insert, Delete* are defined as expressions of the set algebra. The user representation is using the braces $\{, \}$. The type constructors define type systems on basic data schemes, i.e. a collection of constructed data sets. In some database models, the type constructors are based on pointer semantics.

Other useful modeling constructs are *naming* and *referencing*. Each concept type and each concept class has a name. These names can be used for the definition of further types or referenced within a type definition. Often structures include also *optional* components. Optional components and references must be used with highest care since otherwise truly hyper-sophisticated logics such as topoi are required (Schewe 1994). A better approach to database modeling is the requirement of weak value-identifiability of all database objects (Schewe/Thalheim 1993).

2.2 Integrity Constraints

Integrity constraints are used to separate "good" states or sequences of states of a database system from those which are not intended. They are used for specification of semantics of both structures and processes. Therefore, consistency of database applications can not be treated without constraints. At the same time, constraints are given by users at various levels of abstraction, with a variety of vagueness and intensions behind and on the basis of different languages. For treatment and practical use, however, constraints must be specified in a clear and unequivocal form and language. In this case, we may translate these constraints to internal system procedures which are supporting consistency enforcement.

Each structure is also based on a set of **implicit model-inherent integrity constraints**:

Component-construction constraints are based on existence, cardinality and inclusion of components. These constraints must be considered in the translation and implication process.

Identification constraints are implicitly used for the set constructor. Each object either does not belong to a set or belongs only once to the set. Sets are based on simple generic functions. The identification property may be, however, only representable through automorphism groups (Beeri/Thalheim 1998). We shall later see that value-representability or weak-value representability lead to controllable structuring.

Acyclicity and finiteness of structuring supports axiomatization and definition of the algebra. It must, however, be explicitly specified. Constraints such as cardinality constraints may be based on potential infinite cycles.

Superficial structuring leads to representation of constraints through structures. In this case, implication of constraints is difficult to characterize.

Implicit model-inherent constraints belong to the performance and maintenance traps.

Integrity constraints can be specified based on the B(eeri-)V(ardi)-frame, i.e. by an implication with a formula for premises and a formula for the implication. BV-constraints do not lead to rigid limitation of expressibility. If structuring is hierarchic then BV-constraints can be specified within the first-order predicate logic. We may introduce a variety of different classes of integrity constraints defined:

Equality-generating constraints allow to generate for a set of objects from one class or from several classes equalities among these objects or components of these objects.

Object-generating constraints require the existence of another object set for a set of objects satisfying the premises.

A class $\mathcal{C}$ of integrity constraints is called *Hilbert-implication-closed* if it can be axiomatized by a finite set of bounded derivation rules and a finite set of axioms. It is well-known that the set of join dependencies is not Hilbert-implication-closed for relational structuring. However, an axiomatization exists with an unbounded rule, i.e. a rule with potentially infinite premises.

2.3 Representation Alternatives

The classical approach to database objects is to store an object based on strong typing. Each real life thing is thus represented by a number of objects which are either coupled by the object identifier or supported by specific maintenance procedures. In general, however, we might consider two different approaches to representation of objects:

Class-wise, identification-based representation: Things of reality may be represented by several objects. The *object identifier* (OID) supports identification without representing the complex real-life identification. Objects can be elements of several classes. In the early days of object-orientation it has been assumed that objects belong to one and only one class. This assumption has led to a number of migration problems which have not got any satisfying solution.

Structuring based on extended ER models (Thalheim 2000) or object-oriented database systems uses this option. Technology of relational and object-relational database systems is based on this representation alternative.

Object-wise representation: Graph-based models which have been developed in order to simplify the object-oriented approaches (Beeri/Thalheim 1998) display objects by their sub-graphs, i.e. by the set of nodes associated to a certain object and the corresponding edges. This representation corresponds to the representation used in standardization.

XML is based on object-wise representation. It allows to use null values without notification. If a value for an object does not exist, is not known, is not applicable or cannot be obtained etc. the XML schema does not use the tag corresponding to the attribute or the component. Classes are hidden.

Object-wise representation has a high redundancy which must be maintained by the system thus decreasing performance to a significant extent. Beside the performance problems such systems also suffer from low scalability and bad utilization of resources. The operating of such systems leads to lock avalanches. Any modification of data requires a recursive lock of related objects.

For these reasons, objects-wise representation is applicable only under a number of restrictions:

- The application is stable and the data structures and the supporting basic functions necessary for the application are not changed during the lifespan of the system.
- The data set is almost free of updates. Updates, insertions and deletions of data are only allowed in well-defined restricted 'zones' of the database.

A typical application area for object-wise storage are archiving systems, information presentation systems, and content management systems. They use an update system underneath. We call such systems play-out system. The data are stored in the way in which they are transferred to the user. The data modification system has a play-out generator that materializes all views necessary for the play-out system.

Other applications are main-memory databases without update. The SAP database system uses a huge set of related views.

We may use the first representation for our storage engine and the second representation for the input engine or the output engine in data warehouse approaches.

