Efficient Modelling Techniques for Complex Problems

Reinhard Karge
run Software-Werkstatt GmbH
Koepenicker Strasse 325
12555 Berlin, Germany

ABSTRACT

This paper demonstrates theoretical and practical approaches for designing and implementing complex data models. It is based on research work made for developing the OODBMS ODABA2 as well as on practical experiences. Two medium object models will be used as examples: the Integrated Metadata System for statistical offices **Bridge**, developed within the frame of the DOSIS project (EUROSTAT) and an administration system for large enterprises **Belami**. Both consist of more than 100 persistent object types (including more than 300 relationships) and about 300 additional transient classes (implementation classes). This corresponds to a relational database model with about 500 relations.

Keywords: object-oriented database, metadata, modelling techniques, statistical information system.

1. INTRODUCTION

During the last year we got several requests for building complex database applications. As complex database applications applications are considered that are based on database models with more than 100 object types and more than 200 relationships between these object types. Other characteristics of these projects are that instance collections do contain not many object instances (usually less than 50,000), even though, there might be several million instances for certain object types. Relationships between objects are accessed frequently (each object instance access normally requires access to 50 and more 100 related objects that are linked directly or indirectly with the accessed object).

The resources for implementing applications based on those models were limited to between 12 and 36 person month per application. This included problem analysis, building graphical user interfaces with more than 300 different window forms, implementing processing rules and providing documentation and online help.

To be able to do that object-oriented technologies have been chosen. Besides this general approach more efficient analysing, modelling, implementation and documentation technologies became necessary that are described in this paper.

The technologies used will be demonstrated mainly on the Bridge application. Bridge is an integrated metadata system (IMDS) [3][4] developed for statistical offices during the last two years. Bridge is used in central statistical offices of several European countries. Bridge is based on a quite complex and highly linked object model with a number of quite general requirements as version and multilingual support.

Belami, the second example referenced is an application developed for finance and contract administration for a company with a large number of sub contractors. In this project a number of subset relations demonstrate another important facility of object-oriented modelling.

The paper discusses four areas of application development:

1. Problem analysis
   Methods of efficient problem analysis are discussed in chapter 2. A formalised way of natural language based analysis is described that leads directly to the data model.
2. Enhanced facilities of data model
   In chapter 3 advantages of additional database dimensions are discussed. Additional database dimensions can simplify the data model and are able to solve quite complicate problems as versioning and multilingual support. In chapter 4 hierarchical database structures and instance overload technologies are introduced to reflect information in different contexts. Chapter 5 is considering some advantages resulting from the support of modelling subset relations.
3. Enhanced facilities of functional model
   Chapter 6 describes a more general view to the way of presenting behaviour for object types. Beyond functions or expressions it discusses a number of other method types reflecting the behaviour of objects of a certain type. Chapter 7 discusses data exchange as a special type of behaviour for passive communication with other applications.
4. Enhanced facilities of dynamical model
   Chapter 8 discusses events, actions and reactions as enhanced features of the dynamical model. It introduces reaction mechanisms between active and inactive object instances based on events defined on the system and on the model level.

Most of the concepts described in this paper are part of the ODABA2 database system [2]. General statements about facilities of other object-oriented systems are based on [1].

2. Natural language data model design

Efficient technologies start with the analysis. The closer the model is to the reality the easier it is to maintain later on. One way of efficient problem analysis is *natural language analysis* (NLA). NLA consists of the following steps:

1. Finding relevant terms, i.e. determining terms that are related to relevant objects of the problem.
2. Discovering terms that reflect different views to the same object. Object types correspond to object views (not to objects).
3. Defining relevant terms and relationships between terms (specialisation, generalisation). This step includes also discovering missing terms (e.g. for generalisations the customer is not aware about).
4. Determining relevant attributes for relevant object types. This step may also lead to new object types.

Example: *ClassificationRevision* and *Variable* are two relevant terms for statistical metadata (1). Especially variables are reflected on a conceptual level (*Variable*) and on a technical level (*TechnicalVariable*). Both are different views to the same object (2). *ClassificationRevision* and *Variable* are *MetadataObjects* (generalisation) even though statisticians may not be aware of that (3).

There is a formalised way to define the NLA in a type lexicon. This can be used as the base for an application thesaurus as well as for generating documentation and online help.

