to appear in the Journal of Systems Integration, Vol.4 No.4, 1994, Kluwer Academic Publisher.

Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept

Wolfgang Klas, Gisela Fischer, Karl Aberer
GMD–IPSI, Darmstadt
{klas, maus, aberer}@darmstadt.gmd.de

Abstract

This paper presents a specific approach of integrating a relational database system into a federated database system. The underlying database integration process consist of three steps: first, the external database systems have to be connected to the integrated database system environment and the external data models have to be mapped into a canonical data model. This step is often called syntactic transformation including structural enrichment and leads to component schemas for each external DBMS. Second, the resulting schemas from the first step are used to construct export schemas which are then integrated into global, individual schemas or views, in the third step. In this paper we focus on the first step for relational databases, i.e., the connection of a relational database system and the mapping of the relational model into a canonical data model. We take POSTGRES as the relational database system and the object-oriented federated database system VODAK as the integration platform which provides the open, object-oriented data model as the canonical data model for the integration. We show different variations of mapping the relational model. By exploiting the metaclass concept provided by VML, the modelling language of VODAK, we show how to tailor VML such that the canonical data model meets the requirements of integrating POSTGRES into the global database system VODAK in an efficient way.

Keywords: database integration, semantic enrichment, object-oriented modelling, metaclasses

1 Introduction

Electronic information management systems are complex human centered activities which produce and consume the most diverse kinds of information. Today many of these activities are supported by autonomous systems which employ different data management facilities with heterogeneous data models, e.g., a relational model, a hierarchical model, an object-oriented data model, or – specifically valid for public databases – some dedicated file system with specialized retrieval and presentation functionality. In addition, the information is represented at different levels of detail, with mutual inconsistencies in structure, naming, scaling, and behavior, whereby much of this behavior is hidden in the implementation of the autonomous systems.

However, more and more applications like those in the field of cooperative authoring and publishing or telecommunication services and administration definitely need *integrated* access to their underlying, autonomous, heterogeneous information bases. It is needed because these applications demand integrated processing due to consistent management of complex interrelated data as well as integrated exchange of information produced and consumed by the many participants in an application. Such applications should provide for a kind of individual, integrated, global views onto the underlying resources.

There exist several approaches and projects which address *interoperability* or *integration* of information bases. [25], [17], [1], and [3] give good overviews and present fundamental concepts including the terminology of the different approaches, e.g., multidatabase systems, multidatabase languages, and federated database systems. MRDSM [15], OMNIBASE [21], and CALIDA [7] are projects which realized the integration of databases by multidatabase languages, but which require a kind of sophisticated user because the user still needs information about the distribution of data, about how to resolve semantic ambiguities and other typical well-known problems which arise when integrating heterogeneous databases. SIRIUS-DELTA [16], DDTS [2], Mermaid [26], Multibase [14], and our approach taken in KODIM can be mentioned as projects which follow the federated database approach. The tools and techniques developed in KODIM ([23], [8], [4], [5], [9], [19], [18]) for *semantic integration* assist *incremental integration* driven by actual information requests of end users and the *dynamic maintenance* of integrated schemas driven by external schema evolution. This approach tries to meet the requirements of realistic situations with a big number of external information bases (which – due to their autonomy – are subject to only locally controlled constant change) in which completely integrated views valid for all users can hardly be achieved with reasonable effort. In addition, many available information sources (e.g., online-databases) do not even provide fine grained, explicitly structured data like relational databases do. Thus, in KODIM we also develop tools for the *structural enrichment* of data from external information sources which do not provide any kind of schema [6].

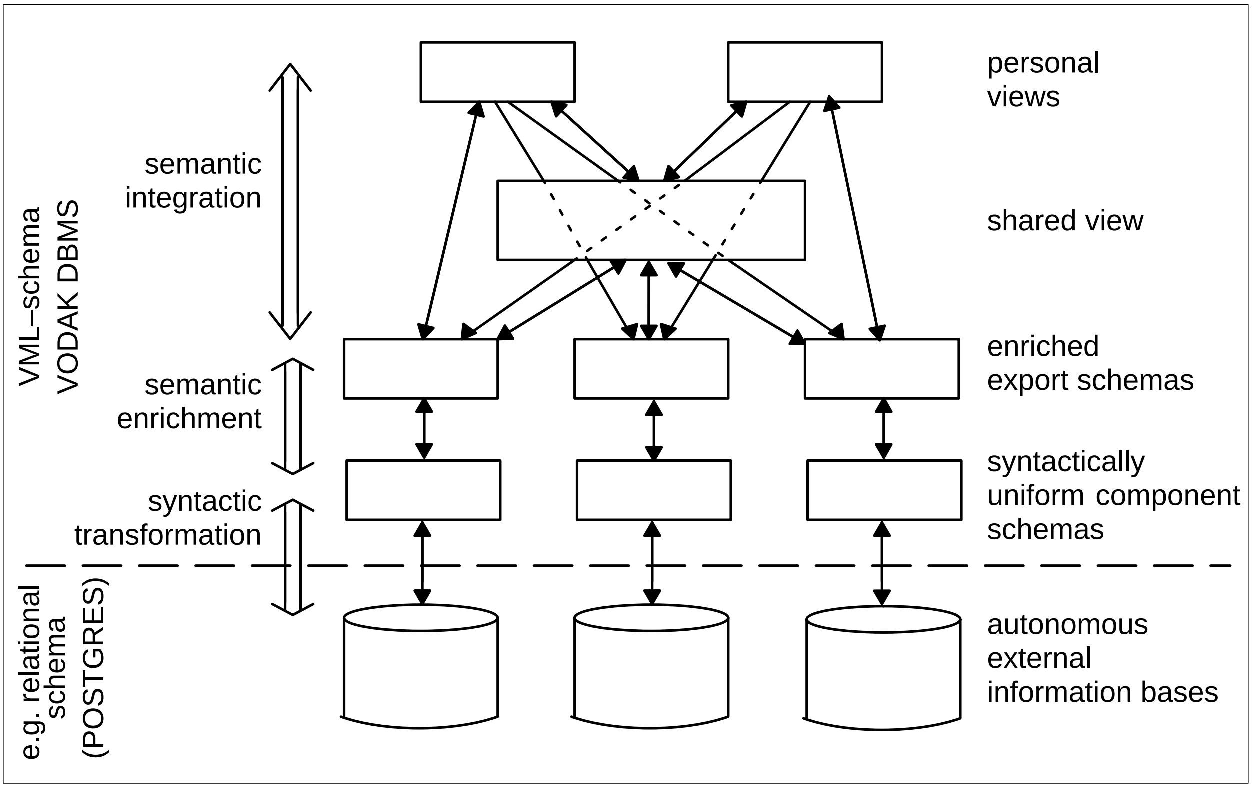
Database integration steps in KODIM can be partitioned into two conceptual layers:

(i) At a base layer, heterogeneous data models have to be mapped to a uniform data model (*syntactic transformation* phase). This requires the translation of manipulation languages and the transformation of diverse data formats as well as the connection of external database management systems or other systems providing external data.

(ii) On top of this bottom layer, i.e., on the basis of the uniform data model, *implicit structure & semantics* have to be made explicit, *inconsistencies* in *structure, naming*, and *scaling* have to be overcome, and *semantic interrelationships* between data have to be acquired in order to establish integrated views onto the external resources (*semantic enrichment and semantic integration steps*).

Figure 1 shows the variety of transformations which data from diverse resources have to undergo in order to be integrated. KODIM uses the data model of the open, object-oriented database system VODAK as the canonical data model, to which the external schemas are mapped to. To use an object-oriented data model as the canonical model is widely recognized to be a very promising choice for easier representation of external data models as well as for schema integration purposes (see [25], [23], [8], and [22]).

The *syntactic transformation* step provides for a syntactically uniform VODAK interface to the external information bases, describing their database schemas (including constraints), retrieval & manipu-



igure 1: Integrated views on heterogeneous information bases

lation capabilities, and their file formats. The transformation is modularized by means of the object-oriented VODAK Modelling Language (VML), i.e., for all imported data (schemas as well as instances) object types and classes are established, supporting the capabilities of external information bases in a uniform language.

By syntactic transformation, external information bases are accessible according to a uniform data model, but they are not interrelated semantically. Two additional mapping steps are required to interrelate and merge these data semantically:

Semantic enrichment makes implicit structure and semantics explicit and associates additional behavior, which is hidden in local application programs or even worse in informal local conventions.

Semantic integration is needed to combine several schemas. Structural and semantic differences in representation, conflicts in naming and scaling have to be resolved, correspondences between objects have to be identified, and appropriate foci have to be specified in order to establish an (or a couple of) integrated user view(s).

In this paper we focus on the problem of integrating relational database systems and how to get (enriched) export schemas. We discuss several alternatives and their limitations for the syntactic transformation step and the impact of semantic enrichment. We will not discuss the semantic integration phase in this paper. Details about the techniques employed for the final semantic integration steps are given in [23], [20], [24], [4], [5], [19], and [18].

The rest of the paper is organized as follows: Chapter 2 gives a general description of the characteristics and problems of different mappings of a relational schema into an object-oriented database model like the data model of VODAK. Chapter 3 gives a brief outline of the concepts of the data model and of VML as far as needed to show the realization of the mappings in Chapter 4 by exploiting specific modelling features provided by VML. Chapter 5 concludes the paper and gives some hints to further improvements.

2 Characteristics of Mappings

In order to characterize mappings from relational schemas and their corresponding databases to object-oriented schemas and their corresponding databases we first have to determine the correspondences between the concepts of both data models. Then we can describe the general characteristics of a variety of alternative mappings, including different forms of semantic enrichment.

