Upgrading a Complex Object DBMS to Full Object-Orientation: a Case Study

H. Dentler, M. Scheurer, F.-J. Schmid

SIEMENS AG, ZFE F2 KOM
Otto-Hahn-Ring 6
D-8000 München 83, West Germany

A. Kotz, B. Schiefer, K. Dittrich[1]

Forschungszentrum Informatik an der Univ. Karlsruhe
Haid-und-Neu-Str. 10-14
D-7500 Karlsruhe 1, West Germany

Abstract

Object-oriented database systems are a promising way to fill the gap between conventional database management systems (DBMS) on one hand and advanced application semantics as well as the world of programming languages on the other. Object-oriented data models differ from traditional models in two main aspects: complex objects are supported instead of flat record structures and objects with predefined behavior replace passive data. Only if both features - structure as well as behavior - are integrated, a DBMS is said to offer full object-orientation. In this paper, we present on the example of the DAMASCUS system an approach to upgrade an existing DBMS with complex objects to support the whole palette of object-oriented features. We show how our current system fits into the overall architecture and which concepts and components have to be added. The resulting DBMS is projected as a data repository for non-standard applications programmed in object-oriented as well as conventional languages. We claim that our approach can be regarded as a general one for non-standard DBMS kernels.

1 Motivation

Since a couple of years, there is general agreement among researchers, developers and users that today's commercial (i.e. relational or network) database systems are less than suitable for handling advanced application semantics ([Kent79], [Sidl80]). At the same time, non-standard applications like CAD/CAM, office automation, artificial intelligence or communication systems are invariably asking for database functionality. The same is true for programming language environments in general. On the one hand, file systems as used as the only persistent data repository so far do not offer anything like physical and logical data independence, recovery, access control or multiuser synchronization. On the other hand, the use of an existing DBMS is awkward and inefficient because of the modelling mismatch between record-oriented data models and data definition and manipulation in programming languages.

As a solution to the problems mentioned, database research has arrived at the idea of object-oriented database systems. In these systems, two directions of development meet: the enhancement of DBMSs by semantic data model concepts ([PM88]) and the equipping of programming languages - especially those following the object-oriented paradigm - with persistent object storage ([TN88], [MSOP86]). Object-oriented database systems try to combine the best of the two worlds.

When filling in the gap between database systems and advanced applications or programming languages respectively, two aspects have to be considered, namely which object structures to provide and how to work with them. As mentioned before, flat record structures are not adequate for application programs dealing with objects of high complexity. Furthermore, application-specific operations should be definable for a database object, defining its dynamic behavior. The latter also includes further features like inheritance and encapsulation by abstract data types. To differentiate between DBMSs which support only one of the two aspects, [Ditt88] introduces the terms structural and behavioral object-orientation. Only when both are supported, **full object-orientation** is obtained.

In this paper, we present an approach to build a fully object-oriented DBMS by upgrading a database system which supports complex object structures. Our starting point is the non-standard database system DAMASCUS[2]. We show how the existing DBMS can be integrated into an enhanced architecture, which modelling concepts and system layers have to be added, and how the final system is going to support various programming languages.

Several evident advantages are gained by retrofitting full object-orientation to an existing complex object DBMS instead of a completely new development. The new system can build upon a stable database kernel with advanced features well-suited for the planned applications. The evolutionary development makes use of existing system modules and interfaces, thus considerably saving implementation effort and providing upward compatibility. The further use of existing application programs of the old DBMS is possible by making intermediate interfaces accessible. In this

[1]now at: Universität Zürich, Institut für Informatik, Winterthurerstr. 190, CH-8057 Zürich, Switzerland.

[2]DAMASCUS (**d**atabase **m**angement **s**ystem for **CA**D using **U**NIX **s**tations) has been developed in a cooperation project between SIEMENS, Munich, and Forschungszentrum Informatik, Karlsruhe.

respect, our way of proceeding differs from other object-oriented DBMSs like POSTGRES [RS87], IRIS[Fish87] etc. (which are similar in their features at the user interface) that were developed from scratch and did not follow a stepwise upgrading approach from structural to full object-orientation.

We claim that the approach can be regarded as a general one for non-standard DBMS kernels and that DAMASCUS is just one example for this class of systems. Note, however, that the same is not true for an arbitrary DBMS, e.g. a relational one, but that the system must already provide a number of advanced object-oriented facilities.

The paper is organized as follows. Chapter 2 will give a short overview of the DAMASCUS features. In chapter 3 we will show which requirements are not met so far and which extensions have to be made to the system architecture. The new conceptual issues are treated in chapter 4, with the emphasis on data modelling. Chapter 5 gives an outlook on some of the open questions that have to be solved in the future.