The data model strictly follows the terminology defined in the NLA and will be generated directly from the type lexicon as a result of the NLA. There is no reason for defining different object types in the data model than described in the type lexicon. However, there might be extensions defining objects on a technical level.

One basic requirement for the OODBMS resulting from the analysis is that the OODBMS is supporting multiple object views. Multiple object views are modelled usually as object types inheriting from the same base type:

Person
Student (based on Person)
Patient (based on Person)
Employee (based on Person)
Customer (based on Person)

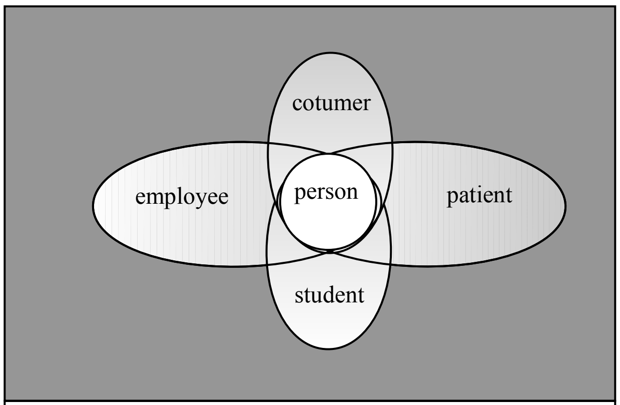


Figure 1: Multiple object views

Most of object oriented systems allow different types of specialisation on the model level, however, any OODBMS do not support multiple object views on the instance level. In this case the database model can not directly reflect the reality.

3. Orthogonal database dimensions

Usually on an abstract level we consider object-oriented databases as two-dimensional databases. One dimension is formed by the data model (types and properties), the other one by instances. Then each value in the database can be defined as value function of the instance identity (i) and a property path (p):

$$v = V(i,p) \qquad (1a)$$

The state or value v_i of an instance i can be defined as a vector containing a value for each property of i.

$$v_i = (V(i, p_1), \dots , V(i, p_m)) \qquad (1b)$$
or
$$v_i = V(i, p_1, \dots ,p_m) \qquad (1c)$$

$p_1, \dots ,p_m$ are the properties associated with the instance type. Other properties could be included as well but do not have any influence on the instance value. For reference properties referring to single instances or instance collections the identity of the referenced instance or collection is considered as value. If the value of a reference is defined as an object identity the state of an instance does not depend on the state of referenced instances or collections. The state or the value of a collection can be defined as the set of instances the collection contains.

$$v_C = \{ i \mid i \in C \} = C \qquad (1d)$$

Hence, the value of a collection does not depend on the state of instances of the collection. But one or more indexes can be defined and stored for each collection. An index x_C for a collection is a function defined on a key domain that returns an instance for each key value:

$$i = x_C(v_k) \qquad (1e)$$

If the index is consistent the key value is the value of the returned instance as well:

$$v_k = V(i, p_k) \qquad (1f)$$
$$p_k = k(p_1, \dots , p_n) \qquad (1g)$$

where p_k can be considered as a function of one or more properties of the instance. This does not include multiple value keys, i.e. keys where (1f) returns a set of values. Now a consistent index x_C for a collection C can be defined as

$$x_C = \{ (i, v_k) \mid i \in C \wedge v_k = V(i, p_k) \} \qquad (1h)$$

or $x_C = X(v_c, p_k)$ (1i)

Thus, (1a), (1c) and (1i) can be considered as fundamental database function within an object-oriented database. Defining orthogonal database dimensions means always to introduce independent arguments in both database functions.

Version dimension

Statistical metadata has to be stored for different versions (e.g. to keep older variable definitions that may influence the observation). Hence, a version dimension becomes necessary to reflect the reality in the model in an adequate way.

The usual way to solve the version problem is to expand the model by inserting a version attribute (e.g. a time stamp) and creating a new instance for each new version. This way has two disadvantages. One is that the instance identity (i) can be determined only based on a version value. That makes it difficult to reference an instance (which version should be referenced?). The other problem is that indexes must include the version attribute as far as they define unique keys. A more natural way is to insert the version (t) as third database dimension:

$$\mathbf{v = V\,(i,p,t)} \qquad \textbf{(2a)}$$

That means that each instance referenced by its instance identity has a version dimension that provides different states for the versions of an object instance. Now a value in the database can be determined by referring to an instance identity, a property and the version (that will be set to "current" as default). References and indexes that are referring to the instance identity (i) are not affected by versioning and it becomes possible to create versioned indexes as well. Within OODBMS ODABA2 the version has been inserted as third database dimension to provide a solution on the OODBMS level. There are two mechanisms to create instance versions.