2.1 Correspondences between the Data Models

The basic concepts of the relational model are *relations*, *tuples* and *attributes*. A relation can be thought of as a table with columns of different types. These columns are called the attributes, whereas the rows of a table, i.e. the actual contents, are the tuples of a relation. In the standard relational model the attribute types are restricted to primitive data types, e.g. String, Integer etc. The structure of the tuples is then determined through the definition of the relation expressed in the data definition language (DDL). Relations and tuples are manipulated (i.e. created/appended, changed, deleted) using the data manipulation language (DML).

In contrast, the most relevant concept of an object-oriented data model in this framework is the concept of *objects*, whereas each object is an instance of a class and may be a class itself. The structure and behavior of an object is determined through a set of property and method definitions specified with the object's class. The type of a property can be any primitive (i.e. String, Integer etc. and also object identifiers) or complex datatype (i.e. Array, List, Set etc.). Access to an object's properties is only allowed through the interface, i.e. the set of methods defined with the object's class. The data manipulation language provides for sending messages to objects in order to make them execute certain method implementations. This ensures object encapsulation and, as a logical consequence, controlled access to data stored in objects.

We concentrate on the structural aspects of the object-oriented data model when defining a mapping from relational to object-oriented schemas since the standard relational models do not provide methods or functions. Then there is a natural correspondence between the following concepts:

- relations and classes
- attributes and properties
- tuples and instances

First the relation definitions of a relational schema have to be translated to class definitions of an object-oriented schema. For example, this can be done in a straightforward way according to the above correspondences. However, the straightforward translation is not necessarily the desirable one since it does not exploit the full expressive power of the object-oriented data model. We will therefore discuss more refined mappings in subsections 2.2.3 and 2.2.4.

For mapping the data in the relational model to data in the object-oriented model things are more complicated. The concept of *object-identity* plays an important role in an object-oriented data model. When using an object-oriented data model as the canonical, global data model, information stored in external database systems is represented as objects in the global database system. These objects possess object identifiers which consequently also identify data stored in the external system. Hence, there must be some mapping from global object identifiers to appropriate external "object identifiers". In case of relational data base systems we miss the concept of object identifiers as conceptual data units are identified by key values. Hence, the system has to maintain somehow a mapping of key values to object identifiers.

At first glance, the mapping between an external identification mechanism and object identity looks quite simple. But in fact, as we will see later in subsection 2.3 one has to take into account different kinds of these mappings depending on the “quality” of the external identification mechanism.

2.2 Mappings from Relational to Object-Oriented Schemas

In the following we will introduce several kinds of mappings from relational to object-oriented databases. We start with the straightforward one already indicated in the previous section by the natural correspondences between the two data models. However, as we will point out the straightforward mapping may in certain cases lead to an unnatural object-oriented modeling. Therefore we will introduce other types of mappings which exploit the additional expressive power of the object-oriented data model. We analyze which support has to be given in the object-oriented world in order to make the mapping efficient.

2.2.1 Straightforward Mapping

A relation is mapped to one class and vice versa. An instance of a class corresponds to exactly one tuple in the relation which corresponds to the class. Each attribute of a relation is mapped to a corresponding property of a class. A relational schema $S=\{R_1,...,R_N\}$, $R_i(a_{i,1}, a_{i,2}, ..., a_{i,ki})$, $1 \le i \le N$, may be translated automatically to an object-oriented schema according to the following rule:

> For each relation R_i in S a class Cl_i with properties $c_{i,j}$, $1 \le j \le k_i$, is defined,
> where property $c_{i,j}$ corresponds to attribute $a_{i,j}$ of relation R_i.

The values of the properties $c_{i,j}$ in the object-oriented database are derived from the corresponding attributes $a_{i,j}$ in the relational database. Therefore access methods to the values stored in $a_{i,j}$ have to be provided. For simplicity we consider for the moment only reading methods. Such a method call which enables the access to a relational database is of the form *get(a):v*, where *a* is the name of an attribute of this relation and *v* is the value of *a* in the tuple which corresponds to the object representing this tuple. Remember that in an object-oriented system each method call has to be sent to an object, which in this case is the instance of a class representing a specific tuple in the corresponding relation.

This mapping type performs only the syntactic transformation step and does not contribute to the semantic enrichment of an external schema. It produces an unnatural object-oriented modelling since relationships in the relational model (which are represented as relations) are mapped to classes and not to references between objects. For instance, a straightforward mapping of the relations R_1 ($\underline{A}$ B),

R_2 ($\underline{A}$ $\underline{C}$) and R_3 ($\underline{C}$ D) yields the classes Cl_1 ($\underline{A}$ B), Cl_2 ($\underline{A}$ $\underline{C}$) and Cl_3 ($\underline{C}$ D)[1]. Instead of mapping R_2 to a class it is more natural to represent it by additional properties of Cl_1 and Cl_3 which then hold references to the corresponding objects of the respective other class (see also Example 4). Therefore in the following we will introduce more sophisticated kinds of mappings.

2.2.2 Comparison between the Expressive Power of the Object-Oriented and the Relational Data Model

Let us recall some characteristics of the relational data model. Data is stored in tables with attributes which can only be of a *primitive datatype* (1NF). Relationships between data are expressed through the values of common attributes in different relations. It is known that this leads to different anomalies if the relations are not designed carefully, i.e. they are not in one of the well-known normal forms (3NF, BCNF, etc.). The basic mechanism to transform an arbitrary relational schema into a normalized schema is to *decompose* the relations. Both the restriction to primitive data types and the decomposition of relations lead to a situation where related data reside in several relations, a fact that leads to frequent computations of joins between relations.

In contrast to this the (structural) object-oriented data model allows a modelling which is much closer to the structure of the "real world". The additional expressive power emerges mainly from two sources: first, there is no restriction on the datatypes which properties can have, e.g. set and tuple constructors may be used. Second, the concept of object identifiers allows to express relationships between data explicitly, e.g. properties may hold references to other objects. While in the relational model the maintenance of referential integrity constraints requires to guarantee that for each foreign key value there exists a corresponding primary key value, in object-oriented models it is necessary to ensure that properties hold only references to existing objects.

We summarize the differences between the data models in the following table in which we relate features of the object-oriented model to the corresponding restrictions in the relational model.

Object-Oriented Data Model	Relational Data Model
complex values	1NF
references	3NF, BCNF

We will now study different cases how mappings can invert the decompositions of relations discussed above and can be used to construct more natural schemas in the object-oriented data model. These mappings automatically lead to semantic enrichments of the relational schemas.

1. We extend the standard notation for relations, denoting a relation by $R(\underline{A}\ B\ ...)$, where $A, B, ...$ are attribute sets, e.g. $A=\{a_1,...,a_{nA}\}$, and the primary key of R is underlined, to classes in the following way: $Cl(A\ B\ ...)$ is the class $Cl(a_1 : T_1, ... , a_{nC}:T_{nC})$, where T_j are primitive property types and a_i are property names.

2.2.3 Reconstructing Complex Values

Case 1:
Let R_1 (<u>A</u> B) be a relation with primary key $A = \{a_1, ..., a_n\}$, and let R_2 (<u>A</u> <u>C</u> D) be a relation which has A as foreign key. Assume that relation R_2 is not needed in any other context (e.g. there are no other foreign keys in R_2). Disregarding the straightforward approach we have the possibility to combine these two relations in one class with the following structure

$$Cl(A\ \ B,\ t{:}\ \{[C\ D]\})^2.$$

The type of the third property t in Cl is set-valued because several tuples in R_2 can correspond to one tuple in R_1 having the same value for A. The elements of this set are tuples in order to maintain the dependency between C and D expressed in R_2.

Example 1:
Let R_1 be the relation

conference(<u>conf_id</u>, title, topics, location, duration, language)

and R_2 be

session(<u>conf_id, sess_id</u>, duration, subject, chair_name, chair_affil).

The class which results from the mapping can then be defined as

Conference(conf_id, title, topics, location, duration, language,
sessions:{[sess_id, duration, subject, chair_name, chair_affil]}) ❑

We call relation R_1 the *base relation* in the mapping, since it contains the primary key, and R_2 is called the *dependent relation*. More generally, we can consider cases where there are several dependent relations. This can happen in two ways: either there are other relations containing the primary key of the base relation as (the only) foreign key or there are relations containing the primary key of a dependent relation as an additional foreign key. In the first case we get additional set-valued properties of the kind introduces in the example, in the second case we can construct deeper nestings of the complex values. This is illustrated in the following case.

Case 2:
Let R_1 (<u>A</u> B) and R_2 (<u>A</u> <u>C</u> D) be defined as before. Let R_3 (A C E) be a relation with A as the primary keysand C of R_2 as foreign keys. Again assume that R_2 and R_3 are not needed in any other context. Now we have the possibility to combine these three relations in one class with the following structure :

$$Cl(A\ \ B,\ s{:}\ \{[C\ D,\ t{:}\ \{[E]\}]\}).$$

---

2. In $Cl(A, a{:}T)$ a denotes an additional property resulting from the mapping, and for tuple types we write $[A]$ for $[a_1 : T_1, ... , a_n{:}T_n]$.

Example 2:
Let R_1 and R_2 be the relations *conference* and *session* as in the previous example. Let R_3 be
lectures(<u>conf_id</u>, <u>sess_id</u>, <u>lecturer_name</u>, lecturer_affil, topic)

The class *Conference* can then be defined as
Conference(conf_id, title, topics, location, duration, language,
sessions:{[sess_id, duration, subject, chair_name, chair_affil,
lectures: {[lecturer_name, lecturer_affil, topic]}]}) ❑

The first obvious advantage is that by restructuring the data in this way we can get closer to the structure of the “real world”. Moreover there is another, less obvious, advantage which becomes clear when we analyze what kind of access to the relational database can be provided in supporting this mapping of a relational schema. Assume we only support access methods *get(a):v* as introduced for the straightforward mapping. Then first all the relations are mapped to classes in the straightforward way; in a next step we build complex-structured classes upon them. Then one has to retrieve the values for the complex classes by value-based joins inside the object-oriented database management system. This will in general be much less efficient than performing the corresponding (optimized) joins in the relational database management system.

