LOGICAL DATA INDEPENDENCE AND MODULARITY THROUGH VIEWS IN OODBMS

Michael Dobrovnik and Johann Eder

Institut für Informatik
Universität Klagenfurt
Klagenfurt,Austria
email:{michi,eder}@ifi.uni-klu.ac.at

ABSTRACT

Views in OODBMS are a concept which is heavily opted for by key database people but rather seldomly implemented in commercial systems. They are an important mechanism to modularize information systems and to gain ogical data independence. This paper presents a concept for updateable views in OODBMS which separates application programs from the conceptual schema by introducing a layer of external schemas.Our approach supports all the traditional purposes of views and is based on a clear distinction of and division between externaland conceptual schemas. The external schemas can be used much in the same way by applications as a conceptual schema could be used. In particular, special attention is given to dynamic binding and invariant (steady) method resolution with respect to static types. This method steadiness ensures that the most special method is found even across schema boundaries. We discuss some aspects of views in OODBMS, concentrating on coupling characteristics, change transparency and security, and we make brief architectural considerations for the implementation of a view system.

1 INTRODUCTION

Concepts for the support of views and external schemas in object oriented database management systems are highly desired (Atkinson 1989, Bancilhon 1990, Kim 1994, Kotz-Dittrich 1995). Nevertheless, there is no commercial implementation available at this point of time (Meier 1995, Zand 1995, Kim 1995b), although several proposals appeared in the scientific literature in the last years (Abiteboul 1991, Barclay 1993, Bertino 1992, Busse 1995, dos Santos 1994, dos Santos 1995, Geppert 1993, Heiler 1988, Heiler 1990, Heuer 1990, Heuer 1991, Heuer 1993, Ishikawa 1992, Jungklaus 1991, Kifer 1992, Kim 1995, Heuhold 1988, Rundensteiner 1992, Scholl 1990, Scholl 1991, Schiefer 1993, Schiefer 1993b, Shilling 1989, Tanaka 1988). For a comparison of the approaches, the reader is referred to Dobrovnik 1995, Geppert 1992, Kuno 1993 and Motschnig 1995.

One reason for this gap between theory and practice surely is the rather preliminary and partial nature of the proposals (especially the early ones), which is quite understandable when one takes into account the dissens on much more fundamental issues in the OODB community.

A major initiative to gain a common understanding of OODBMS and a basic foundation for the implementation is the effort of the ODMG consortium. But up to now, there is no treatment of views in the published proposals Cattell 1993, Cattell 1994 of the group.

The greater power and flexibility of object oriented data models compared to the relational model demands for a much more sophisticated view mechanism rather than a predefined query over some relations (Kim 1995, Motschnig 1995). On the level of single types and classes, views have to allow for *type and class restructuring* which permits to build customized types and support structural as well as behavioral differences.

On the schema level the view system has to provide mechanisms for dealing with *inheritance* and *aggregation* relationships (*schema restructuring*). A clear concept is needed how the elements of the conceptual schema and the external schema interact. This includes the derivation of external schema elements, the mapping of objects (instances) between the schemas, and the adequate treatmentof object identifiers across schema boundaries. For the dynamics the interaction of methods from different schemas plays an important role. Late binding and name resolution have to be extended to external schemas and possibilities for updates through views have to be developed.

This paper gives an overview of a concept for external schemas in OODBMS, which takes into account all the before mentioned issues. We have been developing this concept over the past years (Dobrovnik 1993a, Dobrovnik 1993b, Dobrovnik 1994 and Dobrovnik 1995). The most important contributions of this approach are the clear separation of external and conceptual schemas, the consistent extension of method resolution and subtyping to external schemas and full fledged solution of the view-update problem. In this paper we show how this concept can be used to achieve generally requested design goals for object-oriented databases and application systems.

2 VIEWS IN eXoT/C[1]

The proposal explicitly handles whole external schemas, and not single and isolated views. It provides external type constructors which allow one to derive schema elements of the external schemas on the basis of schema elements of the conceptual schema. In particular, not only support for structural derivation is provided, the approach deals quite extensively with behavioral aspects such as method resolution and method steadiness.