1. Creating individual versions for each object instance.
2. Using sliced versioning for defining versions on the database level.

Individual versions can be easily created and maintained independently for each object instance, i.e. versions for two different object instances do not relate to each other.

Introducing a version slice, however, guaranties database consistence at each time point. Starting a new version (or introducing a version slice) is a logical operation that ensures that each instance, collection or index modification in the new version will be stored as new version. Only in this case the version becomes a really independent database dimension and the index function for a collection **C** becomes time dependent as well:

$$\mathbf{x_C = X(v_c, p_k, t)} \qquad \textbf{(2b)}$$

Generic attributes

Another problem arises with the requirement of multilingual support. Metadata as classification or variable definitions are requested with several languages in European statistical offices.

This creates similar problems as introducing versions. If objects become language dependent a language attribute has to be introduced that creates multiple instances for an object. This creates not only problems when referring to the object instance but also when defining indexes on object collections. Another way is to reference a number of language depending instances that carry language depending information, however, this usually can not be used for creating indexes (usually only attributes are allowed as key components).

A more general solution that does not affect the database model is to consider the language as an attribute type. Then a generic attribute based on this type generates automatically the required value for the type set for the generic attribute. In other words the generic attribute type (g) becomes an additional database dimension. Now the value of a property for an instance depends on the identity, the property, the generic type (if the property is not a generic attribute the type is set to a constant default value) and the version:

$$\mathbf{v = V\,(i,p,t,g)} \qquad \textbf{(3a)}$$

For really introducing the type as an additional database dimension generic indexes have to be supported as well, i.e. for each type (e.g. each language) a type (language) dependent index has to be created. Because of (1g) and (2b) the index is a function of a number of properties and each property can be represented in a different type. When we define a type vector

$$\mathbf{g_k = (g_1, \dots , g_n)} \qquad \textbf{(3b)}$$

where $\mathbf{g_i}$ is the type for the key property $\mathbf{p_i}$ we get the index as function of an (n+3)-dimensional vector

$$\mathbf{x_C = X(v_c, p_k, t, g_1, \dots , g_n)} \qquad \textbf{(3c)}$$

Practically the number of indexes will explode when a key contains a number of generic attributes because with each generic attribute the dimension for the domain of the index function grows. For practical applications, however, it seems to be sufficient to restrict keys to maximal one generic attribute. Then we get a simplified index function

$$\mathbf{x_C = X(v_c, p_k, t, g)} \qquad \textbf{(3d)}$$

where $\mathbf{g}$ is the type for the generic key property $\mathbf{p_i}$ if there is one (otherwise $\mathbf{g}$ is a constant default value and will not affect the index).

The only known example for an OODBMS supporting n-dimensional databases with the restriction (3d) is ODABA2. Using database dimensions instead of extending the database model simplifies the database model and reduces not only the functionality to be implemented in the application but provides facilities that are hardly to achieve on the application level.

4. Hierarchical information structures

This chapter considers hierarchical information structures from the management point of view, i.e. it will not discuss object hierarchies within the object-oriented database model but how to handle global (common) and context related (particular) information instead. The concept of global and context database objects, more general, of hierarchical database objects allows using data of common interest stored in a global database object. Context related data, however, is stored in a subordinated context database object. Each database object defines its separate data space. Instance overloading allows also overloading common information in the context database object.

Statistical metadata systems consist of global specifications that are defined on the level of statistical office (e.g. global variable definitions, classifications etc.). On the other hand there exist a lot of context related metadata that are valid in a certain context (e.g. a survey) because variables or other metadata objects are used with a slightly changed meaning or are even defined in this context, only.

A similar problem exists on the model level. The Bridge system as a statistical metadata system can reflect many of the metadata problems of national statistical offices but not all of them because some of these problems are very particular and should not be solved on a general level. Hence, context (statistical office) related model specifications must be provided for each statistical office as well.

One solution could be to define separate context-related databases for context related information. This, however, causes problems when referring to objects in the central database.

Defining hierarchical database objects solves this problem in a more elegant way. A database object defines a database owning the objects instances created for this database object. If the database object (context related database) is defined as a subordinated database object to another database object (global database) all object instances in the central data base are visible in the context database. Union sets containing all object instances are created automatically when accessing extents in a context database (e.g. variable definitions) of the global database as well as those of the context database. If an object instance is defined in the global and in the context database the instance of the context database overloads the globally stored object instance.

