

# Flattening the Metamodel for Object Databases <sup>1</sup>

Piotr Habela<sup>1</sup>, Mark Roantree<sup>2</sup> and Kazimierz Subieta<sup>1,3</sup>

<sup>1</sup> Polish-Japanese Institute of Information Technology, Warsaw, Poland

<sup>2</sup> School of Computer Applications, Dublin City University, Dublin, Ireland

<sup>3</sup> Institute of Computer Science PAS, Warsaw, Poland

**Abstract.** A metamodel definition presents some important issues in the construction of an object database management system, whose rich data model inevitably increases the metamodel complexity. The required features of an object database metamodel are investigated. Roles of a metamodel in an object-oriented database management system are presented and compared to the proposal defined in the ODMG standard of object-oriented database management systems. After outlining the metamodel definition included in the standard, its main drawbacks are identified and several changes to the ODMG metamodel definition are suggested. The biggest conceptual change concerns flattening the metamodel to reduce complexity and to support extensibility.

## 1 Introduction

A database metamodel is a description of those database properties that are not dependent on a particular database state. A metamodel implemented in a DBMS is provided to formally describe and store the database schema, together with data such as the physical location and organization of database files, optimization information, access rights and integrity and security rules. Metamodels for relational systems are easy to manage due to the simplicity of the data structures implied by the relational model. In these systems, the metamodel is implemented as a collection of system tables storing entities such as: identifiers and names of relations stored in the database; identifiers and names of attributes (together with identifiers of relations they belong to); and so on.

A metamodel presents an important issue in the construction of an object database management system, whose rich data model inevitably increases the metamodel complexity. One of the arguments against object databases concerns their complex metamodels, which (according to [8]) lead to a ‘nightmare’ with their management. The negative opinion concerning object database metamodel seems to be confirmed by examining the corresponding proposal in the recent ODMG standard [3] which contains in excess of 150 features to represent interfaces, bi-directional relationships, inheritance relationships and operations. In our opinion, the metamodel proposed by

---

<sup>1</sup> This work is partly supported by the EU 5th Framework project ICONS (Intelligent Content Management System), IST-2001-32429.

ODMG should not be treated as a decisive solution but as a starting point for further research. Some complexity in comparison to the relational model is inevitable, as a minor cost of major advantages of the object model. It is worth mentioning the Interface Repository of the OMG CORBA standard [12] (a pattern for the ODMG metamodel), which was primarily designed as a facility for the Dynamic Invocations Interface (DII) (i.e. programming through reflection). Despite the complexity, the CORBA community accepts this solution.

In this paper we would like to identify all major issues related to the construction of the metamodel for object databases and to propose some solutions. We address the following goals that a metamodel should fulfill:

- **Data Model Description.** This should be in a form where it can be understood by all parties, including system developers, users, administrators, researchers and students. The metamodel specifies interdependencies among concepts used to build the model, some constraints, and abstract syntax of data description statements; thus it supports the intended usage of the model.
- **Implementation of DBMS.** A metamodel determines the organization of a metabase (usually referred to as a catalog or a schema repository). It is internally implemented as a basis for database operations, including database administration and various purposes of internal optimization, data access and security.
- **Schema evolution.** A metamodel equipped with data manipulation facilities on metadata supports schema evolution. Changes to a schema imply a significant cost in changes to applications acting on the database. Thus schema evolution cannot be separated from software change and configuration management.
- **Generic programming.** The metamodel together with appropriate access functions becomes a part of the programmer's interface to enable generic programming through reflection, similarly to Dynamic SQL or CORBA DII.
- **Formal description of local resources** (called *ontology*) in distributed/federated databases or agent-based systems. This aspect is outside the scope of the paper.

As will be demonstrated, these metamodel goals are contradictory to some extent. The following sections present peculiarities and requirements connected with each of the aforementioned goals. The remainder of the paper is devoted to our contribution, which focuses on the construction of a metamodel for object databases, with the required tradeoff between these goals. The paper is organized as follows: in section 2 we briefly discuss various roles and issues related to an object database metamodel and compare them to the metamodel presented in the ODMG standard; in section 3 we postulate on possible features of the metamodel which can be considered as improvement to the ODMG proposal, where the biggest conceptual change concerns flattening a metamodel to reduce complexity and to support extensibility; section 4 provides an example of a metamodel that includes some of the features suggested in section 3; and in section 5 we provide some conclusions.