In the following, we give a short sketch of our data model, present the idea of the external schemas and describe the derivation operations we provide.

2.1 Data Model and Conceptual Schema

Our data model is somewhat generic, since we tried to make our view concept easily adaptable to existing systems and emerging standards (ODMG). From a theoretical point of view, we follow the principles of Beeri 1990, Beeri 1992, while also keeping an eye on some notable industrial efforts as in Deux 1991 and Cattell 1994.

During the following explanations, the reader is referred to the example given in Fig. 1, where (a part of) a conceptual schema `Research` is defined. We distinguish between extensional and intensional concepts, a schema in our data model consists of a set of types and a set of containers. The types describe the structural and behavioral aspects of the objects and values. We provide some atomic value types (`boolean`, `integer`, `string`, ...) and an atomic object type (`Object`).

The type constructors `set` and `tuple` can be orthogonally applied to types to build set valued and tuple valued structured value types. Object types (cf. `Researcher`, `Professor`, `Paper` and ResearchGroup, in Fig. 1) can be declared through the use of the object type constructor object. They are positioned in an inheritance lattice which supports conventional structural top down multiple inheritance semantics. Conflicts are circumvented by demanding an unambiguous origin of the object type components and methods.

The definition of a subtype can make use of covariant redefinition of attributes and method signatures as can be seen for the component Boss and the method NewBoss of the object types Researcher and Professor in the example. The subtype relation also defines type substitutability and assignment compatibility, namely wherever an instance of a certain type can be used, it is also allowed to use an instance of one of its subtypes.

At the extensional level, we provide containers, which can be described as typed object sets. An instance of an object type can be added to any type compatible container and can also be removed from it. The containers are user defined object sets which also provide for persistence. An object persists the current session when it is in at least one object container or when it is referenced by another persistent object (persistence by reachability). Currently, there is no hierarchy defined between the containers. The object types are the factories and the containers are the warehouses of the object instances.

We assume the existence of a Turing complete procedural language for the implementation of methods and also of a declarative query language.

```
schema Research {

  object Researcher: Object {
    Boss: Researcher;
    Name: string;
    Born: date;
    Skills: set(Qualification);
    PublicationPoints: real;
    Age(): integer;
    NewBoss(B: Researcher);
  }; //Researcher

  object Professor: Researcher {
    Boss: Professor;
    For: string ;
    TeachObligation: integer;
    Offers (s: Semester): set(Course);
    NewBoss(B: Professor);
  }; //Professor

  object ResearchGroup: Object {
    Boss: Professor;
    Name: string;
    MainField: string;
    Members: set(Researcher);
    budget(year: integer): money;
    spend(reason: string; amount: money);
    receive_donation(amount: money);
  }; //ResearchGroup

  object Paper: Object{
    Title: string;
    Authors: set(Researcher);
    accept();
  }; //Paper;

  method accept() in Paper {
    foreach author in self->Authors {
      author->PublicationPoints +=
        1/card(self->Authors);
    }
  }; //accept() in Paper

  container TheResearchers: Researcher;
  container TheResGroups: ResearchGroup;
  container ThePapers: Paper;

}; //schema Research
```

Figure1: A Section of a Conceptual Schema

2.2 External Schema Derivation

An external schema is defined on top of a conceptual schema. The elements of the conceptual schema are starting points for the derivation of the types and containers of the external schema.

We distinguish between derived database types and application types. Derived database types use object preserving semantics whereas instances of application types are constructed in an object generating manner.

[1] eXoT/C (speak: exotic) stands for **ex**ternal **o**bject **t**ypes **i**n **C**arinthia

The access to the database takes place exclusively via an external schema. A direct reference or manipulation of conceptual schema elements is not possible for the user or application program. But of course, the designer of an external schema has available all the components of the conceptual schema. During the definition of the external schema, indirect access to the conceptual layer can be allowed or forbidden to any degree.

During the following discussion we will constantly use the example given in Fig. 2.