Object instances of a context database can be referenced from the global database level as well because context database object instances can be linked to global database instances. Thus, browsing systems, as WEB browsers, are able to provide context-related information as far as required.

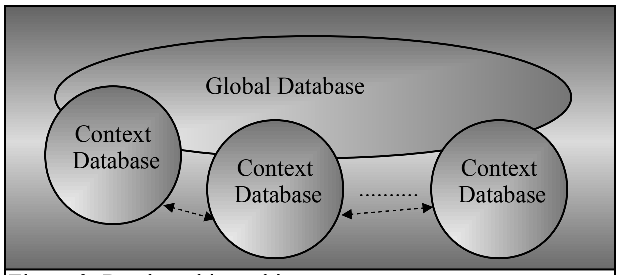


Figure 2: Database hierarchies

Context databases can be defined in the same physical database as well as in a separate physical database. This allows national statistical offices to define their own (context related) Bridge repository and to add definitions or overload global definitions within the Bridge kernel. Thus, statistical offices may add or modify structure definitions as well as other application resources (forms, document templates, data exchange definitions etc). Those modifications and extensions will survive also when a new Bridge version is released.

Hierarchical database objects are not limited to two levels. The terms global and context database define only the relationship between two databases but a global database could act as a context database related to a higher database level as well.

The concept of hierarchical database objects is not yet fully implemented in ODABA2. Context databases in the same physical database, however, are already supported with any number of levels and used in Bridge.

5. Relations between sets

In contrast to relational databases where subset relations are modelled as logical subsets (e.g. by means of SQL statements) the object-oriented model allows storing of persistent collections. Collections are usually stored as reference collections, i.e. an object instance can be element of several collections. In many cases subset relations are implicitly defined between collections.

As far as subset relations are not supported by the OODBMS it becomes a difficult task to keep these relations consistent. This is even more complicate if the collection supports one or more indexes that have to be kept consistent as well when an object instance changes it’s key attributes.

For that purpose the concept of controlled collections has been introduced into ODABA2 as one way of overcoming those consistency problems.

There are usually two types of collections considered:

- Extents
  Extents are defined as global collections on the database level. They are accessible in the context of a database (or database object) by referring to the extent name. Not all OODBMS support extents. Some consider extents as collections containing all objects of a certain type (i.e. there is exactly one extent defined for each type). Others, e.g. ODABA2, support several extents for an object type.
- Reference collections
  Reference collections are collections referenced from within an object instance.

A special case for a reference collection is a single reference, however, this can be considered as a collection as well, just with a limited cardinality.

Defining a collection as controlled collection means that the collection is controlled by another collection that acts as superset. In this case the database system guaranties the consistency of the set relation.

Local collections as subsets

In many practical cases reference collections are defined as a subset of an extent, i.e. they may refer to object instances of an assigned extent, only. When defining a reference collection as controlled collection the OODBMS ensures the consistency of the subset relation, i.e. the object instance will be removed from the reference collection as far as the instance is removed from the collection that defines a superset for the reference collection. Subset and superset relations are expressed in the model as *based on* set relation (e.g. the collection of children for a person is based on the person extent).

Theoretically based on relations could be defined between any collections, however, on the data model level no concrete reference collections are known. Typical based on relations can be defined by means of traversing paths referring to one or more relationships between objects.

Example: A hierarchical classification refers to a number of categories (classes) that are referenced in the relationship **Classification.items**. On the other hand there are a number of top-level categories that should be referenced in the relationship **Classification.top_items**. Then **top_items** obviously forms a subset of **items**. This can be expressed in the data model by defining **top_items** as based on **.items**.

The same way reference collections not directly related to the object can be defined as superset for a reference collection.

Extent hierarchies

Extent hierarchies define subset relations between extents. This is a good way to handle e.g. collections of paid and unpaid invoices for a company. This could be managed also by setting an appropriate attribute indicating a paid invoice as such and selecting all instances marked as paid or not paid. The problem in this case is that it requires usually special user actions to update the selection.