3 Specification of Functionality

3.1 Operations for Information Systems

General operations on type systems can be defined by *structural recursion*. Given types T, T' and a collection type C^T on T (e.g. set of values of type T, bags, lists) and operations such as generalized union $\cup_{C^T}$, generalized intersection $\cap_{C^T}$, and generalized empty elements $\emptyset_{C^T}$ on C^T. Given further an element h_0 on T' and two functions defined on the types $h_1 : T \to T'$ and $h_2 : T' \times T' \to T'$. Then we define the structural recursion by insert presentation for R^C on T as follows

$srec_{h_0,h_1,h_2}(\emptyset_{C^T}) = h_0$

$srec_{h_0,h_1,h_2}(|\!\{\!|s|\!\}\!|) = h_1(s)$
for singleton collections $|\!\{\!|s|\!\}\!|$

$srec_{h_0,h_1,h_2}(|\!\{\!|s|\!\}\!| \cup_{C^T} R^C) =$
$h_2(h_1(s), srec_{h_0,h_1,h_2}(R^C))$
iff $|\!\{\!|s|\!\}\!| \cap_{C^T} R^C = \emptyset_{C^T}$.

All operations of the object-relational database model, the extended entity-relationship model and of other declarative database models can be defined by structural recursion, e.g.,

- selection is defined by $srec_{\emptyset,\iota_\alpha,\cup}$ for the function

$$\iota_\alpha(\{o\}) = \begin{cases} \{o\} & \text{if } \{o\} \models \alpha \\ \emptyset & \text{otherwise} \end{cases}$$

- aggregation functions can be defined based on the two functions for null values

$$h_f^0(s) = \begin{cases} 0 & \text{if } s = \text{NULL} \\ f(s) & \text{if } s \neq \text{NULL} \end{cases}$$

$$h_f^{\text{undef}}(s) = \begin{cases} \text{undef} & \text{if } s = \text{NULL} \\ f(s) & \text{if } s \neq \text{NULL} \end{cases}$$

through structural recursion, e.g.,
$\texttt{sum}_0^{null} = srec_{0,h_{Id}^0,+}$ or
$\texttt{sum}_{undef}^{null} = srec_{0,h_{Id}^{\text{undef}},+}$;
$\texttt{count}_1^{null} = srec_{0,h_1^0,+}$ or
$\texttt{count}_{undef}^{null} = srec_{0,h_1^{\text{undef}},+}$
or the doubtful SQL definition of the average function

$$\frac{\texttt{sum}_0^{null}}{\texttt{count}_1^{null}} .$$

Similarly we may define intersection, union, difference, projection, join, nesting and un-nesting, renaming, insertion, deletion, and update.

Structural recursion is also limited in expressive power. Nondeterministic while tuple-generating programs (or object generating programs) cannot be expressed.

Operations may be either used for retrieval of values from the database or for state changes within the database.

The general frame for operation definition in the co-design approach is based on views used to restrict the scope, pre-, and postconditions used to restrict the applicability and the activation of operations and the explicit description of enforced operations:

Operation φ
[`View:` < View_Name>]
[`Precondition:` < Activation_Condition >]
[`Activated_Operation:` < Specification >]
[`Postcondition:` < Acceptance_Condition >]
[`Enforced_Operation:` < Operation, Condition>]

Operations defined on the basis of this general frame can be directly translated to database programs.

3.2 Dynamic Integrity Constraints

Database dynamics is defined on the basis of transition systems. A transition system on the schema S is a pair

$$\mathcal{TS} = (\mathcal{S}, \{\xrightarrow{a} | \ a \in \mathcal{L}\})$$

where $\mathcal{S}$ is a non-empty set of state variables,
$\mathcal{L}$ is a non-empty set (of labels),
and $\xrightarrow{a} \subseteq \mathcal{S} \times (\mathcal{S} \cup \{\infty\})$ for each $a \in \mathcal{L}$.
State variables are interpreted by states. Transitions are interpreted by transactions on S.
Database lifetime is specified on the basis of paths on $\mathcal{TS}$. A *path* π through a transition system is a finite or ω length sequence of the form $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ....$ The length of a path is its number of transitions.

For the transition system $\mathcal{TS}$ we can introduce now a *temporal dynamic database logic* using the quantifiers $\forall_f$ (always in the future)), $\forall_p$ (always in the past), $\exists_f$ (sometimes in the future), $\exists_p$ (sometimes in the past).

First-order predicate logic can be extended on the basis of temporal operators. The validity function I is extended by time. Assume a temporal class (R^C, l_R). The validity function I is extended by time and is defined on $S(ts, R^C, l_R)$. A formula α is valid for $I_{(R^C,l_R)}$ in ts if it is valid on the snapshot defined on ts, i.e. $I_{(R^C,l_R)}(\alpha, ts) = 1$ iff $I_{S(ts,R^C,l_R)}(\alpha, ts)$.

- For formulas without temporal prefix the extended validity function coincides with the usual validity function.
- $I(\forall_f \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for all $ts' > ts$;
- $I(\forall_p \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for all $ts' < ts$;
- $I(\exists_f \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for some $ts' > ts$;
- $I(\exists_p \alpha, ts) = 1$ iff $I(\alpha, ts') = 1$ for some $ts' < ts$.

The modal operators $\forall_p$ and $\exists_p$ ($\forall_f$ and $\exists_f$ respectively) are dual operators, i.e. the two formulas $\forall_h \alpha$ and $\neg\exists_h \neg\alpha$ are equivalent. These operators can be mapped onto classical modal logic with the following definition:

$$\Box\alpha \equiv (\forall_f \alpha \wedge \forall_p \alpha \wedge \alpha);$$
$$\Diamond\alpha \equiv (\exists_f \alpha \vee \exists_p \alpha \vee \alpha).$$

In addition, temporal operators *until* and *next* can be introduced.

The most important class of dynamic integrity constraint are state-transition constraints $\alpha \ O \ \beta$ which use a pre-condition α and a post-condition β for each operation O. The state-transition constraint $\alpha \, O \, \beta$ can be expressed by the the temporal formula $\alpha \xrightarrow{O} \beta$.