These observations lead to the following conclusion: there is a need for more complex access methods to the relational database, e.g. method calls of the form

get(*R, {keyattr}, [attr]): {[attrval]}*

where *R* is a dependent relation, *{keyattr}* is the set of primary key attributes of the base relation and *[attr]* is a tuple of attributes whose values should be retrieved. In the case of Example 1 the method *get*(R_2*, A, [C D]): {[C D]}* is sent to an instance of class *Cl*, which corresponds to one tuple of R_1, and returns the appropriate set of tuples in R_2. Now the implementation of this access method can arbitrarily use the mechanisms of the relational DBMS, e.g. compute a value-based join between R_1 and R_2 efficiently.

A more general access method, which allows a nesting of arbitrary depth is of the form

get(*<[R, {keyattr}]>, [attr]): {[attrval]}*

where instead of one relation with the corresponding key attributes now lists of relation names and key attributes are given as arguments.

2.2.4 Substituting Value-Based by Reference-Based Relationships

Case 3:
Let R_1 ($\underline{A}$ B) be a relation with primary key A, and let R_2 ($\underline{A}$ $\underline{C}$ D) be a relation which has A as foreign key. Assume now that relation R_2 is needed in another context (e.g. there is another foreign key in R_2). Therefore we cannot map both relations to a single class. Despite the fact that, in this case, we map each relation to a different class we have additional possibilities to enrich the structure of the resulting classes. Some of the alternatives are shown in the following:

- Cl_1*(A B, r: {ref* Cl_2*})*, Cl_2*(A C D)*
- Cl_1*(A B)*, Cl_2*(A C D, r: ref* Cl_1*)*
- Cl_1*(B, r: {ref* Cl_2*})*, Cl_2*(A C D, s: ref* Cl_1*)*
- Cl_1*(B, r: {ref* Cl_2*})*, Cl_2*(C D, s: ref* Cl_1*)*

where *ref Cl* denotes a reference to an instance of class *Cl* which is realized on the basis of object identifiers. This list is not exhaustive and the decision which mapping is preferred depends on which access paths are needed more often and which properties are needed in which classes. For example, in the last case the attributes *A* are dropped assuming that they were only needed for establishing the relationship between R_1 and R_2.

Example 3:
Let R_1 and R_2 again be the relations *conference* and *session* as given in Example 1. Then we can map these relations to the classes *Conference* and *Session* as follows:

- *Conference(conf_id, title, topics, location, duration, language, sessions:{Session})*
  Session(conf_id, sess_id, duration, subject, chair_name, chair_affil)
- *Conference(conf_id, title, topics, location, duration, language)*
  Session(conf_id, sess_id, duration, subject, chair_name, chair_affil, conference: Conference)
- *Conference(title, topics, location, duration, language, sessions:{Session})*
  Session(conf_id, sess_id, duration, subject, chair_name, chair_affil, conference: Conference)
- *Conference(title, topics, location, duration, language, sessions:{Session})*
  Session(sess_id, duration, subject, chair_name, chair_affil, conference: Conference)

❏

Again we call relation R_1 the base relation in the mapping, since it contains the primary key, and R_2 is called the dependent relation.By the same arguments as in the previous section we now have to provide a new type of access methods in order to exploit the relational DBMS. In the simplest case these methods are of the form

$$get(R_d, \{keyattr\}): \{ref\ Cl_d\}\ \ (get(R_b, \{keyattr\}): ref\ Cl_b\ respectively\ )$$

where R_d is the dependent relation, *{keyattr}* is the set of primary key attributes of the base relation R_b and *ref* Cl_d *(ref* Cl_b*)* is a reference to an instance of the class Cl_d *(*Cl_b*)* which corresponds to the dependent (base) relation. The receiver of a call of this method is an instance of $Cl_b(Cl_d)$. For simplicity we assume that the key attributes which are used to join the base and the dependent relation have the same names in both relations. The following case shows a more complicated situation:

Case 4:
Let R_1 ($\underline{A}$ B), R_2 ($\underline{A}$ $\underline{C}$) and R_3 ($\underline{C}$ D) be given. In this case R_2 serves only to represent an n:m relationship between R_1 and R_3 and therefore can be dissolved in the following way

$$Cl_1(A\ B,\ r: \{ref\ Cl_3\}),\ \ Cl_3(C\ D,\ s: \{ref\ Cl_1\})$$

where the classes Cl_1 and Cl_3 correspond to the relations R_1 and R_3.

Example 4:
Let R_1 be the relation conference as defined in Example 1. Let R_2 be

reservation(<u>conf_id</u>, <u>hotel_name</u>)

and R_3 be

hotel(<u>name</u>, mail_addr, city, state, country, phone_no, fax_no)

The relation *reservation* can be dissolved by defining the classes *Conference* and *Hotel* as

Conference(conf_id, title, topics, location, duration, language, hotels: {Hotel})
Hotel(name, mail_addr, city, state, country, phone_no, fax_no, conferences: {Conference})

❏

To accomplish the mapping of the previous example efficiently we have to provide more general access methods, i.e. methods which allow to perform join sequences over several relations. These methods are of the form

$$get(<[R, \{keyattr\}]>): \{ref\ Cl\}$$

where instead of one relation with the corresponding key attributes now lists of relation names and key attributes are given as arguments.

2.2.5 Schema Restructuring

Of course, when mapping relational schemas to object-oriented schemas other restructurings can be performed than those provided by the mappings discussed above. These can exploit additional semantic knowledge of the schema beyond the knowledge about key attributes. However, these restructurings are of a different nature since they are not only inversions of normalizing processes of relational schemas which are necessary due to the restrictions of the relational data model.

For example, in [27] an approach for translating relational schemas into object-oriented schemas is discussed, where some interesting translations are made in the case where inclusion constraints are given for the relational schemas. However, only the problems of view integration and not of database system integration are considered. Some of the mappings that we proposed in subsection 2.2 can also be found there. But the authors do not consider the possibility of substituting value-based by reference-based relationships as discussed in subsection 2.2.4, and thus arrive at schemas containing relationship objects, something we try to avoid where possible by our approach.

2.3 Mapping Populations – the Identification Problem

In the previous subsection we considered the different types of mappings between a relational and an object-oriented schema. In addition to this mapping of schemas we need to describe the mapping of concrete tuples to concrete objects.

From the previous discussions about which kinds of mappings we consider we make the following observation: from a relational schema $S=\{R_1, R_2, ...R_n\}$ a distinguished subset *S'* is chosen. For each relation *R* in *S'* a corresponding class *Cl* is defined. It is clear that the extension of a class *Cl* corresponds one-to-one to the extensions of *R*, i.e. for each tuple in *R* an object in *Cl* is generated. The information stored in the relations $R \in S \setminus S'$ is then accessed through the complex access methods introduced in 2.2.3, i.e. for the tuples in these relations no individual objects are generated.

In an object-oriented data model objects are identified by their unique object identifiers. Object identity is an important concept in order to construct complex objects or to provide object references. Furthermore, object identity is very important to identify objects beyond session boundaries[3].

In a relational database the identification mechanism is based on key values. The mapping of key values to object identifiers may become very complex and may impose restrictions on the usage of the global objects because key values may change. Furthermore it is not sufficient only to consider the key values alone but also the relation involved has to be used as a parameter in the mapping. In POSTGRES, for which we later provide the actual implementation of the mapping functions, there exists the concept of tuple identifiers. These are realized as an additional attribute in each relation and do not change throughout the lifetime of a tuple. In this case we can map tuple and relation identifiers to ob-

3. This is achieved in only those systems which allow to explicitly ask for an object identifier which can be stored in one session and retrieved in another session.

ject identifiers. If a relational system does not provide tuple identifiers, we have to map the values of the ”real” key attributes and the relation identifiers to object identifiers. Objects with corresponding identifiers have to be generated for all tuples in those relations for which a corresponding class is generated.

Once the mapping from key values resp. tuple identifiers to object identifiers is fixed we have to define a strategy how the object-oriented database is populated.

(1) The instances of the classes are generated when the database is initialized. Adding or deleting tuples in the relational database has to be propagated to the object-oriented view.

(2) The instances of the classes are generated on demand. This requires additional access methods which can be sent to classes and trigger queries in the relational database such that the result of the query leads to the generation of instances in the object-oriented database.

In both cases changes of key attributes in the relational database have to be propagated to the object-oriented database if the object identifiers are derived from key values.

Additionally to the question when instances are generated we have to decide how the attributes of the relations are represented within the instances. Obviously this has an important impact on duplicating data and keeping the object database consistent with the external relational database. Again we can distinguish several possibilities.

(1) All attribute values are stored in property values when the database is initialized. This demands that any changes in the attribute values lead to updates in the object-oriented database. This may not only affect values but also references.

(2) Attribute values are stored in property values on demand. Although this reduces the overhead in the initialization phase this leads to similar problems as in (1).

(3) Attribute values are always accessed via the access methods or in other words attribute values are not stored in the object-oriented database. This avoids the difficulties in dealing with updates of the relational database. It means that the instances of the classes in the object-oriented database have no own state expressed by properties.

Note that updates in the object-oriented database can always be easily propagated to the relational database since the correspondence between objects and relational data is known in the object-oriented database.

3 The Canonical Target Data Model of VODAK

In our project, the data model of VODAK serves as the canonical target model in which the syntactically uniform component schemas are defined. In the following we will give a brief outline of this model as far as we need it to show how the VML modelling features are used to realize the mapping of a relational schema. For a more detailed description of the model see [13].