2 What we have: the DAMASCUS system

The DAMASCUS DBMS ([DKM85], [DKM87]) has been developed and implemented as a research prototype and was mainly thought for the application in VLSI design environments. From the beginning, a two-level architectural approach has been followed: the system consists of the DBMS-kernel and a design management layer on top of it. The kernel is a general non-standard DBMS supporting complex objects, whereas the design management layer supports (VLSI-)CAD specific features. In the sequel, we will only deal with the kernel DBMS.

The data model of the DAMASCUS-kernel, called the IODM (internal object data model), features an entity-relationship model extended by, among others, complex objects. The basic units for describing the universe of discourse are objects and n-ary relationships between them. The facilities for complex structures include the following:

1. Treelike object hierarchies as well as object nets of arbitrary complexity can be constructed. The first is done via so-called built-in objects, the second via references.

2. Each object is described as an aggregation of attributes. Apart from simple attribute domains like numbers, text or boolean, several type constructors for complex domains are applicable. They allow for the definition of sets, lists, tuples and vectors (i.e. arrays). Type constructors can be applied in arbitrary combination resulting in sets of vectors, lists of lists of tuples etc. Built-in objects and references are also realized via attributes and can be combined with the domain constructors. A special type constructor 'union' is offered to generalize the type of subobjects.

3. Long fields, i.e. unstructured byte strings of arbitrary length, are supported by the attribute domain 'bytes'. Long fields are treated like random access files.

4. Every object is given a system-generated unique identifier, its surrogate or 'OID'. The surrogate can be used to retrieve the object as well as to refer to it from other objects. Of course, user-defined key attributes for object access are also supported.

5. Objects may be declared to have versions which will be ordered in a version graph. There is a static part of the object common to all its versions and a variant part pertaining to each version individually. The version graph may be declared as linear, treelike or acyclic net, thus allowing for historic versions, variants and merging of variants. Delta storage of versions is supported as an option.

6. Relationships may be n-ary and possess attributes of their own. Several relationship tuples may be grouped and the group given a user-defined key.

In addition, the model provides a database concept, explicit clustering of objects and an integrated data dictionary. The IODM is strongly typed, i.e. every object (relationship) is considered an instance of an objecttype (relationshiptype) defined in the schema by means of the data definition language (DDL).

As a simple example for the use of the IODM, consider graphical symbols which consist of elementary figures like polygons, circles and ellipses, as well as recursively of other symbols which may be scaled and positioned arbitrarily. Furthermore, every symbol has a name serving as its key and a long field for its pixel representation. One of the elementary objecttypes used in GRAPH_SYMBOL - POLYGON - is detailled further; its vertices are modelled as a list of built-in objects of type POINT. The example in its DDL syntax is given in fig. 1.

The DAMASCUS data manipulation language (DML) is designed as a call-interface for PASCAL programs. The DML operators are generic, i.e they equally apply to objects or relationships of any type. Complex objects as a whole as well as parts or attributes of objects may be manipulated individually. For instance, you can delete or copy an object with all ist built-in subobjects. For constructed attributes like sets or lists as well as for long fields, there are specific operations such as scanning, deleting of elements etc. An SQL-style, set-oriented descriptive query language is currently being developed for DAMASCUS.

```
objecttype POINT =
   structure
      x_coord, y_coord : real;
   end;

objecttype POLYGON =
   structure
      vertices  :  list of POINT;
      area      :  real;
      filled    :  boolean;
   end;

objecttype GRAPH_SYMBOL =
   structure
      name             :  text (20);
      element_figures  :  set of union (POLYGON,CIRCLE,
                                              ELLIPSE);
      symbols_used     :  set of structure
                              the_symbol   :  ref SYMBOL;
                              scale_factor :  vector (2) of int;
                              position     :  vector (2) of real;
                          end;
      pixel_represent  :  bytes;
   key name
   end;
```

Fig. 1: Modelling example in the DAMASCUS DDL

DAMASCUS has been built from scratch, i.e. all its layers have been specifically designed and implemented for non-standard applications on UNIX workstations. On top of the operating system - but bypassing the UNIX file system for efficiency reasons - the storage manager has been realized, comprising secondary storage and buffer management, various access path methods, and a manager for variable length records. The storage system forms the basis for the complex object manager that provides the IODM definition and manipulation facilities. At present, we are working on components for distribution, concurrency, recovery and an event-trigger-mechanism (ETM) for rule support [KDM88]. Fig.2 sketches the DAMASCUS kernel architecture. (In fig. 2 and all following figures, the names of the system layers are given in the boxes representing their interface.)