Each finite set of static integrity constraints can be equivalently expressed by a set of state-transition constraints $\{ \wedge_{\alpha \in \Sigma} \alpha \xrightarrow{O} \wedge_{\alpha \in \Sigma} \alpha) \mid O \in Alg(M) \}$.

Integrity constraints may be enforced

- either at the procedural level by application of
  - trigger constructs (Levene/Loizou 1999) in the so-called active event-condition-action setting,
  - greatest consistent specializations of operations (Schewe 1994),
  - or stored procedures, i.e., fully fledged programs considering all possible violations of integrity constraints,
- or at the transaction level by restricting sequences of state changes to those which do not violate integrity constraints,
- or by the DBMS on the basis of declarative specifications depending on the facilities of the DBMS,
- or a the interface level on the basis of consistent state changing operations.

3.3 Specification of Workflows

A large variety of approaches to workflow specification has been proposed in the literature. We prefer formal descriptions with graphical representations and thus avoid pitfalls of methods that are entirely based on graphical specification such as the and/or traps. We use basic computation step algebra introduced in (Thalheim/Düsterhöft 2001):

- Basic control commands are sequence ; (execution of steps in sequence), parallel split $|^{\wedge}|$ (execute steps in parallel), exclusive choice $|^{\oplus}|$ (choose one execution path from many alternatives), synchronization $|^{sync}|$ (synchronize two parallel threads of execution by an synchronization condition ^{sync}, and simple merge + (merge two alternative execution paths). The exclusive choice is considered to be the default parallel operation and is denoted by ||.
- Structural control commands are arbitrary cycles * (execute steps w/out any structural restriction on loops), arbitrary cycles $^+$ (execute steps w/out any structural restriction on loops but at least once), optional execution [] (execute the step zero times or once), implicit termination $\downarrow$ (terminate if there is nothing to be done), entry step in the step $\nearrow$ and termination step in the step $\searrow$.

The basic computation step algebra may be extended by advanced step commands:

- Advanced branching and synchronization control commands are multiple choice $|^{(m,n)}|$ (choose between m and n execution paths from several alternatives), multiple merge (merge many execution paths without synchronizing), discriminator (merge many execution paths without synchronizing, execute the subsequent steps only once) n-out-of-m join (merge many execution paths, perform partial synchronization and execute subsequent step only once), and synchronizing join (merge many execution paths, synchronize if many paths are taken, simple merge if only one execution path is taken).
- We also may define control commands on multiple objects (CMO) such as CMO with a priori known design time knowledge (generate many instances of one step when a number of instances is known at the design time), CMO with a priori known runtime knowledge (generate many instances of one step when a number of instances can be determined at some point during the runtime (as in FOR loops)), CMO with no a priori runtime knowledge (generate many instances of one step when a number of instances cannot be determined (as in a while loop)), and CMO requiring synchronization (synchronization edges) (generate many instances of one activity and synchronize afterwards).
- State-based control commands are deferred choice (execute one of the two alternative threads, the choice which tread is to be executed should be implicit), interleaved parallel executing (execute two activities in random order, but not in parallel), and milestone (enable an activity until a milestone has been reached).

- Finally, cancellation control commands are used, e.g. cancel step (cancel (disable) an enabled step) and cancel case (cancel (disable) the case).

These control composition operators are generalizations of workflow patterns and follow approaches developed for Petri net algebras.

3.4 Architecture of Database Engines

Operating of information systems is modeled by separating the systems state into four state spaces:

$\mathcal{E}R^C$ = (input states $\mathcal{IN}$, output states $\mathcal{OUT}$, engine states $\mathcal{DBMS}$, database states $\mathcal{DB}$).

The input states accommodate the input to the database system, i.e. queries and data. The output space allow to model the output of the DBMS, i.e. output data of the engine and error messages. The internal state space of the engine is represented by engine states. The database content of the database system is represented in the database states. The four state spaces can be structured. This structuring is reflected in all four state spaces. For instance, if the database states are structured by a database schema then the input states are accordingly structured.

Using value-based or object-relational models the database states can be represented by relations. An update imposed to a type of the schema is in this case a change to one of the relations. State changes are modeled on the basis of abstract state machines (Börger/Stärk 2003) through *state change rules*. An engine is specified by its programs and its control. We follow this approach and distinguish between

programs that specify units of work or services and meet service quality obligations and

control **and** coordination that is specified on the level of program blocks with or without *atomicity and consistency* requirements or specified through *job control commands*.

Programs are called with instantiated parameters for their variables. Variables are either static or stack or explicit or implicit variables. We may use furthermore call parameters such as `onSubmit` and `presentationMode`, priority parameters such as `onFocus` and `emphasisMode`, control parameters such as `onRecovery` and `hookOnProcess`, error parameter such as `onError` and `notifyMode`, and finally general transfer parameters such as `onReceive` and `validUntil`.

Atomicity and consistency requirements are supported by the variety of transaction models. Typical examples are flat transactions, sagas, join-and-split transactions, contracts or long running activities (Thalheim 2000).

State changes $T(s_1, ..., s_n) := t$ of a sub-type T' of the database engine $\mathcal{E}R^C$. A set $\mathcal{U} = \{T_i(s_{i,1}, ..., s_{i,n_i}) := o_i \mid 1 \leq i \leq m\}$ of object-based state changes is *consistent*, if the equality $o_i = o_j$ is implied by $T_i(s_{i,1}, ..., s_{i,n_i}) = T_j(s_{j,1}, ..., s_{j,n_j})$ for $1 \leq i < j \leq m$.