## 2 Roles of an Object Metamodel

In this section we discuss the particular roles and issues related to the topic of an object-oriented database metamodel, and compare them to the proposal presented in

the ODMG standard. The general conclusion is that the ODMG metamodel specification has drawbacks, and thus, needs improvements.

## 2.1 Data Model Description

Among the requirements mentioned in the introduction, the descriptive function of the metamodel is probably the most straightforward and intuitive. It is important to attempt to provide a clear and unambiguous definition of data model primitives and their interrelations. An example is provided by the Unified Modeling Language (UML) [2,11], whose metamodel provides quite a useful and expressive (although informal) definition of the meaning of language constructs. It is doubtful as to whether such a metamodel is a full description of UML semantics. This definitional style suffers from the *ignotum per ignotum* logical flaw (concepts are defined through undefined concepts; definitions have cycles but they are not recursive). The metamodel bears informal semantics through commonly understood natural language tokens and a semi-formal language. The formal data semantics of class diagrams can be expressed through a definition of the set of valid data (database) states and by mapping every UML class schema into a subset of the states [18]. Semantics of method specifications requires other formal approaches, e.g. the denotational model. Such a formal approach would radically reduce ambiguities concerning UML; however, due to the rich structure and variety of UML diagrams, the formal semantics presents a hard problem. Instead of using formal semantics, the UML metamodel presents an abstract syntax of data description statements, and various dependencies and constraints among introduced concepts.

The ODMG standard [3] follows the UML style. The metamodel presents interdependencies among concepts introduced in the object model and covers the abstract syntax of the Object Definition Language (ODL). In contrast to UML, however, the ODMG metamodel is associated with a number of retrieval and manipulation capabilities. This suggests that the intention behind the metamodel is description of access functions to the database repository rather than pure description of concepts introduced in the ODMG object model and ODL. Access to the repository is wrapped inside a set of ODL interfaces, where a particular interface usually describes a specified ODMG object model concept. Association and generalization relationships are extensively used to show interdependencies among metamodel elements, which results in a large, tightly-coupled structure.

There are flaws in the style that the ODMG uses to explain the goals and semantics of the metamodel, together with a lack of many definitions and explanations. In its present form, this part of the standard is underspecified and ambiguous, thus making it difficult (or impossible) to understand the intended usage of its features.

## 2.2 Implementation of DBMS

Considering the second role of a metamodel, we distinguish the following criteria to evaluate its quality:

- **Simplicity.** The metamodel and metabase should be simple, natural, minimal and easy to understand, in order to be efficiently used by developers of DBMS and database administrators.
- **Universality.** Implementation of database languages and operations requires various accessing and updating operations to the metabase. The metamodel should support all such operations, and these operations should match, as closely as possible, similar operations for regular data.
- **Performance.** Metabase operations that originate from the database management system or from applications, may be frequent, and thus it is important to organize the metabase so as to guarantee fast run-time access and updating.
- **Physical data structure information support.** Data describing physical structures (e.g. file organizations, indices, etc.) as well as data used for optimization (access statistics, selectivity ratios, etc.) must be included in the metabase. Although this information is not relevant to the conceptual model, the metabase is the only place to store it. Thus, the metamodel structure should be extensible, to provide storage for all necessary information regarding the physical properties of a database.
- **Privacy and security.** As stated previously, this aspect is not relevant to the database conceptual model, but the metabase repository is usually the place to store information on privacy and security rules. The metabase repository itself should be a subject for strong security rules.
- **Extensibility.** The metabase structure and interfaces should be easily extended to support further development and extensions of DBMS functionalities. There are features such as views, constraints, active rules, stored procedures, etc. which could be incorporated into future ODBMS standards and implementations.

The description of the metamodel in the ODMG standard intends to follow this goal. However, in this role the metamodel presented in the ODMG standard is too complex: 31 interfaces, 22 bi-directional associations, 29 inheritance relationships and 64 operations. It is too difficult to understand and use by programmers. Future extensions will cause further growth in the complexity of the metamodel. Methods to access and update a metabase are not described at all. Thus, ODBMS developers must induce the meaning from names and parameters used in the specification, which will probably lead to incompatible (or non-interoperable) solutions. There are many examples showing that the defined methods are not able to fulfill all necessary requests. We conclude that in this role the ODMG standard metamodel is unsatisfactory.

### 2.3 Schema Evolution

This role of metamodel is not discussed explicitly in the ODMG standard. However, interfaces used to define the ODMG metamodel provide the modification operations. Their presence is adequate only in the context of schema evolution. This aspect of database functionality has been present for a long time as one of the main features to be introduced in object-oriented DBMS [1] and its importance is unquestionable. Obviously, the schema evolution problem is not reduced to some combination of simple and sophisticated operations on the schema alone. After changes to a database schema, the corresponding database objects must be reorganized to satisfy the typing

constraints induced by the new schema. Moreover, application programs acting on the database must be altered.