As has been pointed out, the DAMASCUS model supports complex object structures and relationships with the corresponding generic operations. In the present version, however, it does not allow for application-specific operations or encapsulation by abstract data types. The current DAMASCUS DBMS is thus a structurally object-oriented system. The next section will show which requirements remain unsatisfied and what is needed to get a full-fledged object-oriented DBMS.

3 What we need: full object-orientation

In the application world, an object is not only characterized by structure and value, but also by the operations or methods that can be used to manipulate it, i.e. its behavior. The traditional approach to separate data structures (in the DBMS) from data dynamics (in the application programs) leads to severe consistency problems and code redundancy as to method implementation.

As an example, consider a geometric figures database as needed in CAD/CAM applications. Each figure is represented by the coordinate values of its vertices. The figures have to be manipulated by operators like rotate, scale, translate etc. Each operation must preserve the topological as well as certain geometrical properties of the figures. These properties can easily be violated by direct manipulation of the figures' coordinate attributes.

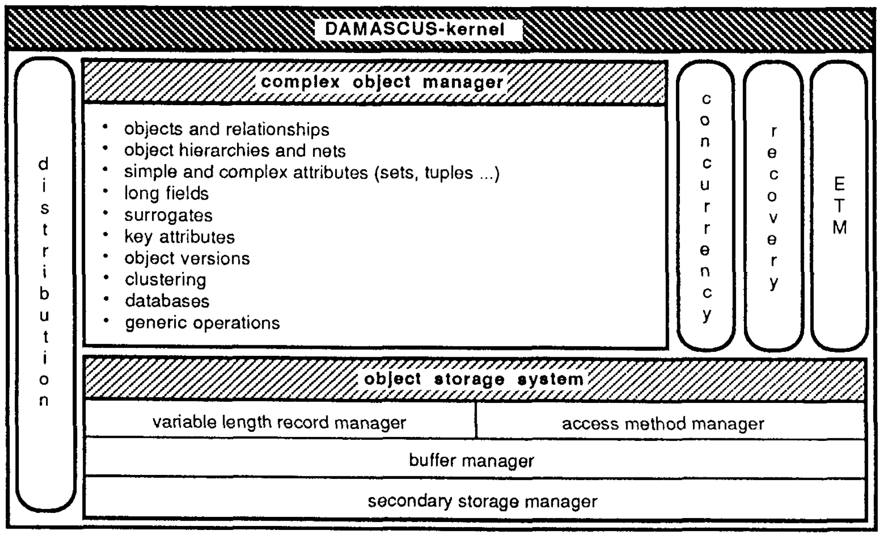

Fig. 2: The DAMASCUS kernel architecture

Therefore, the transformations should be definable as the operational interface of the type *figure*. The generic operations to modify the attributes should not be directly applicable by end users, but only by the method implementor.

To meet these requirements, the DBMS has to allow that:

- application-specific operations can be defined (for objects as well as for attribute domains),
- the direct application of the generic operators can be restricted or even totally excluded.

When the use of generic operations is prohibited, encapsulation of objects in the sense of abstract data types (ADT's) is achieved, resulting in better consistency control and implementation independence. We made first experiences in this area by the specification of CADIF, the interface of the design object management system CADBASE, as an ADT ([LSW89], [SLW89]).

Data models fulfilling the above requirements provide so-called behavioral object-orientation. By combining structural object features with object behavior and the inheritance mechanisms known from object-oriented programming languages, we arrive at full object-orientation. Type hierarchies with inheritance are a means to model generalization or specialization ('IS_A semantics'). Inheritance supports property transfer from supertype to subtype, thus reducing redundancy of definition and implementation, supporting consistency and allowing for stepwise refinement. For example, the transformation methods mentioned above (or at least their interfaces) may be defined for a type *polygon* and inherited by all specialized polynomial types like *triangle*, *rectangle* etc.

In the DAMASCUS system, the structural features are already supported, while object behavior and inheritance are still lacking. System components providing these capabilities have to be added. Fig.3 shows how the DAMASCUS system can be embedded into the overall architecture of a system fulfilling the whole palette of object-oriented requirements. In the following this system will be called the **object management system**. Its main layers are the *object administration kernel* and the *general object manager*.

Within the *object administration kernel* , the shaded parts are already covered by the existing DBMS. An essential part of the *object method manager* has to be added, because only the generic methods are already supplied. The method manager makes use of the *object structure manager* for method description and storage.