The *result* of an execution of a consistent set $\mathcal{U}$ of state changes leads to a new state $\mathcal{E}R^C$ to $\mathcal{E}R^C + \mathcal{U}$

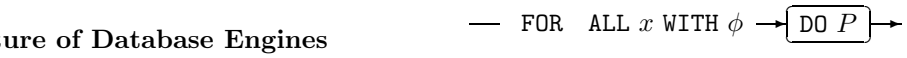
$$(\mathcal{E}R^C + \mathcal{U})(o) = \begin{cases} Update(T_i, s_{i,1}, ..., s_{i,n_i}, o_i) \\ \quad \text{if } T_i(s_{i,1}, ..., s_{i,n_i}) := o_i \in \mathcal{U} \\ \mathcal{E}R^C(o) \\ \quad \text{in the other case} \end{cases}$$

for objects o of $\mathcal{E}R^C$.

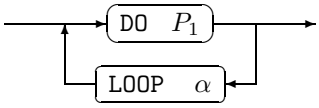
A parameterized programm $r(x_1, ..., x_n) = P$ of arity n consists of a program name r, a transition rule P and a set $\{x_1, ..., x_n\}$ of free variables of P.

An information system $\mathcal{E}R^C$ is a model of ϕ ($\mathcal{E}R^C \models \phi$) if $[\![\phi]\!]_\zeta^{\mathcal{E}R^C} = true$ for all variable assignments ζ for free variables of ϕ.

Two typical program constructors are the execution of a program for all values that satisfy a certain restriction



and the repetition of a program step in a loop



We introduce also other program constructors such as sequential execution, branch, parallel execution, execution after value assignment, execution after choosing an arbitrary value, skip, modification of a information system state, and call of a subprogram.

We use the abstract state machine approach also for definition of semantics of the programs.

A transition rule $\mathcal{P}$ leads to a set $\mathcal{U}$ of state changing operations in a state $\mathcal{E}R^C$ if it is consistent. The state of the information system is changed for a variable assignment ζ to `yields`($\mathcal{P}$, $\mathcal{E}R^C$, ζ, $\mathcal{U}$).

Semantics of transition rules is defined in a calculus that uses rules of the form

$$\frac{\text{prerequisite}_1, ..., \text{prerequisite}_n}{\text{conclusion}} \qquad \text{where condition}$$

For instance, the state change imposed by the first program step is defined by

$$\frac{\forall\, a \in I : \ \text{yields}(\mathcal{P}, \mathcal{E}R^C, \zeta[x \mapsto a], \mathcal{U}_a)}{\text{yields}(\text{FOR ALL } x \text{ WITH } \phi \text{ DO } \mathcal{P}\,,\ \mathcal{E}R^C, \zeta, \bigcup_{a \in I} \mathcal{U}_a)}$$

$$\text{where} \quad I = range(x, \phi, \mathcal{E}R^C, \zeta)$$

The range $range(x, \phi, \mathcal{E}R^C, \zeta))$ is defined by the set $\{o \in \mathcal{E}R^C \mid [\![\phi]\!]_{\zeta[x \mapsto a]}^{\mathcal{E}R^C} = true\}$.

4 Specification of Distribution

Specification of distribution has neglected over a long period. Instead of explicit specification of distribution different collaborating approaches have been tried such as multi-database systems, federated database systems,

4.1 View Suite

Classically, (simple) views are defined as singleton types which data is collected from the database by some query.

```
create view  name (projection variables)
 select       projection expression
   from       Database sub-schema
   where      selection condition
 group by     expression for grouping
   having     selection among groups
 order by     order within the view
```

Since we may have decided to use the class-wise representation simple views are not the most appropriate structure for exchange specification. Instead we use *view suites* for exchange. A *suite* consists of a set of elements, an integration or association schema and obligations requiring maintenance of the association.

Simple examples of a view suites are already discussed in (Thalheim 2000) where view suites are ER schemata. The integration is given by the schema. Obligations are based on the master-slave paradigm, i.e., the state of the view suite classes is changed whenever an appropriate part of the database is changed.

Additionally, views should support services. Services provide their own data and functionality. This object-orientation is a useful approach whenever data should be used without direct or remote connection to the database engine.

We generalize the view specification frame used in relational databases by the frame:

```
generate MAPPING :
                VARS → OUTPUT STRUCTURE
from DATABASE TYPES
where SELECTION CONDITION
represent using GENERAL PRESENTATION
                STYLE
 &    ABSTRACTION (GRANULARITY,
                MEASURE, PRECISION)
 &    ORDERS WITHIN THE PRESENTATION
 &    HIERARCHICAL REPRESENTATIONS
 &    POINTS OF VIEW
 &    SEPARATION
browsing definition CONDITION
 &    NAVIGATION
functions SEARCH FUNCTIONS
 &    EXPORT FUNCTIONS
 &    INPUT FUNCTIONS
 &    SESSION FUNCTIONS
 &    MARKING FUNCTIONS
```

The extension of views by functions seems to be an overhead during database design. Since we extensively use views in distributed environments we save efforts of parallel and repetitive development due to the development of the entire view suite instead of developing each view by its own.