Although the database literature contains over a hundred papers devoted to the problem (e.g. [4,6,9,13,14]), it is far from solved in our opinion. Naive approaches reduce the problem to operations on the metadata repository. This is a minor problem, which can be solved simply (with no research), by removing the old schema and inserting a new schema from scratch. If database application software is designed according to software configuration management (SCM) principles, then the documentation concerning an old and a new schema must be stored in the SCM repository. Hence, storing historical information on previous database schemata in a metadata repository (as postulated by some papers) in the majority of cases is useless. Serious treatment of SCM and software change management excludes ad hoc, undocumented changes in the database schema.

The ODMG solution (as well as many papers devoted to schema evolution) neglects the software configuration and software change management aspects. To effectively support the schema change in larger systems a DBMS should provide features for storing dependency information concerning the schema. This would require new metamodel constructs dedicated to this role. The proposal of dependency-tracking properties in a database metamodel is outside the scope of this paper.

## **2.4 Generic Programming**

As explicitly stated, the ODMG metamodel should have the same role as the Interface Repository of the OMG CORBA [12], which presents some data structures together with operations (collected in interfaces) to interrogate and manipulate the defined IDL interfaces. The primary goal of the Interface Repository of CORBA is dynamic invocations, i.e. generic programming through reflection. This goal is not supported by ODMG. As will be shown, the standard does not define all necessary features.

# **3 Proposed Improvements to a Metamodel**

In this section, we suggest some general directions for improvements of the current standard's metamodel definition that in our opinion would make it flexible and open for future extensions.

## **3.1 Minimality of a Metamodel**

As can be seen, the main problems with the current metamodel definition result from its size and redundancy, making it too complicated for implementation and usage by programmers. Our suggestion is to reduce the number of constructs and in this, there are several options. The most obvious improvement in this direction is the removal of concepts that are redundant or of limited use. For instance, we can postulate to remove the set concept, because the multi-set (bag) covers it and applications of sets are marginal (SQL does not deal with sets but with bags). Another recommendation,

which can considerably simplify the metamodel (as well as a query language), concerns object relativism. It assumes uniform treatment of data elements independently of a data hierarchy level. Thus, differentiating between the concepts of object, attribute and subattribute becomes secondary. Some simplifications can also be expected from the clean definitions of the concepts of interface, type and class.

### 3.2 Flattening a Metamodel

The basic step toward simplifying the metamodel definition concerns flattening its structure. Separate metamodel constructs like *Parameter*, *Interface* or *Attribute* can be replaced with one construct, say *Metaobject*, equipped with additional meta-attribute *kind*, whose values can be strings “parameter”, “interface”, “attribute”, or others, possibly defined in the future; Fig.1.

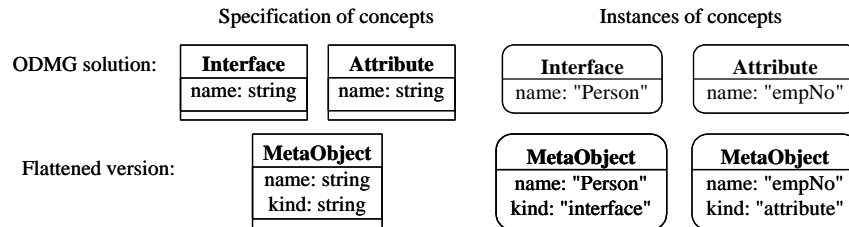


Fig. 1. Original and flattened ODMG concepts

This approach radically reduces the number of concepts that the metadata repository must deal with. The metabase could be limited to only a few constructs, as demonstrated in Fig.2. Despite some shortcomings (e.g. lack of complex and repeating meta-attributes), it seems to be sufficient for the definition of all metamodel concepts. Therefore, the remainder of this paper will refer to it as the *base* of our flattened metamodel proposal.

Flattening the metamodel makes it possible to introduce more generic operations on metadata, thus simplifying them for usage by designers and programmers. It also supports extensibility, as it is easier to augment dictionaries than to modify the structure of meta-interfaces. Such a change could support the run-time performance and maintenance of the metamodel definition.