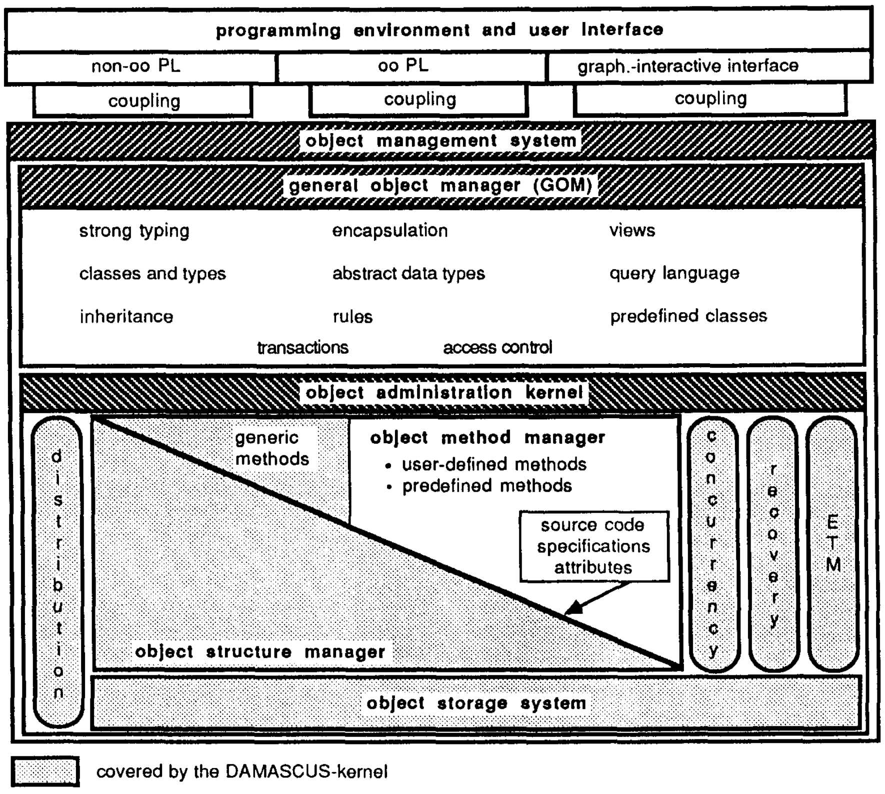

Fig. 3: Object management system integrating the DAMASCUS kernel

In the *general object manager*, the higher-level object-oriented concepts like classes, inheritance and encapsulation in abstract data types are realized. Within the extended framework, several important DBMS features have to be redefined as e.g. query language, views, consistency and derivation rules. Further properties like object-oriented access control and advanced transaction concepts (long transactions, nested transactions) have to be included. By predefined classes, special semantics and system functionality can be supported. The *object management system* may be coupled to object-oriented and conventional programming languages (ooPL and non-oo PL) as well as to an interactive surface with graphical features.

The next section will give an overview on the concepts of the data model provided by the object management system as well as detail the architecture of the general object manager.

4 How we do it: data model and architecture

One of the fundamental questions to be solved when designing the object management system is the coupling with programming languages. The programming interface has to serve two purposes: method implementation using the structures predefined in the DBMS and application programming making use of the methods. In both cases the relationship between programming language data structures ("temporary objects") and database objects ("persistent objects") has to be defined. For the object structures, definition and manipulation facilities have to be provided. There are three ways to solve the interface problem:

a) An existing object-oriented programming language can be extended by the necessary database features (e.g. VBASE+ which extends C++ [Onto88]). The objects of the programming language may directly become database objects, the structural definition and manipulation features are taken from the language.

b) A completely new language is designed (like e.g. OOPL [Schl88]). The same kind of objects are used for programming as well as in the DBMS.

c) Object-oriented data definition concepts are combined with various existing programming languages for method implementation and application programming (an approach chosen e.g. in O_2 [Banc88]). The DBMS objects are made visible in the programming language. The programming language's own data structures are, if at all, only used as temporary variables (e.g. to implement loop counters for algorithm programming).

We agree with the designers of O_2 that it is very useful for reasons of acceptance to be language independent. Assuming on the other hand that C++ [Stro85] has a good chance to become a quasi-standard for object-oriented programming languages, it seems to be a promising way to build a C++ - DBMS. In our approach, we try to combine both advantages. We design special language features for data definition and manipulation, whereas the methods and applications may be implemented by the user in his favourite language. The language designed for data handling will integrate the C++ class concept, so that if the user's programming language is C++, too, the whole system can be considered an extended C++ - DBMS.

In this chapter, we will introduce our data model and give a short outlook on the system's architecture.