2.2.1 Derived Database Types

Conceptual object types do manifest in the external schema by derived database types (cf. the types `Scientist`, `HeadOfGroup` `ThinkTank`). Each derived database type is based on exactly one conceptual type, which is called its base type (with `ResearchGroup` being the base type of `ThinkTank` in the example). As already mentioned, we use an object preserving semantics for this derivation, so each instance of a derived database type is identical to an instance of its base type. During the type derivation, three basic restructuring operations can take place. They are combined and featured in the **derive** operator:

(1) Components and methods of the conceptual object type can be virtually removed by projection (**type restriction**). In the external schema `XRes` only the component `Name` and the method `NewBoss` are mentioned in the projection clause of `Scientist`, the other components and methods of `Researcher` were projected away.

(2) New methods which are specific for this external schema can be defined (**type extension**). This is illustrated by the method `LeadsGroups` of `HeadOfGroup` and the method `budget_per_member` of `ThinkTank`.

(3) References to conceptual types can be substituted by references to external types (**type redefinition**). Such references can occur in many places in a schema. Types are used to declare the supertypes of an object type (see the definition of type `Scientist` as supertype for `HeadOfGroup`); they are used to constitute the aggregation structure (refer to the usage of `HeadOfGroup` to type the attribute `Boss` in `ThinkTank`), and types can also be found in method signatures for typing the parameters and return values (cf. the parameters for the methods `NewBoss` of `Scientist` and `HeadOfGroup`).

When an derived database type is constructed, its position in the type hierarchy of the external schema must be explicitly defined. We permit the designer to omit any information about the supertypes, thereby declaring the external type being defined as the root of a separate type hierarchy. Using that feature, it is possible to define multiple unrelated hierarchies of derived database types (as far as inheritance, not aggregation, is concerned) in one external schema.

Furthermore, it is also allowed to derive several different derived types from the same conceptual type. By combining these multiple external definitions with the unrelated type hierarchies, the schema designer can restrict type compatibility in the external schema. But conceptually incompatible types cannot be made compatible externally. The possibility of multiple external types also provides for different perspectives to the same types in one external schema.

The concept poses some difficulties, as far as method resolution under late binding is concerned. We therefore provide means to ensure unambiguous, covariant and steady method resolution in such situations by introducing a predicate for well formed derivations and posing three schema invariants which deal with problematic constellations by forbidding them.

Methods of the conceptual base type of an derived database type are not necessarily directly callable from the user of an external schema. This is only the case, if the signature of the conceptual method has been included in the projection list of the external type (cf. method `receive_donation` in the example of Fig. 2).

```
derive schema XRes from Research {

  derive Scientist {
    from Researcher {
      Name: string;
      NewBoss(B: Scientist);
    }
  }; //Scientist

  derive HeadOfGroup: Scientist {
    from Professor {
      NewBoss(B: HeadOfGroup);
    }
    LeadsGroups(): set(ThinkTank);
  }; //HeadOfGroup

  derive ThinkTank {
    from ResearchGroup {
      Boss: HeadOfGroup;
      Name: string;
      MainField: string;
      receive_donation(amount: money);
    }
    budget_per_member(year: integer):money;
  }; //ThinkTank

}; //schema XRes
```

Figure 2: Derivation of an External Schema

```
method LeadsGroups() : set(ThinkTank)
in HeadOfGroup {
  return(select t
         from t in TheResGroups@
         where t->Boss@=self;
        );
}; // LeadsGroups() in HeadOfGroup

method budget_per_member(year:integer):money
in ThinkTank {
return(
 self->budget(year)@/card(self->members@));
};// budget_per_member() in ThinkTank
```

Figure 3: Definition of external Methods

Newly defined external methods can also have the same name as methods originating from the conceptual schema, but the signature may be defined differently. Within the body of external methods, the conceptual methods (also those not included in the projection) can be called via explicit qualification. Calling a conceptual defined method means also that a context switch to the conceptual level takes place, so the called method executes in the context it is defined in. A transfer back to the external context is made only when the execution of the transferring method terminates. In Fig. 3 this is illustrated for two externally defined methods. The body of `LeadsGroups()` mainly consists of a query against a conceptually defined container (cf. Fig. 1). The "at" character `@` is used to qualify conceptual elements (the container `TheResGroups` and the attribute `Boss` of `ResearchGroup`, which is the base type of `ThinkTank`). The calculation of the average budget per member of a `ThinkTank` is also shown in Fig. 3. This method makes use of an attribute and a method of `ResearchGroup`. The conceptually defined method `budget` executes in the conceptual context and can use all the schema elements and data defined there. The user of the external schema does not have any clue about the implementation of the computation of the average budget.