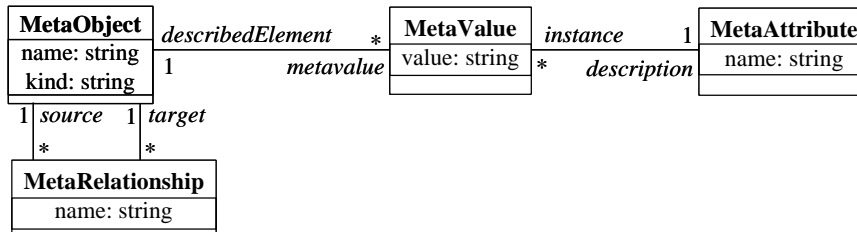


Fig. 2. Concepts of the flattened metamodel

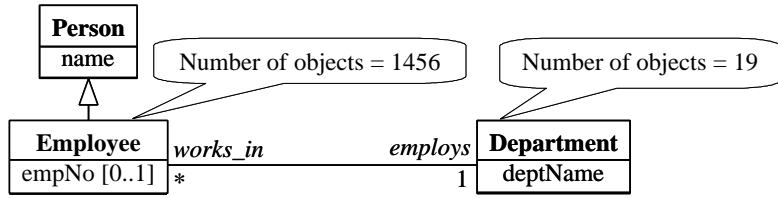


Fig. 3. A sample ODL schema

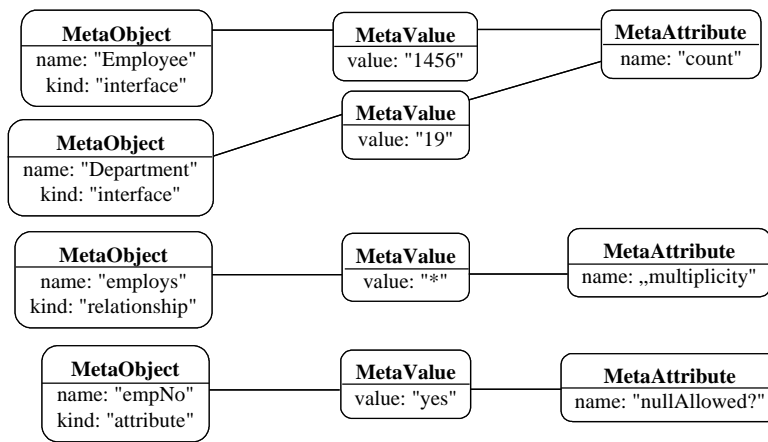


Fig. 4. A metamodel instance: the usage of meta-attributes

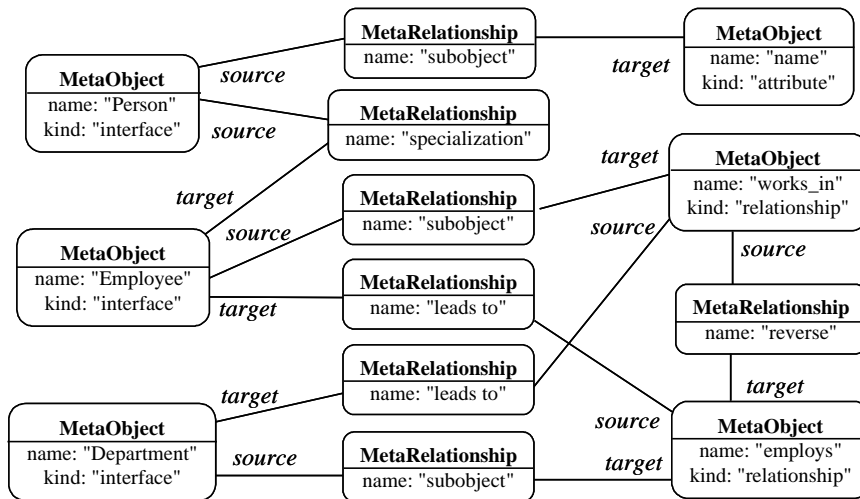


Fig. 5. A metamodel instance: the usage of meta-relationships

Fig.3 presents a sample ODL schema and Fig.4 and Fig.5 present one possible state of the database catalog according to the metamodel presented in Fig.2. Although operations are not present in this example, they can be defined analogously.

Note that we retain the ODMG's abstract concept of *MetaObject*. However, to specify its kind we use attribute values instead of interface specialization. A large part of the presented metadata is used to define appropriate object data model constructs. In order to define a standard metamodel, our flattened metamodel has to be accompanied with additional specifications, which should include:

- Predefined values of the meta-attribute "kind" in the metaclass "MetaObject" (e.g. "interface", "attribute", etc.); they should be collected in an extensible dictionary.
- Predefined values of meta-attributes "name" in metaclasses "MetaAttribute" (e.g. "count") and "MetaRelationship" (e.g. "specialization").
- Constraints defining the allowed combination and context of predefined elements.