3.1 Object Types, Data Types, and Inheritance

Object Types

The structure and the procedural behavior of objects are defined through abstract data types which we call *object types*. Every object type definition is identified by a unique type identifier. The definition of an object type consists of sets of property definitions and method definitions. Every property definition consists of the name and the type of the property. Every method definition is represented by a method signature and an implementation of the method.

Properties can be defined either as public or as private properties. Private properties are only available (accessible) within the scope of the object type which defines them. If properties are declared to be public they are available (accessible) from outside of the object type which defines them by specific access methods which are automatically provided for public properties by the DBMS. Methods can also be defined to be private or public, in analogy to public and private properties. Private methods usually serve as auxiliary methods for the implementation of other methods.

Data Types

The types used for the definition of properties, formal parameters, and results of methods are either primitive types or complex types which can be built from predefined primitive types and object type identifiers by applying type constructors. We call such types *data types* as the values of these types are not stored as separate objects in the database, which could be identified by an object identifier. Similar to an object type, a data type may be identified by a unique identifier.

Object Type Inheritance

In the VODAK data model object types can be derived from other object types by means of specialization. More specialized object types, called *subtypes*, are built through specifying how they differ in their property and method definitions from already defined more general ones, called *supertypes*.

An object type T that is defined as a subtype of another object type T_1, specified through a *subtypeOf* clause, imports the property definitions of its supertype. These are merged with the property definitions given for the object type T itself. If a property (identified via its name) is defined twice, i.e., it is defined at object type T and at a supertype T_1, the specification of type T overrides the one of type T_1. The *subtypeOf* relationship between T and T_1 does not induce any relationship between objects of type T and objects of type T_1.

3.2 Objects, Classes, and Metaclasses

Objects

Objects are representations of material or immaterial real-world entities, or of abstract concepts, e.g., data model primitives. Objects are identified through unique object identifiers. The concrete state of an object identified can conceptually be represented as a set of *factual* properties, i.e., pairs of property names and values. Possible states of an object, i.e., its definitional properties and the kind of property values allowed to be stored with the properties, and corresponding methods are specified through an associated object type.

Classes and their Instances

Every object in the system is defined as an instance of exactly one class that contains all objects of "equal" real world meaning. The structural properties and methods of these objects are defined through an object type (the *instance-type*) associated with the class.

In the data model of VODAK a class is *not* a type, but an object itself. A class serves as the object which (a) collects all its instances, and (b) has associated an object type as the instance-type of the class.

Metaclasses

As classes are objects, they are instances of other classes, called metaclasses. Hence, for a class, three levels may be distinguished: the instance level, constituted by the instances of the class, the class level constituted by the class object itself, and the metaclass level, constituted by the class's metaclass.

Common properties of the instances of a metaclass (which serve as classes) are defined by its instance-type. But, in addition, common properties of instances of several classes may be defined once at the meta level, i.e., at the common metaclass of these classes by an *instance-instance-type*. Additional individual properties and methods may be added at the (meta)class level by associating an object type, called *own-type*, with a (meta)class.

Determining the Structure and Behavior of Objects

Roughly, the structure and the behavior of any object is determined through

- the own-type associated with the object (if it is a (meta)class),
- the instance-type associated with the object's class, and
- the instance-instance-type associated with the object's metaclass.

(Notice, that these types may be defined as subtypes of other types, and not only the properties and methods specified directly with these types have to be considered, but also the properties and methods inherited from the supertypes of these types).

Figure 2 shows how a metaclass *M* can be used to define common structure and behavior for classes and their instances. Classes Cl_1 and Cl_2 are guaranteed to behave in the same way according to the definitions given with the instance-type associated with the metaclass *M*. In general, instances of Cl_1 and Cl_2 have different interfaces because of the different definitions specified with the instance-types associated with Cl_1 and Cl_2. But, these interfaces consist of a common part which correspond to the definitions given with the instance-instance-type of the metaclass *M*. The initial object type and class structure is formed by a few predefined metaclasses (including the metaclass *Metaclass*) and object types, but will not be discussed here in detail.

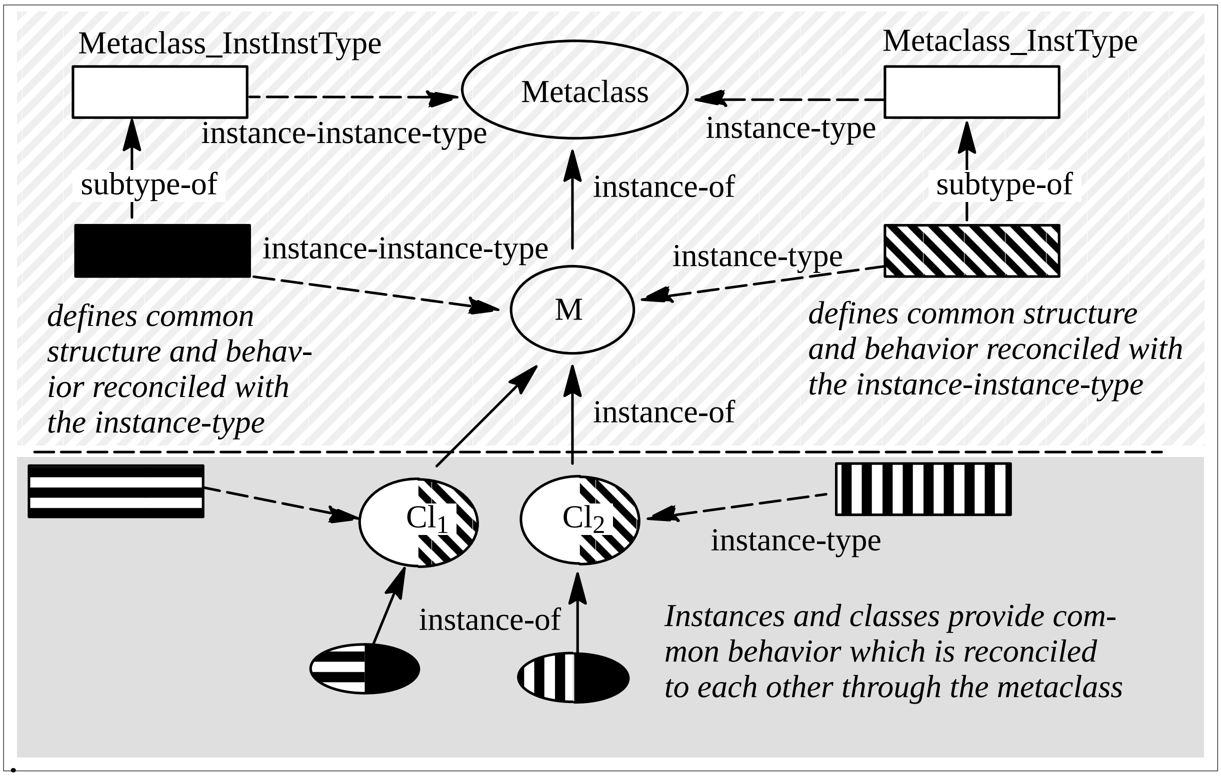


Figure 2: Metaclasses determine structure and behavior of classes *and* their instances

Classes and object type definitions which reflect specific application semantics constitute the application layer (see the area marked with the pattern). The metaclasses and the object types used for their definition constitute the meta layer (see the area marked with the pattern).

3.3 Message Passing and Method Execution

The properties of an object can be accessed (read or manipulated) only through the execution of methods defined for the object. The execution of a method is invoked by sending a message *obj*→*m* (*args*) to the object.

The semantics of sending a message *obj*→*m* (*args*) to an object are as follows:

- If the method *m* is defined for the object *obj*, the code specified for *m* is executed using the actual parameters *args*.
- If the method *m* is not defined for the object *obj*, the message *obj*→*NoMethod* (*m*, *args*) is executed, where the method *m* and its arguments *args* are passed as arguments to the user specifiable method *NoMethod*. The implementation of method *NoMethod* determines the future execution of the method *m* within the scope of other objects existing in the database that may even be members of other object classes.

Delegation of messages to other objects via the method *NoMethod* allows the specification of a particular inheritance behavior for different semantic relationships between objects. In particular, this ability has proven to be useful, when we added specialized modelling primitives for hypermedia and argumentative networks [10], database integration [11], and modelling of multimedia documents [12] to the kernel data model. However, we will not further discuss this feature here.

3.4 Tailoring the Model for Specific Application Needs

The data model of VODAK is an open, adaptable model which provides for the specification of additional modelling primitives at a meta layer of a database schema. That is, a *model designer* can tailor the model to meet specific modelling requirements by introducing appropriate modelling primitives (semantic relationships between classes and their instances) like *aggregation, specialization, generalization, grouping, part-of, etc.* through the definition of metaclasses. The concept of metaclasses and the distinction between classes and types allow to determine a common state and behavior of classes and their instances at the meta layer independent of the specification given at the application layer. In the following, we briefly illustrate how the kernel model can be adapted to meet specific application needs, in our case, the integration of an external relational database system.

Starting with an initial default metaclass system (see the area marked with the pattern in Figure 3) a model designer can adapt the model to integrate external databases by defining metaclasses which provide the semantics needed to map modelling primitives used in the external schema to VML. In Figure 3, the metaclass *PG_METACLASS* is intended to support the straightforward mapping of POSTGRES relations and attributes to VML classes, properties and methods. In analogy to *PG_METACLASS*, the metaclass *SYBASE_METACLASS* could be intended to capture all the common semantics for integrating another relational database system.