Regardless of the context a method of a derived database type executes in, it always operates directly on an instance of the conceptual base type. Updates via the external schema are made against conceptual instances, therefore no explicit propagation is needed.

2.2.2 Method Resolution

As already mentioned, method resolution in external schemas is not straightforward. One has to take into account the external as well as the conceptual inheritance hierarchy both combinated with potential method redefinitions.

Conventionally, all resolution mechanisms are invariant with respect to the static type of the object (the type of the variable an object is referred by). It is solely the dynamic type, by which the most special applicable method can be found. We called this property of dynamic resolution **method steadiness** or steady resolution (Dobrovnik 1994, Dobrovnik 1995), since it expresses that it makes no difference for the result of the resolution, if we call a method

- directly on an object o of type T via $o.m()$ or
- if we call $v.m()$, after we assign o to a variable v of type S with S being a supertype of T.

In the presence of only one inheritance hierarchy, this highly desirable property is rather straightforward to achieve; in the case of two connected and intertwined hierarchies (of the conceptual and of one external schema) it is far from trivial to guarantee method steadiness. For the purpose of this paper, it is sufficient to present our resolution mechanism by means of an example (for a theoretical discussion and a proof for the steady resolution property of the approach, we again refer to Dobrovnik 1994 and Dobrovnik 1995).

Consider the conceptual schema in Fig. 1 and the external schema in Fig. 2, where the type `Scientist` was derived from `Researcher` and `HeadOfGroup` has been derived from `Professor`. In the derivation of both types, the method `Age()` was projected away. Now we will discuss four variations of this situation to present the inherent problems of multi-schema resolution and to illustrate the concept of our approach. We will use o_P to denote objects of (dynamic conceptual) type `Professor`, while v_S stands for variables of (static external) type `Scientist` In each of the following four cases, the sequence $v_S := o_P;\ v_S.Age();$ is executed. The central point here is the question, which of the implementations of `Age()` will be executed in each of the cases, which differ in the positions where `Age()` was projected or redefined. The situations are depicted in Fig. 4, where the initial characters of the type names are used as an abbreviation.

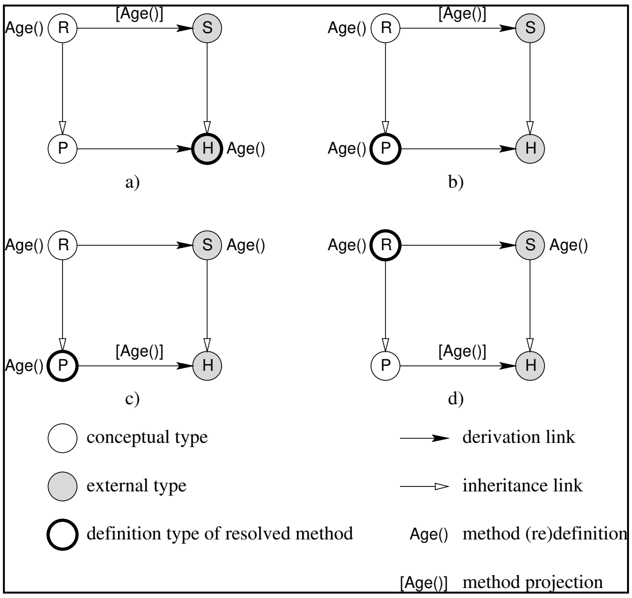

Figure 4: Method Resolution

a) `Age()` was projected in `Scientist`, and was redefined in `HeadOfGroup`. The mentioned sequence should execute the method `Age()` at `HeadOfGroup`, since the projected method in `Scientist` was specialized in `HeadOfGroup`.