### 3.3 Additional Schema Elements and Extensibility

As already suggested, additional information is needed to support data storage. Additional elements may concern information on physical database structure. Some of them (e.g. the number of elements in collections) could be explicitly accessed by application developers, and thus, have to be defined in the standard. Some others, e.g. presence of indices, different kinds of data access statistics, etc., could be the subject of extensions proprietary to a particular ODBMS.

A database metamodel may also provide support for virtual types and their constituent parts. Virtual types represent those types which are not part of the base schema, but are possibly defined to restructure existing types or filter the extents of base types. Virtual types traditionally exist in database views, where (in simple terms) a view is a stored query definition. To retain the semantic information present in the base schema, an object-oriented view should be regarded as a sub-schema, which can comprise all constructs of the base model. Thus a view must be capable of defining multiple virtual classes with heterogeneous relationships connecting classes. The ODMG metamodel provides a mechanism for representing base types, but contains no provision for representing virtual types (and their parts), and this may result in many heterogeneous proposals for ODMG view mechanisms. If views are to be defined for complex ODMG schemas, and these views (and their virtual classes) are to be reused by other views, the storage of a string representation (of a view definition) is not sufficient. An extension to the ODMG metamodel which facilitated the representation of virtual subschemas was presented in [15]. The extension of our metamodel to incorporate views will be treated in a separate paper.

Another example of additional metadata elements is information on ownership and access permissions. Since such mechanisms are built into the DBMS and accessed by applications, appropriate metadata elements could be the subject of standardization.

In contrast to the relational model, type definitions in object systems are separated from data structures. Hence a metamodel repository must store definitions of types/classes/interfaces as distinguishable features connected to meta-information on storage structures.

### 3.4 Support for Reflection

Generic programming through *reflection* requires the following steps:

1. Accessing the metamodel repository to retrieve all data necessary to formulate a dynamic request.
2. Construction of the dynamic request, in form of a parameterized query.
3. Executing the request (with parameters). This assumes the invocation of a special utility, which takes the request as an argument. The result is placed in a data structure specifically prepared for this task. Since a request is usually executed several times, it is desirable to provide a preparation function that stores the optimized request in a convenient run-time format.
4. Utilizing the result. In more complicated cases the type of result is unknown in advance and has to be determined during run time by a special utility that parses the request against the metamodel information.

The four reflection steps are implemented in dynamic SQL (SQL-89 and SQL-92) and in CORBA DII. Although the ODMG standard specifies access to meta-information, thus supporting step 1, it does not provide any support for the subsequent steps (for a detailed discussion see [16]).

Of special interest are the requirements for step 4. For the result returned, it is necessary to construct data structures whose types have to be determined during run-time. A query result type can constitute a complex structure, perhaps different from all types already represented in the schema repository. This structure can refer to types stored in the schema repository. Moreover, it must be inter-mixed (or linked) with sub-values of the request result, because for each atomic or complex sub-element of the result, the programmer must be able to retrieve its type during run time. Hence the metamodel has to guarantee that every separable data item stored in database is connected to information on its type. Construction and utilization of such information presents an essential research problem.

### 3.5 Metamodel Access and Manipulation Language

To simplify the functionality offered by the metamodel and to allow its further extensions, a standard generic set of operations for metadata search and manipulation should be defined. A predefined set of methods is a bad solution. Instead, the interface should be based on generic operations, for instance a query language extended with facilities specific to metadata queries. Such an approach is assumed in [17], where a special metamodel language MetaOQL is proposed. In our opinion, after defining catalogs as object-oriented structures they can be interrogated by a regular OQL-like query language, extended by manipulation capabilities, e.g. as proposed in [20]. Because the structure of the catalogs can be recursive, it is essential to provide corresponding operators in a query/manipulation language, such as transitive closures and/or recursive procedures and views. These operators are not considered for OQL. So far, only the object query language SBQL of the prototype system Loqis [19] fully implements them. Such operators are provided for SQL3/SQL1999 and some variant of them is implemented in the Oracle ORDBMS.

The above suggestions result from the assumed simplicity and minimality of programmer's interface. A similar solution is provided by the SQL-92 standard for relational databases, where catalogs are organized as regular tables accessed via regular SQL. Using the same constructs to access the database and the metamodel repository would not only make it easier for programmers, but would also be advantageous for performance due to utilizing a query optimizer implemented in the corresponding query language.