Since the metaclass *PG_METACLASS* is defined such that it supports a straightforward mapping, a relation is mapped to exactly one class in VML and a tuple is mapped exactly to one instance of a class. The common behavior and structure of an object which represents a tuple in a relation, i.e. the access methods to the POSTGRES database, are specified by the metaclass *PG_METACLASS* since (1) they are common to *all* classes and instances resulting from the mapping, and (2) they are *independent of* the application, i.e., independent of the contents of the concrete schemas to be integrated. Examples for the common structure and behavior provided by the metaclass *PG_METACLASS* are the following ones:

(1) The access method *get* introduced for the straightforward mapping is sent to instances of a class representing a relation *R*. These instances have to know about the relation R they are derived from, i.e. the relation identifier has to be stored for these instances. Since this information is the same for all instances of a class it is sufficient to store it once with the class. Hence, a class (as an object) needs to have a property and appropriate access methods to store and to retrieve the relation name.

(2) Every instance of a class corresponds to some tuple in a relation. Therefore, every instance needs some property and has to respond to appropriate access methods which allow to store, to assign and to retrieve the key values respectively tuple identifiers of the tuple in order to establish the correspondence between a tuple and the instance.

(3) In addition, a method which retrieves the value of a specific tuple attribute must be defined for every instance in order to enable the mapping between attributes and properties. This is exactly the *get* method introduced earlier in 2.1.

Note, that the properties and methods can be defined once for all POSTGRES schemas independent of the concrete contents of a schema.[4] The method in (3) is made available for the schema designer while the structures in (1) and (2) remain hidden.

A database application designer can now use the functionality provided by the metaclass to define the classes which correspond to the relations. Suppose we have given relations *Conference*, *Tutorial*, *Session*, etc. the designer may define classes *CONFERENCE*, *TUTORIAL*, *SESSION*, etc..

4. If we take into account that one can integrate several relational databases managed by different database systems then one can optimize the design of the different metaclasses by defining an appropriate object type hierarchy for the instance-types and instance-instance-types used for the metaclass definitions in order to avoid redundant definitions.

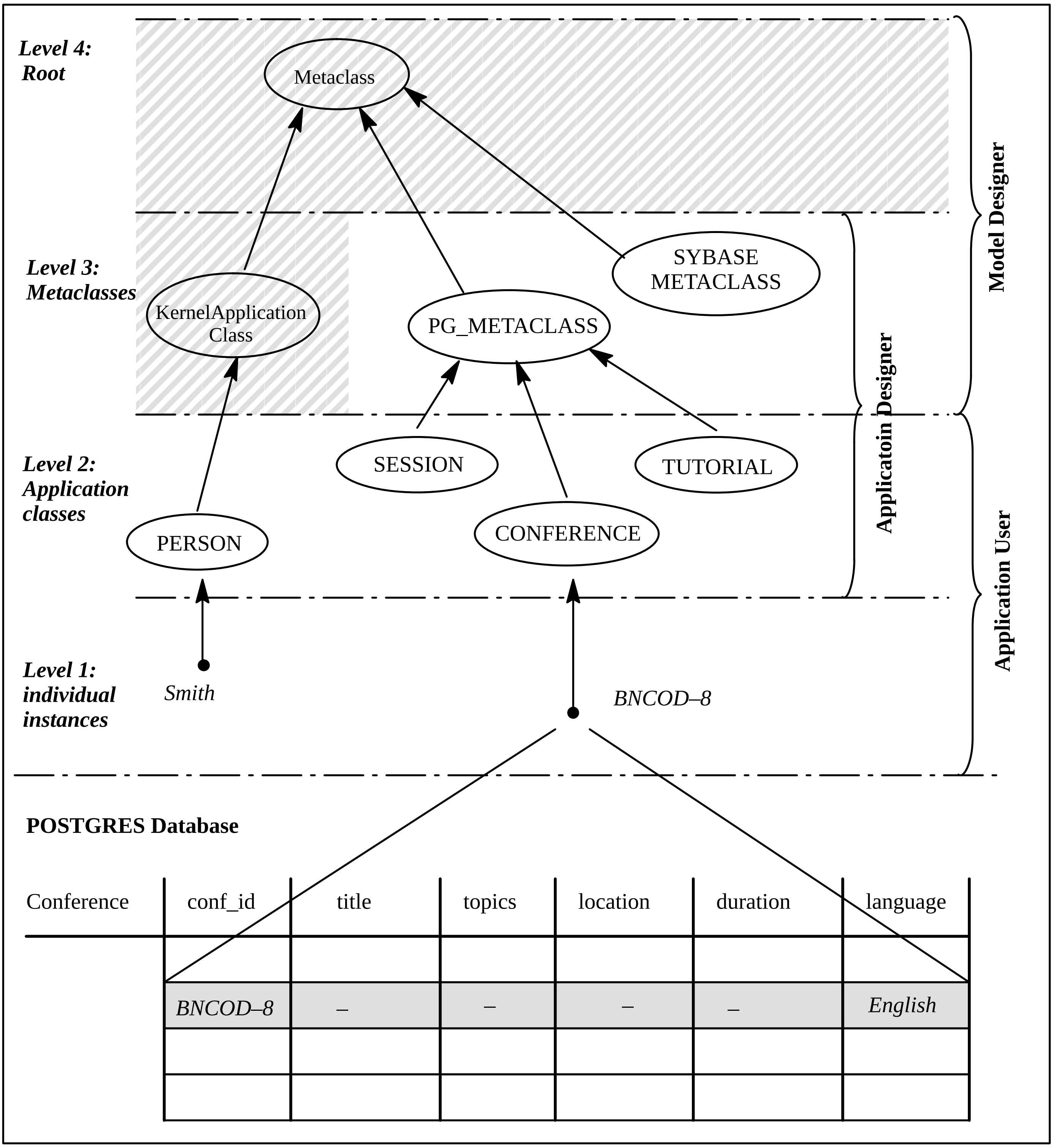


Figure 3: A metaclass modelling the straightforward mapping of a POSTGRES database.

To ensure that these classes actually correspond to specific relations he declares *PG_METACLASS* to be their metaclass, and specifies some initialization (details are shown later). The schema designer is free to specify whatever properties and methods he/she wants to have for the classes and their instances. In order to access the tuples and attribute values in relations he/she just can use the methods provided by the metaclass.

If another type of mapping has to be provided one can introduce another metaclass which provides all the functionality needed by the other kind of mapping. For example, in Figure 4, the metaclass *PG_RECOMPOSE_METACLASS* is intended to provide the common structures and methods needed for the more complex mappings (see 2.2.2 and 2.2.3). Again, these properties and methods can be defined with a metaclass since (1) they are common to *all* classes and instances resulting from the mapping, and (2) they are *independent of* the application.

4 Realization of the Mappings in VML

As we have shown in section 3.4 the mappings of a relational schema to a VML schema can be realized using metaclasses. In this section we will first define the metaclass *PG_METACLASS* for the straightforward mapping and give an example on how to map a relation to a class using the functionality defined in this metaclass. In analogy to that we will define the metaclass *PG_RECOMPOSE_METACLASS,* which provides for complex mappings, and show the recomposition of several relations to a class by means of an example.

4.1 The Metaclass PG_METACLASS

PG_METACLASS realizes a straightforward mapping between relations and classes, i.e. one relation is mapped to one class and vice versa. As we have stated in subsection 2.2.2, the identifier and the key attribute identifiers of a relation have to be stored with the corresponding class to enable the creation of the class' extent with respect to the relation's extent. POSTGRES allows a decisive simplification concerning the key attributes of a relation: the attribute *oid* (a tuple identifier) is implicitly defined with every POSTGRES relation. *oid* serves as a key since its value is automatically computed and left unchanged whenever a new tuple is inserted in the relation using the actual data and time. Because *oid* is defined with every relation it can be regarded as a universal, application independent key. It follows that the access method provided for the instances of a class can use *oid* instead of the actual, application dependent key attributes of the corresponding relation.

The relationships between the predefined metaclass Metaclass, the metaclass PG_METACLASS and some application classes are shown in Figure 3.