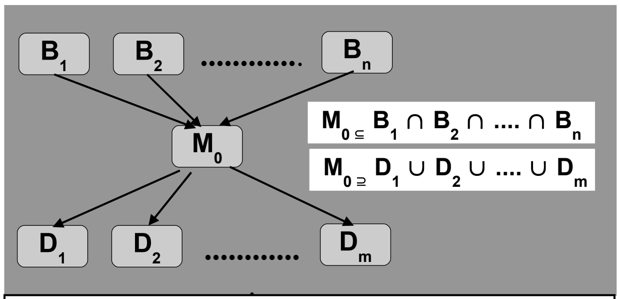
When referring to a derived extent containing the unpaid invoices instead the current state of the collection is immediately visible and it does not require any programming effort.

By means of extent hierarchies not only subset relations can be expressed. Extent hierarchies offer also the possibility to define intersection and union constraints. Thus it becomes possible to reflect simple set algebra by means of extent hierarchies.

The general rule defined between extents in a hierarchy shown in Fig. 3 can be specialised to equation, e.g. the derived extent can be defined as the exact intersection of it's basic extents:

$$M_0 = B_1 \cap B_2 \cap .... \cap B_n$$

The same way it is possible to define M_0 as union of the derived extents D_i. Moreover, extents on the same level can be defined as distinct to each other.



B_1
B_2
B_n
M_0
$M_0 \subseteq B_1 \cap B_2 \cap .... \cap B_n$
$M_0 \supseteq D_1 \cup D_2 \cup .... \cup D_m$
D_1
D_2
D_m

Figure 3: Set operations in extent hierarchies

6. Method types

Usually only methods are implemented using a certain programming or query language. Within ODABA2 there are additional types of methods provided for implementing the behaviour for a defined structure or class.

1. C++
   C++ functions are implemented in corresponding C++ classes generated from the database structure definition.
2. Visual Basic
   Visual basic functions can be implemented by means of an COM interface providing the necessary database access functionality.
3. OQL
   OQL expressions are another way to implement functions for an application.
4. Forms, dialogues
   These are template methods defining the layout of a form or dialogue. After defining the form it can be activated directly with an object instance.
5. Document templates
   These are template methods defining a document layout or the layout for a part of a document. Document templates can be used to generate complex structured documents as well as online help systems.
6. Data exchange
   Data exchange methods are special methods that are used to define data exchange with other systems. Data exchange methods are described separately in the next chapter.

Template development is a technique providing high flexibility to the user. Forms (dialogues) or documents can be easily adapted to user's requirements as far as they are provided as templates. Templates are defined in the target system (e.g. word processor) and contain links to the required data. They appear to the developer, as the user will see them after filling the template with data (what you see is what you get).

Considering templates as method implementations allows applying the same principles for templates, which are known for functions from object-oriented programming. In particular this means that templates can be overloaded in derived classes. Thus, the application is able to select the proper form e.g. for presenting data for an object instance to be displayed in a certain form.

7. Data Exchange

Data exchange with relational databases and other data sources was an urgent requirement for Bridge as well as for Belami. There is a big amount of metadata available in statistical offices, which are stored in relational databases as well as in documents or other sources. On the other hand systems as Belami and Bridge need to communicate with other systems and one way of passive communication is the data exchange. Standards as ODBC[1] provide a technical base but they do not allow defining value assignments for data exchange.

Semantic assignment

To be able to exchange data with any entity-relationship or object-oriented model requires, that value items can be addressed logically within the database system. Hence, the problem is the semantic assignment (mapping algorithms) of values between two data sources.

According to (3a) a value in an ODABA2 database is defined as:

$$\mathbf{v = V\ (i,p,t,g)}$$

Unfortunately, the instance identity (i) is not known in other systems and must be replaced by a logical identification.

a1) Data exchange is possible if there exists a unique key identifying an instance in a given collection.

This assumption allows defining data exchange for a certain collection, only. If there is a key function (1e) that uniquely identifies an instance $\mathbf{i}$ in a collection by a given keyvalue $\mathbf{v_k}$ then $\mathbf{i}$ can be substituted by $\mathbf{v_k}$ in (3a):

$$\mathbf{v = V_c(v_K,\ p,\ t,\ g)} \qquad \mathbf{(4a)}$$

Object instances may refer to instances in local collections, which are not uniquely identified by a key in the whole database but in the local collection, only. In this case a view path can be defined in the OO model traversing the referenced instances and providing a derived collection.