In case of metadata items that are to be accessed in a number of ways it is critical to provide a fully universal generic interface. Even an extension to the current collection of methods proposed by ODMG cannot guarantee that all requests are available. In summary, this inevitably leads us to the solution, where the metadata repository could be interrogated and processed by a universal query/programming language *a la* PL/SQL of Oracle or SQL3/SQL1999.

## **4 Suggested Simplified Metamodel**

In this section, we present the interrelations among the necessary constructs in a fashion similar to the UML metamodel, discuss the most important features of that structure and finally flatten it to achieve maximum flexibility.

### **4.1 The Base for Metamodel Definition**

Initially we will follow the common four-level approach to metamodeling (see e.g. [5,10,11]), where the entities constituting a system are categorized into four layers: user data, model, metamodel and meta-metamodel. User data is structured according to the definition provided by a model, and the model is defined in terms of a metamodel etc. Fig. 6 outlines the core concepts of a metamodel, which represents the third layer of the system. The highest layer of that architecture – a meta-metamodel could be necessary to provide a formal basis for the definition of other layers, as well as a means of extending the existing set of metamodel concepts. However, in the case of a database metamodel, all concepts must be related both to the appropriate structures of a storage model and the appropriate constructs of a query language (that itself requires a formal definition). This should remove ambiguities, making a separate definition of meta-metamodel unnecessary. Thus, we limit our discussion to the metamodel layer. After describing the conceptual view (Fig. 6), we progress to the flattened form (Fig. 2). As already stated, the flattened metamodel is well prepared for extensions. In fact, despite its simple structure it accumulates the responsibilities specific to both traditional metamodel and a meta-metamodel.

### **4.2 The metamodel**

In Fig.6 we present an exemplary solution defining the core elements of the discussed metamodel. It is focused on the most essential elements of the object data model and, taking into account the different requirements concerning a database schema, it is by

no means complete. Even with such reduced scope, the model becomes quite complex. However, this form makes it convenient to discuss some essential improvements introduced here, as a comparison to the ODMG standard.

All basic metamodel concepts inherit from the *MetaObject* and therefore possess the meta-attribute *name*. The most important branches of this generalization graph are *Property*, which generalizes all the properties owned by *Interface*, and *Type* (described later), which describe any information on database object's structure and constraints. The procedure (method) definition is conventional. It allows for declaring parameters, events and return types in case of a functional procedure. The parameter's *mutability* determines whether it is passed as "input", "output" or "input-output".

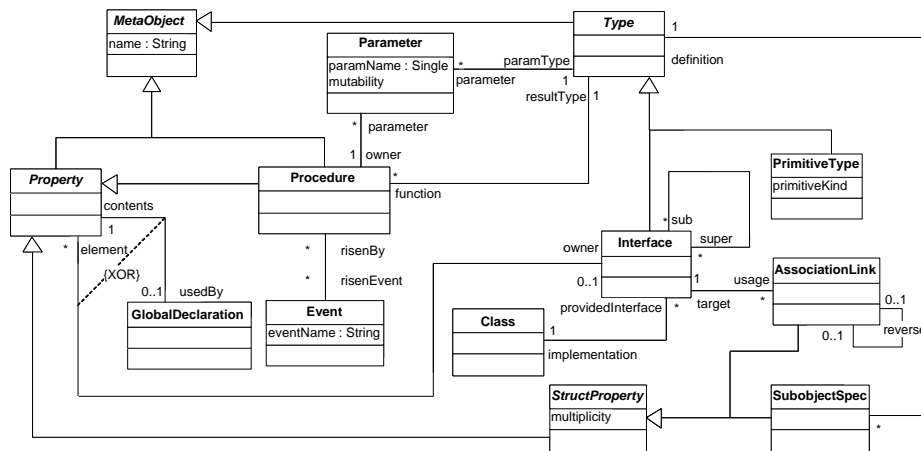


Fig. 6. Conceptual view of the core concepts of the proposed metamodel

Below we comment on the most important features of the presented metamodel, which distinguish it from the ODMG solution.