4.2 Services

Services are usually investigating on one of the (seven) layers of communication systems. They are characterized by two parameters: Functionality and quality of service. Nowadays we prefer a more modern approach (Lockemann 2003). Instead of functions we consider *informational processes*. *Quality of service* is bounded by a number of properties that are stated either at implementation layer or at conceptual layer or at business user layer. Services consist of informational processes, the characteristics provided and properties guaranteeing service quality, i.e. $\mathcal{S} = (\mathcal{I}, \mathcal{F}, \Sigma_{\mathcal{S}})$ where $\mathcal{I} = (\mathcal{V}, \mathcal{M}, \Sigma_{\mathcal{T}})$.

Informational processes are specified by the ingredients:

Views from the view suite $\mathcal{V}$ are the resources for informational processes. Since views are extended by functions they are computational and may be used as statistical packages, data warehouses or data mining algorithms.

The services manager $\mathcal{M}$ supports functionality and quality of services and manages containers, their play-out and their delivery to the client. It is referred to as a service provider.

The competence of a service manifests itself in the set of tasks $\mathcal{T}$ that may be performed.

Service characteristics $\mathcal{F}$ is characterized depending on the abstraction layers:

Service characteristics at business user layer are based on service level agreements and the informational processes at this layer.

Service characteristics at conceptual layer describe properties the service must provide in order to meet the service level agreements. Further, functions available to the client at specified by their interfaces and semantic effects.

Service characteristics at implementation layer specify the syntactical interfaces of functions, the data sets provided and their behavior and constraints to the information system and to the client.

Quality of service $\Sigma_{\mathcal{S}}$ is characterized depending on the abstraction layers:

Quality parameters at business user layer may include *ubiquity* (access unrestricted in time and space) and *security* (against failures, attacks, errors; trustworthy).

Quality parameters at conceptual layer subsume *interpretability* (formal framework for interpretation) and *consistency* (of data and functions).

Quality parameters at implementation layer include *durability* (access to the entire information unless it is explicitly overwritten), *robustness* (based on a failure model for resilience, conflicts, and persistency), *performance* (depending on the cost model, response time and throughput), and *scalability* (to changes in services, number of clients and servers).

4.3 Exchange Frames

The exchange frame is defined by

exchange architecture usually provided a system architecture integrating the information systems through communication and exchange systems,

collaboration style specifying the supporting programs, the style of cooperation and the coordination facilities, and

collaboration pattern specifying the roles of the partners, their responsibilities, their rights and the protocols they may rely on.

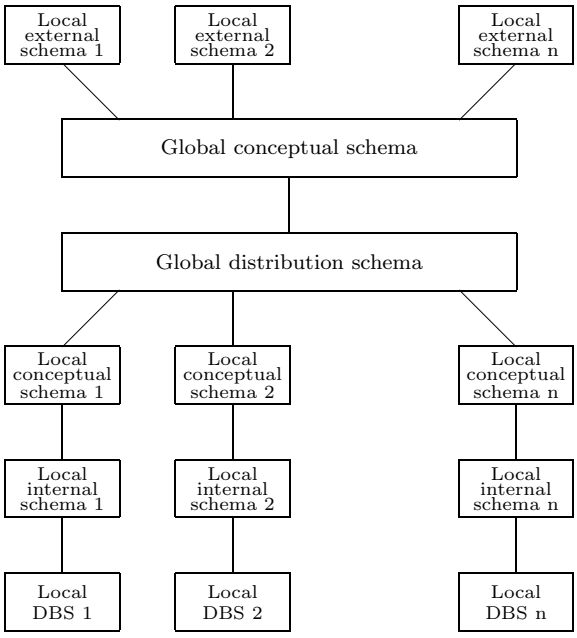


Figure 1: Generalization of the Three-Level Architecture to Distributed Schema

Distributed database systems are based on local database systems and follow a certain integration strategy. Integration is based on total integration of the local conceptual schemata into a global distribution schema. The architecture is displayed in Figure 1.

Beside the classical distributed system we support also other architecture such as *database farms*, *incremental information system societies* and *cooperating information systems*. The later are based on the concept of cooperating views (Thalheim 2000). Incremental information system societies are the basis for facility management systems. Simple incremental information systems are data warehouses and content management systems.

Database farms are generalizing and extending the approaches to federated information systems and mediators. Their architecture is displayed in Figure 2. Farms are based on the co-design approach and the

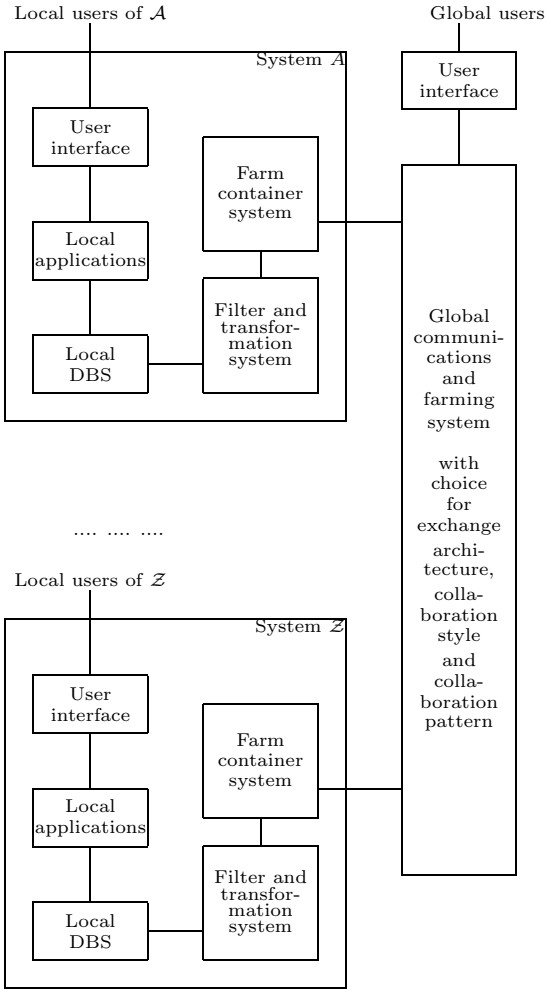


Figure 2: Database Systems Farm

information unit and container paradigm:

Information units are generalized views. Views are generated on the basis of the database. Units are views extended by functionality necessary for the utilization of view data. We distinguish between *retrieval information units* and *modification information units*. The first are used for data injection. The later allow to modify the local database.