Instances and attributes are independent database dimensions in the relational model as well as in the object model, i.e. there is no functional dependency between instances and attributes. Moreover, it has already been stated, that time and generic type are also database dimensions in the ODABA2 object model but not in the ER model and other object-oriented models where version and generic types are usually stored as attributes.

$$\mathbf{v = V_R\ (k_R\ ,p_R\ ,a_t\ ,a_g(p))} \qquad \mathbf{(4b)}$$

Here $\mathbf{a_t}$ is the time or version attribute and $\mathbf{a_g}$ the attribute carrying the generic type for $\mathbf{p}$. $\mathbf{a_g}$ is a function of the property $\mathbf{p}$ even though in many cases only one attribute may be defined for reflecting the generic type.

a2) The version is assumed not to depend on the property, i.e. the version attribute is the same for all properties in an instance.

Then a mapping between two collections of data instances in two databases can be defined in following definition steps:

$\mathbf{c = c_R}$	(collection assignment)	**(5a)**
$\mathbf{p = p_R}$	(attribute assignment)	**(5b)**
$\mathbf{g = a_g(p_R)}$	(type assignment)	**(5c)**
$\mathbf{t = a_t}$	(version assignment)	**(5d)**
$\mathbf{k = F_K(k_R)}$	(instance assignment)	**(5e)**

That means practically that mapping data between two databases can be done quite simple by mapping collections. A collection may either refer to a database collection or to view. Within each collection attributes or properties corresponding to each other have to be defined (5b). When assigning values for properties carrying generic attributes the attribute containing the generic type has to be defined as part of the property assignment (5c). When the instance collections have several versions an appropriate version attribute has to be assigned on the instance level (5d). These mappings can be defined on the model level.

Instance assignment can not be defined on the model level because the model does not know the instance domain. Hence, instance assignment can be defined by providing a function that defines the assignment rule. In most cases, however, those instances are assigned to each other that refer to the same key values. As long as no key conversion is required this is supposed to be the default, i.e. 5a – 5d are sufficient for defining a data exchange.

After defining the data exchange on the model level data can be exchanged (exported and imported) with following external data sources:

1. Data exchange with relational databases

[1] Open database connectivity (Microsoft standard)

Data exchange with relational databases is supported in ODABA2 for all databases providing ODBC access. Moreover, simple files with a logical relational structure (fixed structure records, self-delimiter format) can be exchanged as well.

2. Exporting/importing OEL files
The OEL (object exchange format) [2] format is a special format that allows to exchange complex object instances. OEL may contain data as well as metadata and is provided as ASCII file. Properties are referenced by property names and may consist of atomic values as well as complex instances or collection of instances.

3. Data exchange with documents
Data exchange with documents is a bit more difficult because within a document there are usually no property or instance key tags. In many cases, however, documents have special paragraph formats or styles that can be associated with properties. In other cases special topic titles are used that can be associated with attributes. Keys are usually part of document content and have to be identified as such, only. Thus, importing especially statistical metadata from documents into Bridge was successful in many cases.

The advantage of defining data exchanges on the model level is that the logical definition of a data exchange is independent of the physical implementation of the data source (or target). Data imported from a SQL database can be easily exported to an ASCII file (e.g. for generating an input file for an SAS program). As long as there are systems that are not prepared for active communication data exchange is the best facility for communicating with such systems.

8. Context Classes for modeling active databases

In many cases the behaviour of an instance is depending on the environment or the context the instance appears within. Thus, it can be a difference adding a person to the children reference of another person (because the child should be younger than the person) or adding a person to the employee reference of a company. The children reference as well as the employee reference defines the context determining the behaviour of the object in this case. In general we can say that the behaviour of objects depends on the context they are acting within.

In contrast to ordinary object classes describing the general behaviour of an object a context class defines the behaviour of an object in a certain situation within the application. Thus, context objects contain also information about the access path, e.g. the context object for a person reference knows that the person instance has been activated e.g. as an employee of a company.

Within ODABA2 databases four general types of context classes are defined:

- Database context
- Database object context
- Structure context
- Property context

That means practically that each resource defining the database model can be associated with a context class. But context classes itself are not a big step forward as long as there are no rules defining when functions implemented in context classed are called.

Hence, a context class does not only define the behaviour of a certain context but the reaction on special events as well. There are different views to what an event is. We consider events as relevant state transitions (not as actions causing a state transition). In the dynamical model of ODABA2 reactions can be defined that associate an event with a certain action. Then the defined action is executed whenever the event is raised. Because execution of an action may cause other events a chain of reactions is possible just as the consequence of a little event.