- **Lack of method declarations.** In contrast to the ODMG definition and similarly to the UML metamodel, there are no method declarations in our metamodel definition. We prefer to rely on a generic query/programming language for the modification or retrieval of the schema information.
- **Information on global properties included in the schema as a separate construct.** For some purposes (e.g. the ownership and security management), the schema has to be aware of its instances. Since we avoid introducing the *extent* concept, we need the means to designate the possible locations of the instances of a given type. Thus the *Property* construct has two roles. When connected with the *GlobalDeclaration* metaobject instead of the *Interface*, it denotes the global variable or procedure rather than a part of an interface specification.
- **No explicit collection types.** With the multiplicity declaration describing associations and sub-attribute declarations, the introduction of the collection concept into the metamodel can be considered redundant. The required properties of a collection can be described by the *multiplicity* (and perhaps also *isOrdered*) attribute value of the *StructProperty*.

- **Application of the terms “Interface”, “Type” and “Class”.** We can describe the *Type* as a constraint concerning the structure of an object, as well as the context of its use. The role of an *Interface* is to provide all the information necessary to properly handle a given object. The typing information remains the central component of an interface definition. Additionally it specifies public structural and behavioral properties, including raised events and possibly other properties. *Class* is an entity providing implementation for interfaces declared in a system.
- **Object relativism.** For both the simplicity and flexibility of a DBMS it is desirable to treat complex and primitive objects in a uniform way. A *Type* concept, serving as a “common denominator” for both the complex objects’ interfaces and primitive types has been introduced. Distinguishing *Subobject Link* from the *Association Link* allows for potentially arbitrarily nested object compositions.

### 4.3 Flattened Metamodel Form

By flattening the metamodel, we move the majority of meta-metadata into the metadata level. The resulting schema (Fig.2) is not only very small in terms of its structure, but also it uses only the simplest concepts in its definition. In this subsection we focus on discussing implications of using such a simplified structure.

The process of mapping the metamodel structure like the one shown in the previous section can be described by the following rules:

- Every concrete entity from the conceptual view of a metamodel is reflected into the separate value of meta-attribute “kind” (see Fig.2) of *MetaObject*.
- Inherited properties and constraints are imported into the set of features connected with a given value of “kind”.
- The meta-attribute “name” (required for every entity of our metamodel) is mapped into the meta-attribute “name” of *MetaObject*.
- Every meta-attribute other than “name” is mapped into the instance of *MetaAttribute* in “flat” metamodel. All instances of *MetaObject* having an appropriate “kind” value are connected (through the *MetaValue* instance) to a single instance of *MetaAttribute* of a given name. *MetaValue* connects exactly one *MetaObject* with exactly one *MetaAttribute* used to describe that *MetaObject*.
- Every association existing in conceptual metamodel is reflected into the separate value the meta-attribute “name” of *MetaRelationship*, and the second, other value, to provide the reverse relationship.<sup>2</sup>

It is now possible to summarize the meaning of the operations that can be performed on the flattened metamodel. Below we iterate through its constructs and describe the meaning of generic operations that can affect them.

- **MetaObject:**
  - Add / remove an instance (the combination of values of “name” and “kind” meta-attribute is unique among the meta-objects) => schema (model) modification;

---

<sup>2</sup> Note the difference in the nature between the meta-attribute “name” of *MetaObject* and the meta-attributes “name” of *MetaAttribute* and *MetaRelationship*. The former are the names defined for a given model, e.g. “Employee”. The latter are determined by a metamodel, e.g. “NoOfElements” or “InheritsFrom”.

- Introduction of a new value of “kind” or its removal => change to the metamodel;
- Add / remove connected *MetaRelationship* instances => schema modification.
- *MetaAttribute*:
  - Add / remove an instance (the values of “name” are unique among *MetaAttributes* describing the *MetaObjects* of a given kind) => change to the metamodel.
- *MetaRelationship*:
  - Add / remove an instance => schema modification;
  - Introduction of a new value of “name” or its removal => change to the metamodel.

As can be seen, due to moving the majority of meta-metadata elements into the metadata level, some of the operations identified above have more significant implications than just schema modification: they affect an established data model.

Another important remark concerns the constraints connected with a given kind of metaobject. The metamodel form presented in Fig.6 requires some well-formed rules that were not explicitly formulated on that diagram. However, in case of the flattened metamodel, such additional constraints become critical, since practically no constraints (like e.g. multiplicities of connected meta-entities) are included into the metamodel structure. Therefore, in addition to the set of predefined values for meta-attributes like “kind” from *MetaObject* or “name” from *MetaAttribute* and *MetaRelationship*, the standard needs to define the constraints specific for each value.