b) `Age()` was projected in `Scientist`, and was redefined in `Professor`. The same sequence as above in case a) should lead to the execution of method `Age()` at `Professor`, since this is the most special method for objects of type `Professor` in this constellation.

c) `Age()` was defined as new external method `Scientist`, was projected in `HeadOfGroup`, and was redefined in `Professor`. Here again, method `Age()` at `Professor` should be executed, since the projection took place below `Scientist`, and should therefore be considered as a specialization of `Age()` at `Scientist`.

d) `Age()` was defined as new external method in `Scientist`, and was projected in `HeadOfGroup`. Also in this case, the explicit definition of a new method at `Scientist` is being overridden by the projection at `HeadOfGroup`, thereby declaring that the conceptual method at `Researcher` should be regarded as appropriate for `HeadOfGroup`.The mentioned program sequence should therefore execute the method `Age()` at `Researcher`.

Of course, the execution of $o_H.Age()$ should lead to the execution of the same method body in each of the four cases. Our resolution mechanism is constructed to take into account the complex interaction between projection, redefinition, inheritance and derivation in exactly the way illustrated by the example. Thereby the most special method is found irrespective of the static type of the variable referencing the receiver.

2.2.3 Application Types

The second ‘‘kind’’ of external object types are application types. In contrast to derived database types, the application types are not based on exactly one, but on an arbitrary number of conceptual object types. In particular, the construction of application types without a base type is allowed. But application types can also base on more than one conceptual type, which makes them well suited to deal with explicit joins.

For the instantiation of the objects of application types, we use an object generating semantics, which builds new external objects based on the states and identities of the conceptual objects participating in the join. These objects do exist only in the external context, so it is not possible to reference them from derived database types. Taking into account our definition of persistence via reachability, this also means that application objects cannot persist the session.

```
derive aptype Group_Paper {
 from
   ResearchGroup R {
     Boss: HeadOfGroup;
     Name: string;
     MainField: string;
   }
   Paper P {
     Title : string;
     Authors : set (Scientist);
     accept();
   }
 group_share_of_paper: real;
 };//Group_Paper
```

Figure 5: Definition of an Application Type

In Fig. 5 an application type `Group_Paper` is defined. It has two base types, `ResearchGroup` and `Paper`. The variables `R` and `P` are called base variables and make it possible to use a conceptual type more than one time as a base type in an application type (for instance when joining researchers with their bosses).

The type constructor **aptype** for application types again allows three restructuring mechanisms to be applied. In principle, these are the same as for derived database types, namely type restriction, type extension and type redefinition, but differ in subtle ways.

The projection of attributes and methods is not carried out just virtually in terms of visibility, but really applied to the types, so those components are not longer accessible by any means. In addition to the extension of types by newly defined methods, application types can be extended by stored attributes, too (cf. the component `group_share_of_paper` in Fig. 5). Via type redefinition, conceptual types can be substituted by application types or derived database types. So it is possible to use a derived database type or an application type as type of a component (note the usage of `HeadOfGroup` as component type for `Boss` in the example) or as type in a method signature of an application type. But application types form a type hierarchy of their own in an external schema. There is no subtype relationship between any derived database type and an application type, and as already mentioned, derived database types cannot refer to application types.

On the contrary to derived database types, updates posed on instances of application types do not automatically manifest as updates of conceptual objects, since there is no direct one-to-one correspondence between instances of application types and instances of conceptual types. Updates are feasible by means of specially written methods which have access to the conceptual objects that were used in the instantiation process of the application object. So it is possible to explicitly propagate updates to the underlying conceptual objects.

2.2.4 External Containers

Containers in the external schema are constructed on top of one ore more conceptual containers via a query expression, which is illustrated in Fig. 6.

```
container CSGroupLeaders : HeadOfGroup =
          select G.Boss
          from G in TheResGroups@
          where G.MainField='CS';

container TheGroupPapers : Group_Paper =
          select (r,p)
          from r in TheResGroups@,
               p in ThePapers@
          where r.Members intersect
                p.Authors;
```

Figure 6: Definition of External Containers

The set of instances of the external containers cannot be operated on directly, it is solely the query expression which is used to generate this set. This instantiation takes place every time the container is referenced, containers could therefore be viewed as snapshots of the database. Updates of the extension sets of containers in the conceptual schema are possible if the external schema designer provides explicit methods for doing so.