4.1 Classes and types

In our approach, we follow the strong typing paradigm as e.g. in EIFFEL [Meye88] instead of a SMALLTALK-like philosophy [GR83]. Our notions of type and class are as follows: a type is the description of all properties, structural as well as behavioral, that are common to all its potential instances. A class is a type with an explicit operation to create instances. Instances of classes are called 'objects'. As a consequence of this notion, attribute values like e.g. numbers or characters will not be considered objects (in contrast to e.g. the SMALLTALK-approach) and can only be used as dependent components of class instances. Dependent types may either be predefined (with generic operations) or user-defined (with application-specific methods).

A type definition comprises the specification, the methods applicable to its instances and the attributes used for their internal representation. The ***specification*** of a type contains the following features: type invariants, consistency constraints, derivation rules, pre- and post-conditions of methods, and the method interfaces themselves. ***Methods*** are either functions or procedures that may be implemented in the user's preferred programming language. ***Attributes*** may be of simple types like integer, boolean and string, of composed types built by constructors like sets, lists and tuples, or of arbitrary user-defined types.

Another important kind of attributes are typed references to objects. We distinguish two kinds of references: general references and compositional references. ***General references*** represent mere pointers from one object to another. Operations on a referencing object have no influence on the referenced objects. ***Compositional references*** have a more elaborate semantics. The referenced object is treated as an integral part of the referencing object (it is 'built in' there). There may be operations with 'deep' effect which apply recursively to an object and all its strongly referenced subobjects, provided that a corresponding operation exists for the subobject. Some 'deep' operations will be predefined for all objects. If e.g. the referencing object is deleted, the referenced objects will also disappear (vice versa, the deletion of the referenced object will have no influence on the referencing object). The user has the choice whether system support for referential integrity should be supplied or not. To manage symmetric references that will automatically be removed, if one of the participating objects is removed, we provide the concept of ***relationships***, which we present later on.

The interface of a type contains all the specification parts - especially the method headers - that are visible and may be used from outside. Attributes may be explicitly declared 'public', which is regarded equivalent to the declaration of canonical methods to read and set their values.

The user may also define generic types like LIST[T] or STACK[T]. On creation of an instance of such a generic type, the parameter T has to be specified.

4.2 Inheritance and class hierarchy

Classes may be arranged in a hierarchy for which an inheritance mechanism is supported. By this facility, the 'IS_A semantics' can be modelled, already implemented methods may be reused, and redundancy is minimized.

In our DBMS, there is a predefined class OBJECT which forms the root of the class hierarchy. All the other classes, whether system- or user-defined, are direct or indirect heirs of this class. OBJECT contains the methods applicable to any object, e.g. operations to retrieve general information about an object, generic methods for reading, updating or printing objects etc.

We had to decide between single inheritance, where every class may have at most one direct ancestor, and multiple inheritance, where each class may inherit from an arbitrary number of classes. To support complex engineering applications in which classes may be specialized under various aspects (logical, physical, geometric etc.), multiple inheritance proves indispensable for our model. The intrinsic problem with multiple inheritance is the naming conflict arising when a class inherits identically named methods from different ancestors. For the resolution of these conflicts, the following strategy has been chosen: if the conflicting methods have been originally defined in some common root and no ambiguous redefinitions on the different paths have occurred, the conflict is resolved automatically. Otherwise, the definer of the new class must resolve the conflict explicitly by renaming (at least one) of the methods.

Methods which are inherited from an ancestor class may not only be renamed, but also overwritten, i.e. reimplemented, in the subclass. By this feature, a general algorithm may be replaced by a more specialized one (for example the area can be calculated more efficiently for a rectangle than for a general polygon). Inherited attributes may also be redefined in the sense of specialization, i.e. their domain may be converted to a more restricted one (as for example integer instead of real). Further specifications of the ancestor class like class invariants, consistency constraints or postconditions of methods may also be restricted in the inheriting class.

To support a top down development, we think it useful to offer deferred (or abstract) classes. These classes may contain just the interfaces of methods the implementation of which is deferred to the inheriting classes. Of course, for a deferred class, no instances may be created.

As a consequence of the class hierarchy and the inheritance mechanism with method overwriting, the data model language supports *polymorphism* and *dynamic binding* ([CW85]). Although strong typing is provided in the language, static type checking is not possible whenever the class hierarchy is involved. Variables and attributes referencing objects of a given type may be polymorphic, i.e. they may refer to instances of subtypes alternatively. For methods that have been overwritten, the currently applicable implementation has to be selected dynamically at runtime. Appropriate mechanisms are supplied by the object method manager of the administration kernel.