## 5 Conclusions

In this paper, we began by discussing desirable properties of an object metamodel, and assessed the ODMG standard in this respect. The metamodel and associated schema repository are necessary to implement internal operations of a DBMS. Such a repository is also a proper place to store physical data structure information, privacy and security information, and information needed for optimization. An important role of the metadata repository concerns the support for generic programming through reflection. Besides the precise definition of facilities for constructing and executing the dynamic request, special attention should be paid to the problem of the query result metamodel. Another important issue is schema evolution. Providing metadata manipulation features is insufficient. A standard should consider a much broader perspective of this problem in the spirit of change management and SCM state of the art. Some issues of the configuration management require an explicit support in the schema repository and appropriate constructs need to be standardized.

Although the ODMG provides the definition of a metamodel, the solution is incomplete and in many aspects invalid from a practical viewpoint. To make it useful, the metamodel must be simplified, both by reducing redundant concepts and by “flattening” its structure. Such an approach would also simplify possible future extensions to metamodel. It is desirable to define generic access mechanisms to metadata repository, not limiting its functionality to the set of predefined operations.

In this paper, the sketch of such flattened metamodel has been presented. The next step in our work is to introduce the *view* concept into the metamodel definition. Other issues that require more thorough investigation are incorporation of the dynamic object roles mechanism [7], and metamodel elements supporting the SCM.

## References

1. J.Banerjee, H.Chou, J.Garza, W.Kim, D.Woelk, N.Ballou. *Data Model Issues for Object-Oriented Applications*. ACM TOIS, April 1987
2. G.Booch, I.Jacobson, J.Rumbaugh. *The UML User Guide*, Addison-Wesley, 1998
3. R.Cattell, D.Barry. (eds.) *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000
4. K.T.Claypool, J.Jin, E.A.Rundensteiner. *OQL SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework*. In Centre for Advanced Studies Conference, 1998, 108-122
5. R.Geisler, M.Klar and S.Mann. *Precise UML Semantics Through Formal Metamodeling*. Proc. of the OOPSLA'98 Workshop on Formalizing UML, 1998
6. I.A.Goralwalla, D.Szafron, M.T.Özsu, R.J.Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. DKE 28(1), 1998
7. A.Jodlowski, P.Habela, J.Plodzien, K.Subieta. *Dynamic Object Roles in Conceptual Modeling and Databases*. Institute of Computer Science PAS Report 932, Warsaw, Dec. 2001 (submitted for publication)
8. W.Kim. *Observations on the ODMG-93 Proposal for an Object-Oriented Database Language*. ACM SIGMOD Record, 23(1), 1994, 4-9
9. S.-E.Lautemann. *Change Management with Roles*. DASFAA, 1999, 291-300
10. Object Management Group: *Meta Object Facility (MOF) Specification. Version 1.3*, March 2000 [<http://www.omg.org/>]
11. Object Management Group: *Unified Modeling Language (UML) Specification. Version 1.4*, September 2001 [<http://www.omg.org/>]
12. R.Orfali, D.Harkey. *Client/Server Programming with Java and CORBA*, Wiley, 1998
13. R.J.Peters, M.T.Özsu. *An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems*. TODS 22(1), 1997 75-114
14. Y.-G.Ra, E.A.Rundensteiner. *A Transparent Object-Oriented Schema Change Approach Using View Evolution*. ICDE, 1995, 165-172
15. M. Roantree, J. Kennedy, P. Barclay. *Integrating View Schemata Using an Extended Object Definition Language*. Proc. of the 9th International Conference on Cooperative Information Systems, LNCS 2172, pp. 150-162, Springer, 2001
16. M.Roantree, K.Subieta. *Generic Applications for Object-Oriented Databases*. (submitted for publication, 2002)
17. H.Su, K.T.Claypool, E.A.Rundensteiner. *Extending the Object Query Language for Transparent Metadata Access*. *Database Schema Evolution and Meta-Modeling*, 9th Intl. Workshop on Foundations of Models and Languages for Data and Objects, 2000, Springer LNCS 2065, 2001 182-201
18. K.Subieta, M.Missala. *Semantics of Query Languages for the Entity-Relationship Model*. *Entity-Relationship Approach*. Elsevier Science Publishers, 1987, 197-216
19. K.Subieta, M.Missala, K.Anacki. *The LOQIS System, Description and Programmer Manual*. Institute of Computer Science PAS Report 695, 1990
20. K.Subieta. *Object-Oriented Standards. Can ODMG OQL Be Extended to a Programming Language?* Proc. of International Symposium on Cooperative Database Systems, Kyoto, Japan, December 1996. (In) *Cooperative Databases and Applications*, World Scientific, 1997, 459-468