3 APPLICATIONS AND CONTRIBUTIONS OF VIEWS

3.1 Better Coupling Characteristics

A major reason to introduce views in database systems was to achieve logical data independence. From a more general software engineering point of view this means to reduce the degree of coupling between the database and the application programs. Since object oriented databases provide methods additional to data containers, we believe that this more general notion is adequate for discussing views in OODBMS.

The introduction of views makes the coupling between the database and the application schema explicit. All elements of the conceptual schema needed by the application program are explicitly contained in the external model. This has great influence on the complexity of changes in application programs or schemas. We will discuss this important issue in the following subsection.

Application programmers have only to know the external model. They perceive the external schema as a closed world where all elements needed are defined or declared. All interaction between an application program and the database has to go through the external schema. In our approach the derived types and containers of the external schema can be presented to the application programmer or user just as a simple data model. There are no links and references to the conceptual schema contained in this presentation nor are they needed to understand the external schema or to write programs using the database. This enormously reduces the complexity of designing and implementing application programs.

With object oriented databases it is in principle possible to write one complex system containing the database and all application programs using it. In contrast to relational databases there is no system defined boundary between the database and the application program. It is the designer who decides where types or methods are defined according to design methods, quality criteria, and - last but not least - the modularization concepts provided by the system. Our concept for external schemas offers the designer great flexibility in this decision. Our view concept allows the design of 'lean' conceptual schemas, i.e. the conceptual schema contains the definition of all stored data and the methods which are intrinsically needed to manipulate these data objects and those which are shared by several applications or users. There is no need to place methods and even whole types of (transient) objects in the conceptual schema, if they are only needed by single applications.

This support of modularization will allow object-oriented databases to become building blocks of very large application systems. Without such modularization object-oriented application system are likely to become to complex for being manageable.

3.2 Change Transparency

The management of change is one of the big issues in software engineering. It is folklore knowledge by now that the largest share of the overall life cycle cost of software products is in the maintenance phase. For databases this observation holds even more due to their comparatively long lifetime.

The lower degree of coupling between the database and the application programs subsumes the effects of changes, as the traditional logical data independence does for conventional databases. The influence of changes should be as local as possible. Furthermore, the interdependencies of application programs and the conceptual schema are explicitly given in the external schemas. So we can easily determine which external schemas (and which application programs) are affected by changes of the conceptual schema. External schemas serve also as interface modules. Quite often changes in the conceptual schema can be dealt with by changing solely external schemas without the need to even check the corresponding application program.

For simple changes in an application program, it is only required to change the definition of the external schema in order to provide a suitable interface for the application. A schema change at the conceptual level can be avoided, thereby the change is confined in the rather narrow boundaries of one external schema and does not interfere neither with other applications nor with other external schemas.

Support for new applications can be provided easily. If a fresh application cannot operate on top of an existing external schema, then an additional external schema specially designed for the new application can be constructed. Again, other external schemas, other applications and especially the conceptual schema are not affected by this extensions.

Even, if requirements for drastic changes of applications which cannot be accomplished by changing an external schema definition arise, or if the conceptual schema itself really needs to be changed, the view system has great advantages. Since the connections between an external schema and the conceptual schema are made explicit in the view definition phase, it is straightforward to find out which of the external schemas are afflicted by the change, and which schemas are not. So the scope of the effect of the applied change can be narrowed to a subset of the external schemas and equally to a subset of the applications. The effects of the changes of the conceptual schema can then be tackled by adapting the definitions of the afflicted external schemas. The original interface to the application programs (the generated schema) can also often be derived from the new conceptual schema in their original form. Therefore applications will be immune to changes of the conceptual schema to a large extent.

3.3 Tighter Security

Views can also be used as an effective means of access control and to implement a fine grained security mechanism. Since the usage of components and notably also of methods can be restricted via projection, the definition of external schemas enables the DBA to draw a tight security boundary between the parts of the database the user is entitled to work on and the part he has no clearance for. In particular, the usage of views does not only allow to restrict the rights of an user to access data elements. The power and flexibility of external schemas with respect to security is the ability to put strong confinements on the user concerning his possibilities to apply certain operations on those data objects.