4.3 Classes and objects

For every class, an arbitrary number of objects may be created. Each object is given a unique identifier (surrogate), generated by the DBMS. In addition, the user may define key attributes and object names for personal use. These concepts receive special support by the DBMS.

As usual in DBMS terminology, we use a class not only as the notation for the type description, but also for refering to its extension (i.e. the set of all instances of a given class). This is advantageous in that the user needs not create a new class, for example "set_of_objects_of_class", if he wants to consider all objects of a class as a whole.

Due to the IS_A-semantics of the inheritance hierarchy, we choose a subset semantics for the extensions of the participating classes. This means that a member of the extension of a class also belongs to the extensions of all its ancestor classes. For instance, the objects of a class *triangle* which is a heir of the class *polygon* are elements of the extension of *polygon*, too. A user is allowed to change an object's class membership dynamically from a class to a more specialized one, if the constraints of the heir class are satisfied. Automatic movement of objects along the hierarchy is not considered desirable. In our example, this means that a *polygon* with three corners satisfying all constraints for the class *triangle* will not become a member of the extension of *triangle*, unless it is explicitly moved there.

Most object-oriented systems only provide the concept of ***instance properties*** in their class definitions. These are properties dedicated to single instances, e.g. the methods applicable to an instance or the attributes whose value may vary from one instance to the other. To define methods applicable to the extension of a class as a whole, we additionally provide the concept of ***class properties***. An example for a public class attribute often required is the number of instances of the class. The most important class method needed in every class (except deferred classes) is the create_object operator, which may provide initial values for the attributes of newly created objects, individually for every class. Some basic class properties are located at the top of the inheritance hierarchie, as class methods of the superclass OBJECT.

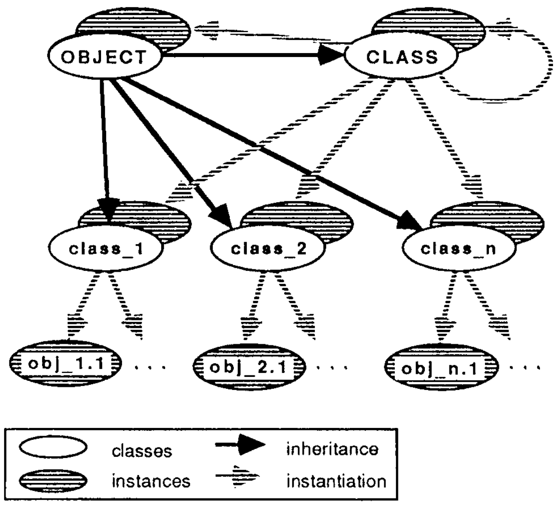

Fig. 4: Relationships between classes and objects

To meet the important requirements of engineering applications with respect to schema modifications and extensions, we provide a metaclass CLASS whose instances are representatives of *all* currently available classes. Like every class, CLASS inherits the properties of OBJECT, so the conventional operations on instances may be used to express schema modifications. The modification of an instance of CLASS will of course have far-reaching consequences for the corresponding class that will not be discussed here.

The existence of CLASS is also justified by the necessity to formulate queries involving several classes. The method QUERY will therefore be considered as a class method of CLASS. By representing CLASS as an instance of its own extension, the user is enabled in a natural manner to formulate queries involving metadata, e.g. select the number of classes, the names of all classes, the attributes of a given class etc. A graphical visualization of the relationship between classes and objects is given in fig. 4.

4.4 Predefined classes

To simplify schema design, there are more predefined classes than just CLASS and OBJECT provided. Some of them are generic classes like stack, list, tree, etc. that may be added to the schema if needed, in order to reduce notational efforts and provide the user with frequently needed abstract data types. Others are classes with a very special, powerful system supported semantics.

The most prominent of them is the class RELATION which can be used to model relationships between objects - a widespread feature in the database world, but yet unusual in the context of object oriented programming. Relationships are a powerful modelling concept to express associations between an arbitrary number of objects and to provide describing attributes for these associations. Among other features, relationships are automatically deleted, if one of the participating objects is removed. The user will not have to take care of them when implementing the delete operations of his object classes.

Another important predefined class in the area of engineering applications is the class VERSIONED_OBJECT which provides mechanisms for efficient version handling. A subclass *su* of class VERSIONED_OBJECT inherits the operations for version management like create_new_version, delete_version, get_version_nr, find_next_version, etc. Note that an object of class *su* represents a generic object with all its actually existing versions; every individual version is itself an object in its own right. A similar concept may be found in [Zdon86].