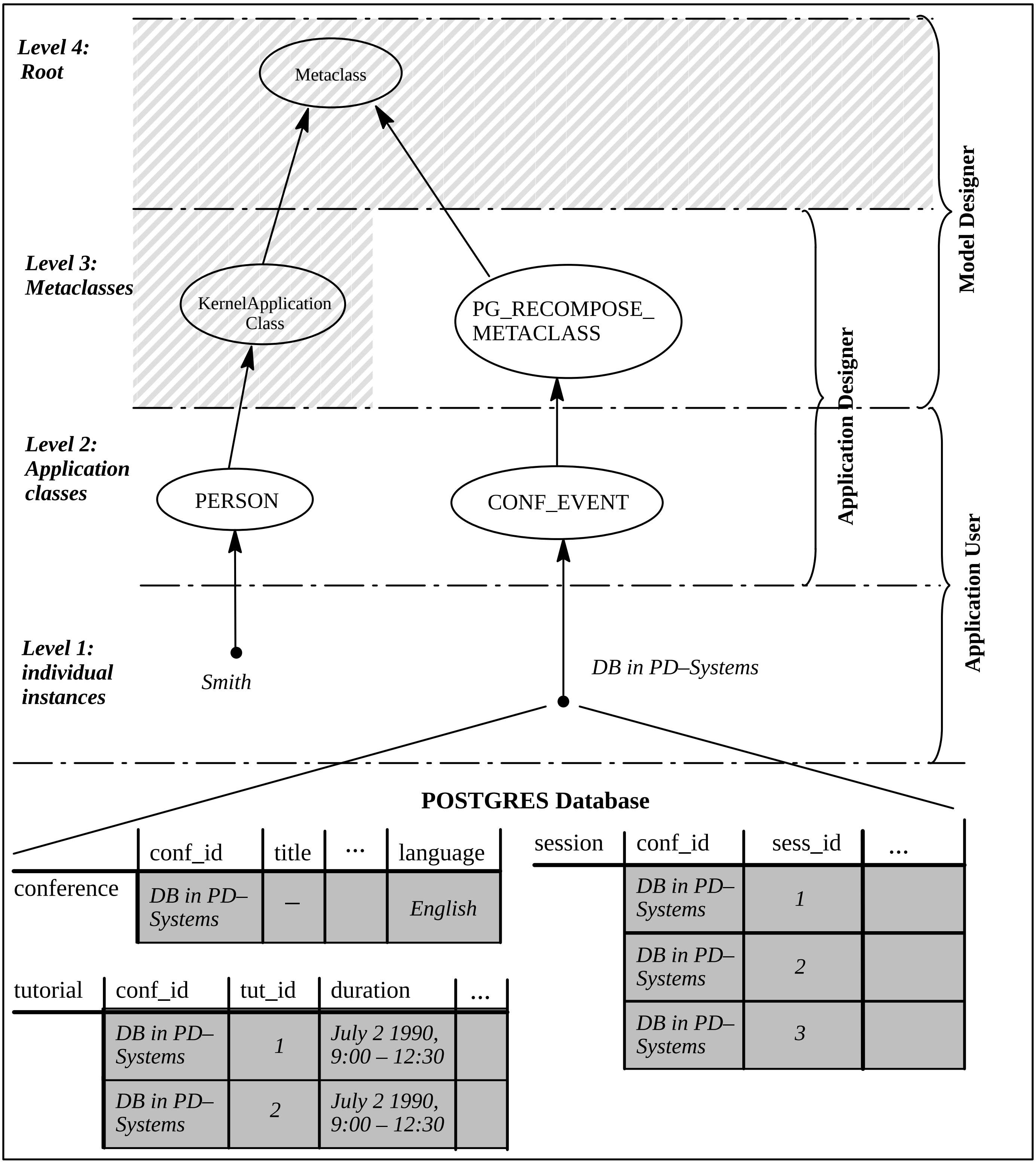


Figure 4: A metaclass modelling more complex mappings of a POSTGRES database.

4.1.1 Definition of the Metaclass PG_METACLASS

The access to the POSTGRES database within the methods of the own type, instance type and instance–instance type of PG_METACLASS is realized using POSTGRES C library functions. In order to provide a better understanding of the realization of the mapping with VML we describe the effects of those methods instead of showing the actual VML/C++ method implementation.

The metaclass PG_METACLASS and its associated object types are defined as follows:

(1) Definition of PG_METACLASS

```
CLASS PG_METACLASS METACLASS Metaclass
OWNTYPE PG_Metaclass_OwnType
INSTTYPE PG_Metaclass_InstType
INSTINSTTYPE PG_Metaclass_InstInstType
END;
```

(2) Definition of the object type PG_Metaclass_OwnType

The own type of the metaclass PG_METACLASS defines the methods *linkdb* and *unlinkdb* which initialize and terminate the communication to a POSTGRES database.

```
OBJECTTYPE PG_Metaclass_OwnType;
   INTERFACE
   METHODS  linkdb(db: STRING);
                  unlinkdb();
   IMPLEMENTATION
      METHODS
      linkdb(db: STRING);
         { // initialize communication with POSTGRES database db
            };
      unlinkdb();
         { // terminate communication with currently accessed POSTGRES database
         };
END;
```

(3) Definition of the object type PG_Metaclass_InstType

The instance type of the metaclass PG_METACLASS defines the property *relation* and the methods *getRel* and *init* which are available for application classes defined as instances of PG_METACLASS.

The identifier of the relation corresponding to a class is stored in property *relation*; the method *getRel* just returns the value of this property. The method *init* assigns the value of its parameter to the property *relation* and retrieves all actual values of the attribute *oid* from the corresponding relation, creates an instance of the class for each value and stores this value with the new instance in property *PG_Oid* in order to enable further access to non-key attributes[5].

```
  OBJECTTYPE PG_Metaclass_InstType SUBTYPEOF Metaclass_InstType;
      INTERFACE
      PROPERTIES  relation: STRING;
      METHODS     init(rel: STRING);
                    getRel(): STRING;
      IMPLEMENTATION
         METHODS
            init(rel: STRING);
           { relation := rel;
                 // retrieve set of tuple identifiers from relation rel;
                 // create an instance of the class which executes the method
                 // init for each retrieved value and store this value in
                 // property PG_Oid of the new instance
           };
         getRel(): STRING;
           { RETURN relation; };
END;
```

(4) Definition of the object type PG_Metaclass_InstInstType

The instance-instance type of the metaclass PG_METACLASS defines the property *PG_Oid* and the methods *setPG_Oid*, *getPG_Oid* and *getValue* which are available for the instances of application classes defined as instances of PG_METACLASS. The value of the attribute *oid* of the tuple corresponding to the instance is stored in property *PG_Oid*. The methods *setPG_Oid* and *getPG_Oid* store and return the value of property *PG_Oid*. The method *getValue(att: STRING): STRING* is used to retrieve single values of attributes of the relation which corresponds to the class of the receiver object. As parameters it takes the identifier of the attribute of which the value is requested. *getValue* uses the identifier of the corresponding tuple stored with the receiver to determine the correct tuple in the database.

5. Usually, the creation of the instances which correspond to the tuples of the relation will be done dynamically on demand. But to simplify the presentation we will not show the implementation for this alternative.

```
OBJECTTYPE PG_Metaclass_InstInstType SUBTYPEOF Metaclass_InstInstType;
   INTERFACE
   METHODS      getValue(att: STRING): STRING;
                      setPG_Oid(oid: STRING);
   IMPLEMENTATION
    PROPERTIES    PG_Oid: STRING;
      METHODS
         setPG_Oid(oid: STRING);
         { PG_Oid := oid; // is used in init() };
      getPG_Oid(): STRING;
         { RETURN PG_Oid; // used in method getValue()};
      getValue(att: STRING): STRING;
             {  // Retrieve attribute att from the tuple corresponding to the
                // instance receiving the method;
                // use the relation identifier stored with the class of the instance
                // and the value of PG_Oid in order to construct the appropriate
                // POSTGRES retrieval statement.
                // Since the POSTGRES C library functions only return attribute values
                // as strings, return retrieved value as a string and leave the conversion to
                // other datatypes to the application programmer
             };
END;
```

4.1.2 Example for a Straightforward Mapping

Let us assume that we have given the following POSTGRES relation *conference* with the key attribute *conf_id*.

conference (conf_id, title, topics, location, duration, language)

We define now a class CONFERENCE by using PG_METACLASS as its metaclass. The structure and behavior of instances of class CONFERENCE is defined by the object type *conference_InstType*. We use the *init* method provided with the metaclass PG_METACLASS to express that the relation *conference* is mapped to the class CONFERENCE.

For each attribute given in the relation *conference* we define a property as we want to have this attribute in our application domain. In this example we defined properties *title, location,* and *language*, which correspond to the appropriate attributes of the relation. The properties *conf_id, topics,* and *duration* are defined in the same way through the supertypes of *conference_InstType*. This is, because the corresponding attributes of the relation *conference* appear in other relations of the POSTGRES schema too. Hence, in order to avoid repeated definitions of these properties they are defined once by supertypes which are shared by the object types of several classes.

All implementations of methods which retrieve the property values, i.e., which retrieve the attribute values from the POSTGRES database, follow the same scheme: First, they test whether the value for the property has already been retrieved from the database. If not, the value is retrieved from the underlying POSTGRES database by using the method *getValue* provided with the metaclass PG_META-CLASS. This method actually generates a POSTGRES retrieval statement to get the value from the database and then returns it. In our implementation, we assign this value to the property, before returning the value to the calling method. Note, that the (intermediate) storage of the value as a property value is optional and depends on the requirements of an application. Storing the value with a property allows for much faster subsequent retrievals, but imposes restrictions to the applications with respect to autonomous updates of the external database. Note, that for a given strategy, one can generate the object type definitions and the implementations of the methods automatically.

```
OBJECTTYPE conference_InstType;
    INTERFACE
    PROPERTIES conf_id: STRING;
               title: STRING;
               topics: STRING;
               location: STRING;
               duration: STRING;
               language: STRING;
    METHODS    getconf_id() : STRING;
               gettitle(): STRING;
               gettopics): STRING;
               getlocation(): STRING;
               getduration(): STRING;
               getlanguage(): STRING;
    IMPLEMENTATION
      METHODS
      getconf_id(): STRING;
        { IF ( conf_id == ’UNKNOWN VALUE’ )   title := SELF→getValue(’conf_id’);
            RETURN conf_id; };
      gettitle(): STRING;
        { IF ( title == ’UNKNOWN VALUE’ ) title := SELF→getValue(’title’);
            RETURN title; };
        gettopics(): STRING;
        { IF ( topics == ’UNKNOWN VALUE’ ) topics := SELF→getValue(’topics’);
            RETURN topics; };
// The methods getlocation, getduration and getttitle are implemented analogously.
  END;
```

```
CLASS CONFERENCE METACLASS PG_METACLASS
   INSTTYPE conference_InstType
INIT CONFERENCE→init('conference')
END;
```

4.2 The Metaclass PG_RECOMPOSE_METACLASS

PG_RECOMPOSE_METACLASS realizes more complex mappings between relations and classes. In general, the definition is structured similarly as shown for the metaclass PG_METACLASS. It differs insofar as we now have to provide more complex access methods as described in subsections 2.2.3 and 2.2.4. The relationships between the predefined metaclass *Metaclass*, the metaclass PG_RECOMPOSE_METACLASS and an application class have been shown in Figure 4.

4.2.1 Definition of the Metaclass PG_RECOMPOSE_METACLASS

Again, the access to the POSTGRES database within the methods of the own type, instance type and instance-instance type of PG_RECOMPOSE_METACLASS is realized using POSTGRES C library functions. In order to provide a better understanding of the realization of the mapping with VML we describe the effects of those methods instead of showing the actual VML/C++ method implementation.