Containers support the export and the import of data by bundling information units provided by view states. Units are composed to containers which can be loaded and unloaded in a specific way. The unloading procedure supports the dialogue scenes and steps.

The global communication and farming system provides the exchange protocols, has facilities for loading and unloading containers and for modification of modification information units.

We do not want to integrate entirely the local databases but provide only *cooperating views*.

The exchange architecture may include the workplace of the client describing the *actors*, *groups*, *roles* and *rights* of actors within a group, the *task portfolio* and the *organization* of the collaboration, communication, and cooperation.

The collaboration style is based on four components describing

supporting programs of the information system including session management, user management, and payment or billing systems;

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 peer-to-peer models or component models or push-event models which restrict possible communication;

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

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):

Proxy collaboration uses partial system copies (remote proxy, protection proxy, cache proxy, synchronization proxy, etc.).

Broker collaboration supports coordination of communication either directly, through message passing, based on trading paradigms, by adapter-broker systems, or callback-broker systems.

Master/slave collaboration uses tight replication in various application scenarios (fault tolerance, parallel execution, precision improvement; as processes, threads; with(out) coordination).

Client/dispatcher collaboration is based on name spaces and mappings.

Publisher/subscriber collaboration is also known as the observer-dependents paradigm. It may use active subscribers or passive ones. Subscribes have their subscription profile.

Model/view/controller collaboration is similar to the three-layer architecture of database systems. Views and controllers define the interfaces.

Collaboration pattern generalize protocols. They include the description of partners, their responsibilities, roles and rights.

5 Specification of Interactivity

Interactivity of information systems has been mainly considered on the level of presentation systems by the Arch or Seeheim separation between the application system and the presentation system. Structuring and functionality are specified within a database modeling language and its corresponding algebra. Pragmatics is usually not considered within the database model. The interaction with the application system is based on a set of views which are defined on the database structure and are supported by some functionality.

In the co-design framework we generalize this approach by

introduction of media objects which are generalized views that have been extended by functionality necessary, are adapted to the users needs and delivered to the actor by a container (Schewe/Thalheim 2000) and by

introduction of story spaces (Srinivasa 2001) which specify the stories of usage by groups of users (called actors) in their context, can be specialized to the actual scenario of usage and use a variety of play-out facilities.

User interaction modeling involves several partners (grouped according to characteristics; group representatives are called 'actors'), manifests itself in diverse activities and creates an interplay between these activities. Interaction modeling includes modeling of environments, tasks and actors beside modeling of interaction flow, interaction content and interaction form.

The general architecture of a web information system is shown in Figure 3. This architecture has successfully been applied in more than 30 projects resulting in huge or very large information-intensive websites and in more than 100 projects aiming in building large information systems.

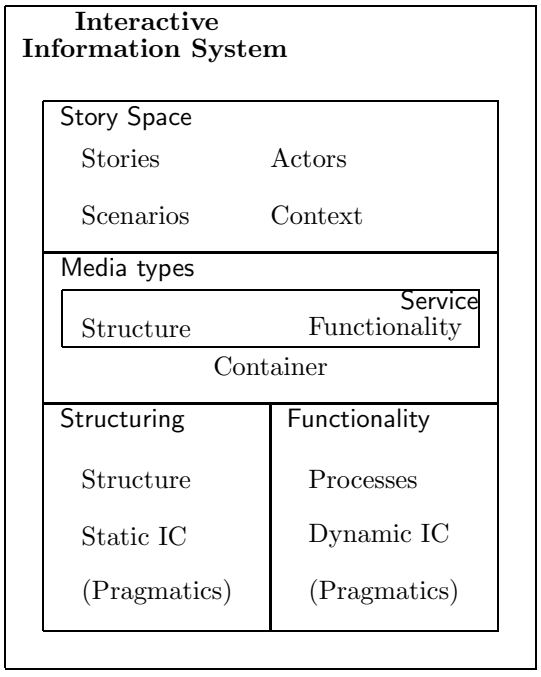


Figure 3: Specification Approach for Information Systems

5.1 Story Space

Modeling of interaction must support multiple scenarios. In this case, user profiles, user portfolios, and the user environment must be taken into consideration. The *story* of interaction is the intrigue or plot of a narrative work or an account of events. The language SiteLang (Thalheim/Düsterhöft 2001) offers concepts and notation for specification of story spaces, scene and scenarios in them.