Events

There are several events defined on the system level (system events) defining state transitions of a context object within an application. Typical system events are read and update events indicating that a structure or property instance has been read or updated. But also model specific events (model events) can be defined. Model events are defined as a set of potential state transitions for the object the context is based on (property, structure). If **S** is the set of possible states for an instance then the set of possible state transitions can be defined as the product set of possible states:

$$\mathbf{T = S \times S} \qquad \textbf{(6a)}$$

An event is now defined in general as a subset of **T**:

$$\mathbf{E = \{ e \mid e \subseteq T \}} \qquad \textbf{(6b)}$$

Special (and most typical) events are defined as independent set of pre-states $\mathbf{S_b}$ (before transition takes place) and post-states $\mathbf{S_a}$ (after transition took place):

$$\mathbf{E_0 = \{ e \mid e = S_b \times S_a \} \subseteq E} \qquad \textbf{(6c)}$$

A typical way to define a set of relevant object states $\mathbf{S_0}$ as a subset of **S** is to define a condition (e.g. by means of logical expressions or functions):

$$\mathbf{S_0 = \{ s \in S \mid f_0 (s) = true \}} \qquad \textbf{(6d)}$$

Now we can define a number of events just by defining a pre-condition $\mathbf{f_b}$ and a post-condition $\mathbf{f_a}$ for the object the context is based on. Even though this allows only defining events in $\mathbf{E_0}$ it provides quite good facilities for handling model events.

Actions

System and model events can be associated with actions now. An action defines a more general type of behaviour than function or expression. In fact, there might be a number of action types supported beyond functions and expressions. Within ODABA2 e.g. dialogue or menu actions, document actions and other types of actions can called as well.

The association between events and actions is a relevant part of designing the dynamical model. However, most of system events are associated with actions automatically by naming conventions.

Reactions

Reactions are defined as actions that are executed as consequence of an event. In many cases the object instance changing the state is the same as the one that is reacting. That, however, is not necessarily the case. In real life usually an observing object reacts on events caused by other objects.

Therefore, defining a reaction must include the possibility to associate an event generated by an object1 with an action for another object2. That is quite difficult on the model level because object instances are not known here. The situation can be improved a little bit by referring to defined relationships between objects. As far as the event generating object and the objects that should react on the event are connected directly or indirectly by relationships a property path can be defined from the event generating object to a set of reacting objects.

Within an application it is usually no problem to signal an event because (transient) application objects are active and ready to react. Objects in an object-oriented database, however, are usually inactive (not loaded) when an event is signalled, i.e. they will not take notice of the event. But the event generating object is active in any case (otherwise is could not change it's state) and the OODBMS can activate the associated objects so that they can react on the event.

Conclusions

Applications design tents to become more and more a process of implementing pieces in complex structures. More and more rules can be defined simply on the data model level covering a quite complex functionality. Context classes and reactions for database objects are defined as business rules and can be considered as an integral part of the database model and it's behaviour. Most of these activities do not depend on each other.

This is a quite efficient and comfortable way for developing applications. Instead of solving one big problem a number of small problems are solved separately. The efficiency of these technologies becomes obvious when considering the development resources for the two example projects. But even more interesting is, that developers can be replaced because it is no problem for a developer familiar with the technology to locate a problem in an application that has been developed by another one.

On the other hand this paper has shown that there are still a number of open problems that have to be solved in the near future. And another important aspect becomes obviously: How can we guaranty the consistency of a database model and its business rules or causal relations defined in the dynamical model?

Most systems have consistency checks for the data model and it is also no problem to define consistency rules for enhanced database models. But it happens quite often that rules defined in the dynamical model conflict with each other. And so far we are not able to define general ways to avoid such conflicts. This will be one of the big tasks for the future.

REFERENCES

[1] Heuer A.: *Objektorientierte Datenbanken* (German), Addison-Wesley, Bonn, 1998

[2] Karge R.: *Real Objects* (German), Addison Wesley, Bonn, 1996

[3] *Bridge – Object Architecture, Bridge – User's Guide,*
IMIM work papers[2], 1998

[4] Karge R.: *Integrated Metadata Systems within Statistical Offices*, Conference on Scientific and Statistical Database Management, Capri, 1998

[2] IMIM working papers are available at http://imim.scb.se