The metaclass PG_RECOMPOSE_METACLASS and its associated object types are defined as follows:

(1) Definition of PG_RECOMPOSE_METACLASS

```
CLASS PG_RECOMPOSE_METACLASS METACLASS Metaclass
OWNTYPE PG_Metaclass_OwnType
INSTTYPE PG_Metaclass_InstType
INSTINSTTYPE PG_RECOMPOSE_Metaclass_InstInstType
END;
```

(2) Definition of the object type PG_RECOMPOSE_InstInstType

The instance-instance type of the metaclass PG_RECOMPOSE_METACLASS is a subtype of PG_Metaclass_InstInstType, and hence the method *getValue(att: STRING): STRING* and the mechanisms to support the mapping of the tuple to object identifiers are inherited from this type. Additionally the type PG_RECOMPOSE_InstInstType provides three further methods, which are specializations of the general methods *getValue* and *getOID* introduced in subsection 2.2.3 and 2.2.4. The method *getValue (rel: STRING, joinatt : {STRING}, att : { STRING }): { || STRING —> STRING|| }* recomposes complex values by joining the relation corresponding with the receiver object's class with relation *rel* using the set of join attributes *joinatt* and retrieving the values of the attributes *att* of those tuples of *rel* meeting the join condition. These values are returned as a set of dictionaries. Dictionaries are used here as a technique to represent arbitrary tuple structures. The methods g*etOID(rel: STRING, joinatt : {STRING}): {OID}* and *getOID(*rel_1*:STRING,* $joinatt_1$*: {STRING},* rel_2*:STRING,* $joinatt_2$*: {STRING}): {OID}* are provided in order to establish reference-based relationships. The former computes the same join as the method *getValue* just described before, but retrieves the tuple identifiers of the tuples of *rel* which meet the join conditions and returns the object identifiers of the instances of the class corresponding to *rel* which represent those tuples. The latter simply joins three relations (the one corresponding with the receiver object's class, rel_1 and rel_2) in the same way, retrieves the tuple identifiers of those tuples in rel_2 meeting the join conditions and returns the object identifiers of the appropriate instances of the class corresponding to rel_2. For better readability we define only the two methods described above for computing object-based references from value-based references. It is straightforward to define a general, highly parametrized method which computes a join between *n* relations.

```
OBJECTTYPE   PG_RECOMPOSE_Metaclass_InstInstType
             SUBTYPEOF PG_Metaclass_InstInstType;
    INTERFACE
       METHODS  getOID(rel: STRING, joinatt: {STRING}): {OID};
                getValue(rel:STRING,joinatt:{STRING},att:{STRING}):{||STRING—>STRING||};
                getOID(rel1:STRING,joinatt1:{STRING},rel2:STRING,joinatt2:{STRING}):{OID};
  IMPLEMENTATION
       METHODS
          getOID(rel: STRING, joinatt : {STRING}): {OID};
            {  //  Join the relation corresponding to the receiver object's class with
               // the relation rel using joinatt. Retrieve the tuple identifiers of the tuples of rel
               // which meet the join condition; find the class corresponding to rel and return the
               //  object identifiers of its instances corresponding with those tupels.};
          getValue (rel: STRING, joinatt : {STRING}, att : { STRING }): { ||STRING —> STRING|| } ;
               {// Join the relation corresponding to the receiver object's class with the relation rel
               //  using joinatt; retrieve the values of attributes {att} from those tuples in rel which
               // meet the join condition and return those values. Those values are returned as a
               //  set of dictionaries, where the key of the dictionary represents the attribute name
               // and the value of the dictionary the attribute value as a string in relation rel
               //  (this preserves the relational tuple structure) };
```

```
          getOID(rel1:STRING, joinatt1: {STRING}, rel2:STRING, joinatt2: {STRING}): { OID };
             {  //  Join the relation corresponding to the receiver object's class with
                //  the relations rel1 and rel2 using joinatt1 and joinatt2. Retrieve the
                //  tuple identifiers of the tuples of rel2 which meet the join condition;
                //  find the class corresponding to rel2 and return the
                //  object identifiers of its instances corresponding with those tupels.};
END:
```

4.2.2 Example for a Recomposition Mapping

Let us assume that we have given the following fragment of a POSTGRES schema:

conference	(conf_id, title, topics, location, duration, language)
conf_info_address	(conf_id, name, mail_addr, city, state, country, phone_no, fax_no)
session	(conf_id, sess_id, duration, subject, chair_name, chair_affil)
lectures	(conf_id, sess_id, lecturer_name, lecturer_affil, topic)
hotel	(name, mail_addr, city, state, country, phone_no, fax_no)
reservation	(conf_id, hotel_name)

Let us assume that we want to recompose these relations to classes according to the mapping described previously.

The attribute *conf_id* is the primary key of relation *conference* and a foreign key in the other relations. Therefore *conference* is the base relation, *conf_info_adress*, *session* and *lectures* are non-base relations. The instances of the class CONF_EVENT which results from the mapping correspond to the tuples of *conference*; the attributes of *conference* and *conf_info_adress* are modelled as single-valued properties since *conf_id* is the only key attribute in these relations. In contrast to that, *conf_id* is only a part of the set of key attributes in the relations *session* and *lectures*; there may exist several tuples in *session* for a tuple in *conference*. Furthermore, since the key attribute *sess_id* of relation *session* is a foreign key attribute in relation *lectures*, for a tuple in *session* there may exist several tuples in *lectures*. Therefore the attributes of *session* and *lectures* are combined to a complex set-valued property with nested tuple structure which models all sessions of a conference including the corresponding lectures.

The relation *hotel* is a base relation since it does not contain a foreign key. It is therefore mapped to a separate class HOTEL. The relation *reservation* represents a relationship between *conference* and *hotel*. It was generated as a consequence of the decomposition of the relational schema. Consequently this relation will dissolve in the mapping since it can be substituted by object references between the instances of the classes CONF_EVENT and HOTEL.

The classes CONF_EVENT and HOTEL and their instance types CONF_EVENT_InstType and HOTEL_InstType are defined as follows:

```
DATATYPE conf_type = [conf_id: STRING, title : STRING, topics : STRING,
           location : STRING, duration : STRING, language : STRING];
DATATYPE conf_info_address_type = [name : STRING, mail_addr : STRING,  city : STRING,
           state : STRING, country : STRING, phone_no: ARRAY[SUBRANGE 0..15] OF INT,
          fax_no : ARRAY[SUBRANGE 0..15] OF INT];
DATATYPE lectures_type = [ lecturer_name : STRING, lecturer_affil : STRING, topic : STRING];
DATATYPE session_type =  [sess_id: INT, duration : STRING , subject : STRING,
           chair_name : STRING, chair_affil : STRING, lectures : { lectures_type  } ];
```

```
OBJECTTYPE CONF_EVENT_InstType;
 INTERFACE
   PROPERTIES
     conference: conf_type;
     conf_info_address: conf_info_address_type;
     sessions : {session_type}; // nested structure of sessions and lectures
     hotels: {HOTEL}
   METHODS
     getConfernce_conf_id(): STRING;
     getConference_title(): STRING;
     // ...
     getConf_info_address_name(): STRING;
     // ...
    getSessions(): {session_type} ;
 IMPLEMENTATION
   EXTERN StringToInt(s: STRING): INT; // converts a string to an integer
   METHODS
     getConference_conf_id(): STRING;
     { IF (conference.conf_id == 'UNKNOWN VALUE')
       conference.conf_id := SELF→getValue('conf_id');
       RETURN conference.conf_id;};
     getConference_title(): STRING;
     { IF (conference.title == 'UNKNOWN VALUE')
       conference.title := SELF→getValue('title');
       RETURN conference.title;};
//  Other methods operating on property conference are implemented analogously
     getConf_info_address_name(): STRING;
     { IF (conf_info_address.name == 'UNKNOWN VALUE')
       conf_info_address.name := SELF→getValue('conf_info_address', {'conf_id'}, {'name}');
       RETURN conf_info_address.name;};
 // Other methods operating on property conf_info_address are implemented analogously
```

```
    getSessions(): {session_type};
    { VAR actSess : {||STRING—>STRING||};
      VAR actLect: {||STRING—>STRING||};
      VAR actSessTuple: session_type;
      VAR actLectTuple: lectures_type;
      VAR s : ||STRING—>STRING||;
      VAR l : ||STRING—>STRING||;

        IF (sessions == {})
      { actSess := SELF→getValue('session', {'conf_id'},
                  {'sess_id', 'duration', 'subject', chair_name', chair_affil'}); // retrieve all sessions
          actLect := SELF→getValue('lectures', {'conf_id'},
                      {'sess_id', 'lecturer_name', 'lecturer_affil', 'topic'}); // retrieve all lectures
    // combine sessions and lectures to one nested structure
          FORALL (s IN actSess)
          {  actSessTuple.sess_id:= StringToInt (GETVALUE s FROM 'sess_id');
             actSessTuple.duration:= GETVALUE s FROM 'duration';
             actSessTuple.subject:= GETVALUE s FROM 'subject';
             actSessTuple.chair_name:= GETVALUE s FROM 'chair_name';
             actSessTuple.chair_affil:= GETVALUE s FROM 'chair_affil';
             FORALL (l IN actLect)
             { IF (actSessTuple.sess_id == StringToInt(GETVALUE l FROM 'sess_id'))
                { actLectTuple.lecturer_name := GETVALUE l FROM 'lecturer_name';
                 actLectTuple.lecturer_affil := GETVALUE l FROM 'lecturer_affil';
                 actLectTuple.topic := GETVALUE l FROM 'topic';
                 INSERT actLectTuple INTO actSessTuple.lectures; } }
             INSERT actSessTuple INTO sessions;}}
        RETURN sessions;};
   getHotels(): {OID};
    { IF (hotels == {})
        hotel:= SELF→getOID('reservation', {'conf_id'}, 'hotel', {'hotel_name'});
      RETURN conferences;}
 END;

CLASS CONF_EVENT METACLASS PG_RECOMPOSE_METACLASS
  INSTTYPE CONF_EVENT_InstType
  INIT CONF_EVENT→init ('conference')
 END;
```

```
OBJECTTYPE HOTEL_InstType;
INTERFACE
   PROPERTIES
      hotel_name: STRING;
      mail_addr: STRING;
      city: STRING;
      state: STRING;
      country: STRING;
      phone_no: ARRAY [SUBRANGE 0..15] OF INT;
      fax_no: ARRAY [SUBRANGE 0..15] OF INT;
      conferences: {CONF_EVENT};
   METHODS
     getHotel_name(): STRING;
     getMail_addr(): STRING;
     getCity(): STRING;
     getState(): STRING;
     getCountry(): STRING;
     getPhone_no(): ARRAY [SUBRANGE 0..15] OF INT;
     getFax_no(): ARRAY [SUBRANGE 0..15] OF INT;
     getConferences(): {OID};
IMPLEMENTATION
   EXTERN StringToArray (s: STRING) : ARRAY [SUBRANGE 0..15] OF INT;
   // This is an external function to convert strings to arrays (provided the string
   // contains single numbers separated by spaces. It can be used in the following
   // method implementations
   getHotel_name(): STRING;
     { IF ( hotel_name == 'UNKNOWN VALUE' )   hotel_name := SELF→getValue('hotel_name');
        RETURN hotel_name; };
//  The methods getMail_Addr, ..,  gettCountry are implemented analogously.
   getPhone_no(): ARRAY [SUBRANGE 0..15] OF INT;
     { IF ( phone_no[0] == 0)
          phone_no := StringToArray (SELF→getValue('phone_no'));
       RETURN phone_no;};
//  The method getFax_no is implemented analogously.
     getConferences(): {OID};
       { IF (conferences == {})
          conferences:= SELF→getOID('reservation', {'hotel_name'}, 'conference', {'conf_id'});
         RETURN conferences;};
END:

CLASS HOTEL METACLASS PG_RECOMPOSE_METACLASS
  INSTTYPE HOTEL_InstType
  INIT HOTEL→init ('hotel')
END;
```