Within a story one can distinguish threads of activity, so-called *scenarios*, i.e., paths of scenes that are connected by transitions. We define the story space Σ_W as the 7-tuple $(S_W, T_W, E_W, G_W, A_W, \lambda_W, \kappa_W)$ where S_W, T_W, E_W, G_W and A_W are the set of scenes created by W, the set of scene transitions and events that can occur, the set of guards and the set of actions that are relevant for W, respectively. Thus, T_W is a subset of $S_W \times S_W$. Furthermore $\lambda_W : S_W \to SceneSpec$ is a function associating a scene specification with each scene in S_W, and $\kappa_W : T_W \to E_W \times G_W \times A_W$, $t \mapsto (e, g, a)$ is a function associating with each scene transition t occurring in W the event e that triggers transition t, the guard g, i.e. a logical condition blocking the transition if it evaluates to false on occurrence of e, and the action a that is performed while the transition takes place.

We consider scenes as the conceptual locations at which the interaction, i.e., dialogue takes place. Dialogues can be specified using so-called dialogue-step expressions. Scenes can be distinguished from each other by means of their identifier: Scene-ID. With each scene there is associated a media object and the set of actors that are involved in it. Furthermore, with each scene a representation specification is associated as well as a context. Scenes therefore can be specified using the following frame:

Scene = (Scene-ID
  DialogueStepExpression
  Data views with associated functions
  User
    UserID
    UserRight
    UserTasksAssigned
    UserRoles
  Representation (styles, defaults, emphasis, ...)
  Context (equipment, channel, particular)

Dialogue-step expressions consist of dialogues and operators applied to them. A typical scene is displayed in Figure 4. A learner may submit solutions in

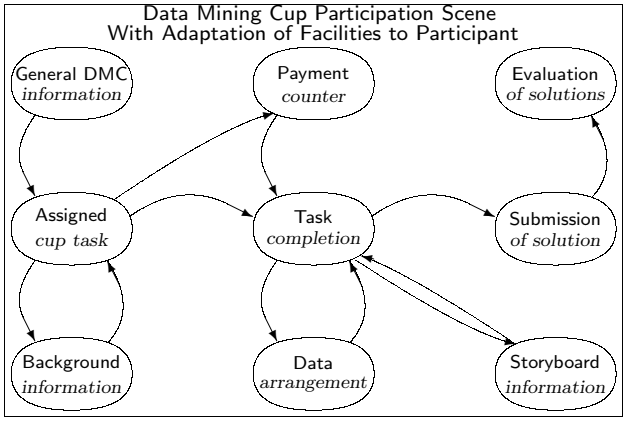


Figure 4: One of the Scenes for Active Learning

the data mining cup. Before doing so, the user must pay a certain fee if she has not already been paying. The system either knows the user and his/her profile. If the user has already paid the fee then the payment dialogue step is not shown. If the user has not paid the fee or is an anonymous user then the fee dialogue step must be visited and the dialogue step for task completion is achievable only after payment.

5.2 Media Type Suite

Media types have been introduced in (Schewe/Thalheim 2000). Since users have very different needs in data depending on their work history, their portfolio, their profile and their environment we send the data packed into containers. Containers have the full functionality of the view suite. Media type suites are based on view suites and use a special delivery and extraction facility. The media type suite is managed by a system consisting of three components:

Media object extraction system: Media objects are extracted and purged from database, information or knowledge base systems and summarized and compiled into media objects. Media objects have a structuring and a functionality which allows to use these in a variety of ways depending on the current task.

Media object storage and retrieval system: Media objects can be generated on the fly whenever we need the content or can be stored in the storage and retrieval subsystem. Since their generation is usually complex and a variety of versions must be kept, we store these media objects in the subsystem.

Media object delivery system: Media objects are used in a large variety of tasks, by a large variety of users in various social and organizational contexts and further in various environments. We use a media object delivery system for delivering data to the user in form the user has requested. Containers contain and manage the set of media object that are delivered to one user. The user receives the user-adapted container and may use this container as the desktop database.

This understanding closely follows the data warehouse paradigm. It is also based on the classical model-view-control paradigm. We generalize this paradigm to media objects, which may be viewed in a large variety of ways and which can be generated and controlled by generators.

6 Integrating Specification Aspects into Co-Design

The languages introduced so far seem to be rather complex and the consistent development of all aspects of information systems seems to be rather difficult. We developed a number of methodologies to development in order to overcome difficulties in consistent and complete development. Most of them are based on top-down or refinement approaches that separate aspects of concern into abstraction layers and that use extension, detailisation, restructuring as refinement operations.

6.1 The Abstraction Layer Model for Information Systems Development

We observe that information systems are specified at different abstraction layers:

1. The *motivation layer* addresses the purpose of the information system, i.e. its mission statement and the anticipated customer types including their goals. The results of the design process are conducted into the *stakeholder contract specification*.

2. The *strategic layer* describes the information system, analyzes business processes and aims in elicitation of the requirements to the information system. The results of the design process are combined into the *system specification*.

3. The *business layer* deals with modelling the anticipated usage of the information system in terms of customer types, locations of the information space, transitions between them, and dialogues and discourses between categories of users (called actors). The result of this abstraction layer is compiled into an *extended system manual* including mockups of the interfaces and scenarios of utilization.

4. The *conceptual layer* integrates the conceptual specification of structuring, functionality, distribution and interactivity. The results of this step are the database schema, the workflows, the view and media type suites, the specification of services and exchange frames, and the story space.

5. At the *implementation layer*, logical and physical database structures, integrity enforcement procedures, programs, and interfaces are specified within the language framework of the intended platform. The result of specification at the implementation layer is the *implementation model*. This model is influenced by the builder of the information system.

6. The exploitation layer is not considered here. Maintenance, education, introduction and administration are usually out of the scope of conceptualization of an application.