4.5 Realization

To realize the concepts of the model described in chapter 4, the description of classes and types as well as the object instances themselves have to be represented by means of DAMASCUS concepts. The internal structure of instances will be mapped directly to IODM structures. Thus, references, relationships, versions etc. are easily supported. The metainformation about instances, i.e. descriptions of classes, inheritance hierarchy etc. will also be represented by primary objects and relationships of the IODM. In this respect, DAMASCUS serves as a kind of data dictionary for the general object manager (GOM).

Fig. 5 shows the complete system architecture giving details of the general object manager. The GOM realizes the data model described above, including further features like access control and transactions, which are not treated in this paper. The administration kernel, integrating the DAMASCUS system, corresponds to fig. 3. It sets the GOM free from all tasks concerning complex object storage and access as well as method binding and execution.

The GOM consists of a definition and a manipulation part. The definition part allows for the static declaration of classes (comprizing methods, rules and storage structure options). Sets of classes are analysed and compiled by special tools, the DDL compiler and the method compiler. Beside the utilities for user-defined classes, the GOM's definition part provides the predefined classes, including OBJECT and CLASS.

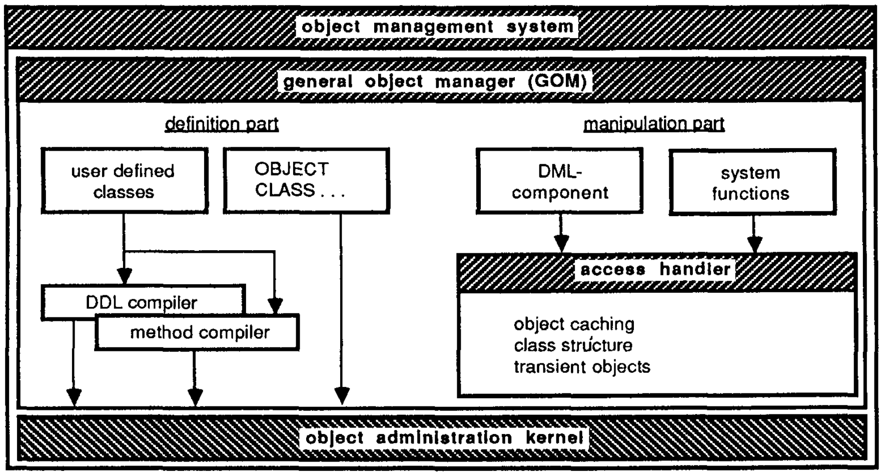

Fig. 5: Architecture of the general object manager

In contrast to the static features of the definition part, the manipulation part covers the dynamic functionality at runtime. The modification and evaluation of data is handled by the DML component which selects the objects and the methods to be applied. As classes are considered instances of a metaclass and thus are objects of their own, they may be modified and added via the manipulation part, too. System functions like transaction definition, tuning or granting of access rights are also provided by the manipulation part in an object-oriented manner.

The access handler forms the link between the administration kernel and the runtime system. It does the mapping from persistent object storage to main memory data structures and vice versa. Its main tasks are object caching, administration of main memory access paths, transient object mangement and maintenance of information about the class structure.

5. Conclusion and outlook

In this paper, we have presented a case study of how to extend an existing DBMS kernel supporting complex objects with generic operations - a structurally object-oriented system - to include the complete palette of object-oriented features. Our approach, studied at the example of the DAMASCUS DBMS, has the general advantages of an evolutionary development, namely economic use of already available components and upward compatibility.

Within the overall system architecture projected, the persistent storage facilities, a complex object manager, as well as the generic part of the method manager are already covered. The existing DBMS, extended by some features for the management of application-specific methods, can be used as an object administration kernel. On top of this, a general object manager realizes the higher-level concepts like classes, encapsulation, inheritance etc., providing a coupling mechanism to various programming languages.

During our case study, we came across several open questions that we think worthwhile to be treated in the future. We would like to conclude the paper by sketching the problems concerning schemas and schema evolution, which might be supported by the basic schema and view facilities of DAMASCUS.

The design of an object-oriented DBMS rises new questions as to the elaboration of the three classical schema levels (conceptual, external and internal). As to the conceptual schema, it has to be decided whether this notion is still relevant in a system that may be regarded as a set of independent object classes. Of course, the set of all existing classes might be regarded as 'the' conceptual schema. On the other hand, for reasons of managing large disjunct amounts of heterogeneous data, it might be useful to administer more than one conceptual schema, for example by a special metaclass SCHEMA. The DAMASCUS system already allows for an arbitrary number of schemas.