5 Conclusion

In this paper we have shown the integration of a relational database into the object-oriented federated database management system VODAK using the metaclass concept of the VODAK Modelling Language VML. First we described a straightforward mapping between the relational and the object-oriented data model. Then, based on comparison of the expressive power of the two data models we defined more complex mappings which reduce the gap between the two worlds by allowing a semantic enrichment of the relational schema within the object-oriented schema. In order to support these mappings efficiently we introduced specific access methods. Then we introduced the object-oriented data model of VODAK which is used as the canonical model for the mapping, and focused on the metaclass concept of VML. As an example we gave a short overview of an actual implementation of the access methods to POSTGRES databases making use of the metaclass concept. This implementation is based on POSTGRES V4.0. We illustrated the prototype implementation applied to a fairly complex relational schema.

Beside the standard features of a relational database management system POSTGRES provides advanced concepts like functions and rules. In principle one could incorporate such schema information in method bodies. This was not investigated so far since the access to this schema information was not readily available.

The metaclasses are the bases for the integration of relational databases into VODAK. (Semi-)Automatic integration tools can use these metaclasses in the integration process. We propose two interesting directions of further research: first, a limited but automatic translation capability of relational into object-oriented schemas more advanced than the straightforward approach; second, interactive tools for schema integration, both based on the mappings which allow for semantic enrichment of the relational schemas.

6 Literature

[1] Bright, M.W.; Hurson, A.R.; Pakzad, S.H.: A Taxonomy and Current Issues in Multidatabase Systems. *IEEE Computer*, Vol. 25, No. 3, p. 50–59, March 1992.

[2] Dwyer, P.; Larson, J.: Some experiences with a distributed database testbed system. *Procedings of the IEEE 75*, 5, pp. 633-647, May 1987.

[3] Elmasri, R.; Navathe, S.B.: *Fundamentals of Database Systems*.The Benjamin/Cummings Publishing Company, 1989.

[4] Fankhauser, P.; Kracker M.; Neuhold, E.J: Semantic vs. Structural Resemblance of Classes. *SIGMOD Record*, Vol. 20, No. 4, pp. 50–63, December 1991.

[5] Fankhauser, P.; Neuhold E.J.: Knowledge-Based Integration of Heterogeneous Databases. *Proceedings of IFIP DS-5 Conference – Semantics of Interoperable Database Systems*, pp. 150–170, Lorne, Victoria, Australia, November 16th-20th, 1992.

[6] Goettke T.; Fankhauser P.: DREAM 2.0, User Manual. *Technical Report No.660*, GMD-IPSI, St. Augustin, July 1992

[7] Jacobsen, G.; Piatetsky-Shapiro, G.; Lafond, C.; Rajinikanth, M.; Hernandez, J.: CALIDA: A knowledge-based system for integrating multiple heterogeneous databases. *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pp. 3-18, Jerusalem, Israel, June 1988.

[8] Kaul, M.; Drosten, K.; Neuhold, E.J.: ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. *Proceedings of the 6th IEEE International Conference on Data Engineering*, pp. 2–10, Los Angeles, California, February 5–9, 1990.

[9] Kaul, M.; Neuhold, E.J.: Supporting Interoperability of Heterogeneous Information Bases by Complex View Heterarchies. *Workshop "Perspektiven der Datenbank-Technik"*, University of Bern, October 24–25, 1990, .

[10] Klas, W.; Neuhold, E.J.: Designing Intelligent Hypertext Systems Using an Open Object-Oriented Database Model. *Technical Report No. 489*, GMD-IPSI, St. Augustin, 1990.

[11] Klas, W.; Neuhold, E.J.; Schrefl, M.: Metaclasses in VODAK and their Application in Database Integration. *Technical Report No. 462*, GMD-IPSI, St. Augustin, 1990.

[12] Klas, W.; Neuhold, E.J.; Schrefl, M.: Using an Object-Oriented Approach to Model Multimedia Data. In: *Computer Communications, Special Issue on Multimedia Systems*, Vol. 13, No. 4, May 1990.

[13] Klas, W. Aberer K., Neuhold E.J.: Object-Oriented Modeling for Hypermedia Systems using the VODAK Modelling Language (VML), *Object-Oriented Database Management Systems*, NATO ASI Series, Springer Verlag Berlin Heidelberg, August 1993.

[14] Landers, T.; Rosenberg, R.: An overview of Multibase. In: *Distributed Databases*, H.-J. Schneider, Ed., North-Holland, The Netherlands, pp. 153-184, 1982.

[15] Litwin, W.: An overview of the multidatabase system MRDSM. *Proceedings of the ACM National Conference*, pp. 495-504, Denver, October 1985.

[16] Litwin, W.; Boudenant, J.; Esculier, C.; Ferrier, A.; Glorieux, A.; La Chimia, J.; Kabbaj, K.; Moulinoux, C.; Rolin, P.; and Stangret, C.: SIRIUS Systems for Distributed Data Management. In: *Distributed Data Bases*, H.-J. Schneider, Ed. North-Holland, The Netherlands, pp. 311-366, 1982.

[17] Litwin, W.; Mark, L.; Roussopoulos, N.: Interoperability of Multiple Autonomous Databases. *ACM Computing Survey, Vol. 22, No. 3,* pp. 267–293, September 1990.

[18] Mehta A.; Geller, J.; Perl,Y.; Fankhauser, P.: Algorithms for Computing Access Relevance in Object-Oriented Databases. *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, Maryland, November 8–11, 1992.

[19] Mehta, A.: Geller, J.: Perl, Y.: Fankhauser, P.: Algorithms for Access Relevance to Support Path-Method Generation in OODBs. *Proceedings of the Fourth International Hong Kong Computer Soc. Database Workshop*, Chinese University of Hong Kong, Shatin, Hong Kong, December 12 – 13, 1992.

[20] Neuhold, E. J.; Schrefl, M.: Dynamic Derivation of Personalized Views. *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 183–194, Los Angeles, CA, August 29 – September 1, 1988.

[21] Rusinkiewicz, M.; Elmasri, R.; Czejdo, B.; Georakopoulous, D.; Karabatis, G.; Jamoussi, A.; Loa, L.; Li, Y.: OMNIBASE: Design and implementation of a multidatabase system. *Proceedings of the 1st Annual Symposium in Parallel and Distributed Processing*, pp. 162-169, Dallas, Texas, May 1989.

[22] Saltor, F.; Castellanos, M; Carcia-Solaco M.: Suitability of data models as canonical models for federated datasbases. *SIGMOD Record*, Vol. 20, No. 4, pp. 44–48, December 1991.

[23] Schrefl, M.; Neuhold, E.: A Knowledge-Based Approach to Overcome Structural Differences in Object Oriented Database Integration. In*: The Role of Artificial Intelligence in Database & Information Systems*, IFIP Working Conference, pp. 265–304, Guangzhou, China, July 4–8 1988.

[24] Schrefl, M.; Neuhold, E.J.: Object class definition by generalization using upward inheritance. *Proceedings of the 4th IEEE International Conference on Data Engineering*, pp. 4–13, Los Angeles, California, February 1–5, 1988.

[25] Sheth, A. P.; Larson, J. A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Survey, Vol. 22, No. 3, September 1990.*

[26] Templeton, M.; Brill, D.; Hwang, A.; Kameny, I.; Lund, E.: An overview of the mermaid system. A frontend to heterogeneous databases. *Proceedings of EASCON 1983.*

[27] Yan, L.L.; Ling, T.W.: Translating Relational Schema with Constraints into OODB Schema. *Proceedings of IFIP DS-5 Conference – Semantics of Interoperable Database Systems*, pp. 65–81 , Lorne, Victoria, Australia, November 16th-20th, 1992.