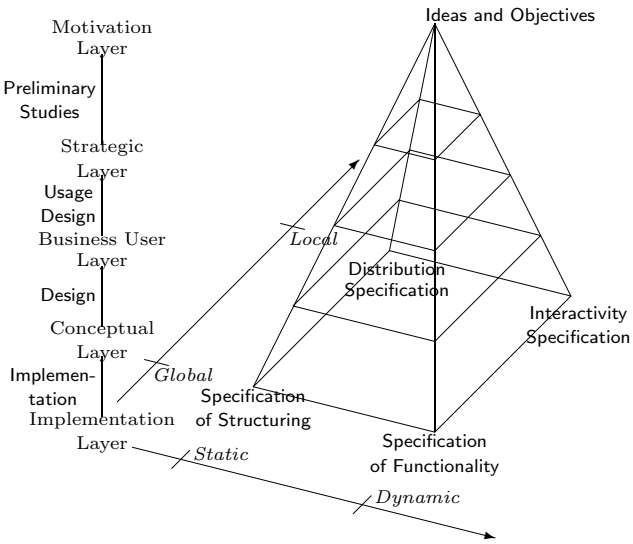


Figure 5: The Abstraction Layer Model of the Database Design Process

6.2 The Co-Design Methodology

Methodologies must confirm both with the SPICE v. 2.0 and SW-CMM v. 2.0 requirements for consistent system development. The co-design framework is based on a step-wise refinement along the abstraction layers. Since the four aspects of information systems - structuring, functionality, distribution and interactivity - are interrelated they cannot be developed separately. The methodology sketched below is based on steps in the following specification frame:

Rule # i **Name of the step**	Task 1. Task 2. ...
Used documents	Documents of previous steps (IS development documents) Customer documents and information

Documents under change	IS development documents Contracts
Aims, purpose and subject	General aims of the step Agreed goals for this step Purpose Matter, artifact
Actors involved	Actor A, e.g., customer representatives Actor B, e.g. developer
Theoretical foundations	Database theory Organization theory Computer science Cognition, psychology, pedagogic
Methods and heuristics	Syntax and pragmatics Used specification languages Simplification approaches
Developed documents Results	IS development documents Results and deliverables
Enabling condition for step	Information gathering conditions fulfilled by the customer side Information dependence conditions Conditions on the participation
Termination condition for step	Completeness and correctness criteria Sign-offs, contracts Quality criteria Obligation for the step fulfilled

The steps used in one of the methodologies are:

Motivation layer

1. Developing visions, aims and goals
2. Analysis of challenges and competitors

Strategic layer

3. Separation into system components
4. Sketching the story space
5. Sketching the view suite
6. Specifying business processes

Business user layer

7. Development of scenarios of the story space
8. Elicitation of main data types and their associations
9. Development of kernel integrity constraints, e.g., identification constraints
10. Specification of user actions, usability requirements, and sketching media types
11. Elicitation of ubiquity and security requirements

Conceptual layer

12. Specification of the story space
13. Development of data types, integrity constraint, their enforcement
14. Specification of the view suite, services and exchange frames
15. Development of workflows
16. Control of results by sample data, sample processes, and sample scenarios
17. Specification of the media type suite
18. Modular refinement of types, views, operations, services, and scenes
19. Normalization of structures
20. Integration of components along architecture

Implementation layer

21. Transformation of conceptual schemata into logical schemata, programs, and interfaces
22. Development of logical services and exchange frames
23. Developing solutions for performance improvement, tuning
24. Transformation of logical schemata into physical schemata
25. Checking durability, robustness, scalability, and extensibility

Concluding Remark

The co-design methodology has been practically applied in a large number of information system projects and has nevertheless a sound theoretical basis. We do not want to compete with UML but support system development at a sound basis without ambiguity, ellipses and conceptual mismatches.

References

Beeri C. & Thalheim B. (1998), 'Identification as a Primitive of Database Models', *FoMLaDO'98*, Kluwer Acad. Publ., London 19-36.

Börger E. & Stärk R. (2003), *Abstract state machines*, Springer, Berlin.

Lockemann P.C. (2003), 'Information system architectures: From art to science', *Proc. BTW'2003*, 1-27.

Levene M. & Loizou, G. (1999), *A guided tour to relational databases and beyond*, Springer.

Schewe K.-D. & Thalheim B., 'Fundamental concepts of object oriented databases', *Acta Cybernetica*, **11**, No. 4, 1993, 49-81.

Schewe K.-D. (1994), *The specification of data-intensive application systems*, Advanced PhD, TU Cottbus, 1994. 5 'Towards a theory of consistency enforcement', *Acta Informatica*,

Schewe K.-D. & Thalheim B. (2000), 'Modeling interaction and media objects', *Proc. NLDB' 2000*, LNCS 1959, 313-324.

Srinivasa S. (2001), *A calculus of fixpoints for characterizing interactive behavior of information systems*, PhD, BTU Cottbus, Faculty of Mathematics, Natural Sciences and Computer Science, Cottbus.

Thalheim, B. (2000), *Entity-relationship modeling – Foundations of database technology*, Springer.

Thalheim B. & Düsterhöft A. (2001), 'SiteLang: Conceptual modeling of internet site', *Proc. ER'2001*, LNCS 2224, Springer, 179-192.

Remark: Our main aim has been the presentation of the co-design framework. We restrict thus the bibliography only to those references which are necessary for this paper. An extensive bibliography on relevant literature in this field can be found in (Thalheim 2000).

Acknowledgement. I want to thank Hans-Joachim Klein for his comments and discussions.