The issue of schema evolution has also to be revised in the object-oriented context. Which kinds of modifications should be supported (just adding new classes, adding/modifying methods and attributes, flattening the class hierarchy)? If a class is modified, what should happen to its already existing instances (keep several schema versions, automatic/user-driven adaptation of objects etc.)?

In an object-oriented system, an external schema - describing how data are to be viewed by an application - can be considered a set of classes derived from the original classes. As the object-oriented paradigm supports encapsulation, the security aspects of views are assured in a natural manner. A view class may either be derived directly from one class, optionally selecting methods and attributes by a special public clause, or as the result of a query formulated in the usual query language. It has to be investigated which views have to be treated as virtual classes with transient objects only, which objects constitute the extension of a view, and what kind of manipulations (read only, modify, delete etc.) are applicable to the instances. The basic question is which possibilities for deriving views should be allowed (even rearrangement of references within complex objects, restructuring of the inheritance hierarchy?) and how they may be mapped to the DAMASCUS view concept.

Literature

[Banc88] F. Bancilhon et al.: *The Design and Implementation of O_2, an Object-Oriented Database System.* In [Ditt88].

[CW85] L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Vol. 17, No. 4, pp. 471-522, Dec. 1985.

[Ditt88] K.R. Dittrich: *Advances in Object-Oriented Database Systems.* Proc. of the 2nd Int. Workshop on Object-Oriented Systems, Lecture Notes in Computer Science, Vol. 334, Springer 1988.

[DKM85] K.R. Dittrich, A.M. Kotz, J.A. Mülle: *A Multilevel Approach to Design Database Systems and its Basic Mechanisms.* Proc. IEEE COMPINT, Montreal 1985.

[DKM87] K.R. Dittrich, A.M. Kotz, J.A. Mülle: *Database Support fcr VLSI Design: The DAMASCUS System.* In: M.H. Ungerer (ed.): CAD-Schnittstellen und Datentransferformate im Elektronik-Bereich, Springer 1987.

[Fish87] D.H. Fishman et al.: *Iris: an Object-Oriented Database Management System.* TOOIS 5(1987)1, pp.48-69.

[GR83] A. Goldberg, D. Robson: *SMALLTALK80: The Language and its Implementation.* Addison-Wesley, May 1983.

[Kent79] W. Kent: *Limitations of Record-Based Information Models.* ACM Transactions on Database Systems, Vol.4, No.1, 1979, pp.107-131.

[KDM88] A.M. Kotz, K.R. Dittrich, J.A. Mülle: *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism.* Proc. EDBT, Venice 1988, LNCS Vol. 303, Springer.

[LSW89] J. Loers, F.J. Schmid, W. Wenderoth: *CADBASE - a Database System for Object and Version Management in CAE-Applications.* Informatik Fachberichte No. 204, Springer Verlag 1989 (in german).

[Meye88] B. Meyer: *Object-Oriented Software Construction.* Prentice Hall Int. Series in Computer Science1988.

[MSOP86] D. Maier, J. Stein, A. Otis, A. Purdy: *Development of an Object-Oriented DBMS.* Proc. OOPSLA'86.

[Onto88] Ontologic Inc.: *Vbase+ - Object Database for C++ - Functional Specification.* Bilerica, Dec. 1988.

[PM88] J. Peckham, F. Maryanski: *Semantic Data Models.* ACM Computing Surveys, Vol.20, No.3, Sept. 1988, pp.153-190.

[RS87] L.A. Rowe, M.R. Stonebraker: *The POSTGRES Data Model.* Proc. VLDB 13, 1987, pp.83-96.

[Sidl80] T.W. Sidle: *Weaknesses of Commercial Database Management Systems in Engineering Applications.* Proc. Design Automation Conf., Minneapolis, Vol.17, June 1980, pp.57-61.

[Schl88] G. Schlageter et al.: *OOPS - an Object-Oriented Programming System with Integrated Data Management Facility.* Proc. 4th Int. Conf. on Data Engineering, 1988, pp.118-125.

[Stro85] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley 1985.

[SLW89] F.J. Schmid, J. Loers, W. Wenderoth: *CADBASE - an Object-Oriented Database System for CAE-Applications.* VDI Berichte No. 723, 1989 (in german).

[TN88] D.C. Tsichritzis, O.M. Nierstrasz: *Fitting Round Objects into Square Databases.* Proc. ECOOP, Oslo 1988, Springer Verlag.

[Zdon86] S.B. Zdonik: *Version Management in an Object-Oriented Database.* Proc. of an Intl. Workshop on Advanced Programming Environments, Trondheim 1986, Springer Verlag.