The external schema can be constructed on top of a small and narrow part of the conceptual schema, thereby effectively limiting the visibility of the omitted conceptual schema and data elements.

Only the parts of the database, which are explicitly mapped to the outer interface of the external schema are accessible to the user of the external schema, and the way of manipulation of those visible parts can be restricted to any degree. By providing methods for certain selections or updates of the data, the designer of the external schema can make available operations for the user of the external schema in a strictly controlled way. These methods can be made arbitrarily restrictive on their execution. Checks for permissions of users on schema elements, validation of receivers and parameters of the methods and arbitrary complex access and modification rights can be implemented in such methods.

Nevertheless, these restrictions do not necessarily diminish the power or the usability of the external schemas to an unwanted degree. Since the implementor of the external schema has access to all parts of the whole underlying conceptual layer, he can design and provide quite powerful methods which are not restricted in any sense. Neither the scope of those methods is inherently confined to just a part of the conceptual schema, nor are there any system implied limitations on the operations such methods can execute.

So the designer of the external schema has available the full power of the complete conceptual schema. It lies in his responsibility to construct an adequate interface for users in terms of power and security. Such an external schema should provide only the necessary and sufficient operations on the data objects the users are entitled to see, but it should also be restrictive in terms of validation, plausibility and consistency.

Obviously, the security aspects mentioned above do not only apply to human users of external schemas but also to applications which are built on top of an external schema.

3.4 Architectural Considerations

The client-server architecture of current OODBMS might be used quite profitable for the implementation of external schemas.Through the introduction of external schemas, a boundary in the database between the conceptual and the external level is drawn. This schema boundary could be mapped directly onto the client-server paradigm.

Such an implementation would put the responsibility of the mapping between the external schema and the conceptual schema onto the client components of the DBMS. The client itself would have to deal with those aspects of the external schemas which have no correspondence in the conceptual schema. In particular, late binding and method resolution have to be performed in the client as well as execution of externally defined methods and also the creation and administration of instances of application types.

The server could remain relatively unchanged. Virtually no additional functions would have to be performed by the server at run time. Of course, the server would be responsible for the storage and administration of the external schemas themselves in the form of an adequate meta-schema. But this is merely a rather trivial extension of a function already present in the server components.

This considerations on an implementation of external schemas via extension of the client side of an OODBMS cannot be more than a sketch without any further assumptions about the real architecture of existing systems. Besides ease of implementation and smooth integration, a proposal for such an implementation would have to take into account the performance degradation of the overall system and the mere resources needed at runtime to support the extended functionality.

The single most influential property to be considered would be the general architecture of the division of operations between the client and the server already in the underlying OODBMS itself. Since this design would be virtually impossible to change without massive, difficult and costly rework, it will be used as the underlying framework without major revisions.

3.5 Application Packaging

An external schema can also be quite useful as a starting point for application packaging. Since applications are executed against an external schema, it is exactly the definition context of the external schema that is required to run the application. This is the set of all the types and methods that are used directly or indirectly in the external schema definition.

This transitive closure of the schema elements used by the external schema under consideration can be constructed out of the derivation relationships between the conceptual and the external schema. An external schema can thereby define a subschema of the conceptual schema which is necessary and sufficient to run all applications which are based on this external schema.

This ability to partition the conceptual schema into a set of relevant and a set of irrelevant elements from an application specific point of view can be used to construct a conceptual subschema which forms the basis of the desired application. Together with the definition of the external schema and the application itself it comprises a closed application package unit.

4 CONCLUSION

In this paper, we presented some general considerations for view support in OODBMS. We argued, that for an adequate degree of power, the system has to provide functionality for all major elements of the object oriented paradigm. In particular, mechanisms for restructuring at the type and schema level, the provision of object preserving as well as object generating semantics and the inclusion of behavior and dynamics are crucial for the flexibility and applicability of the views.

With eXoT/C, we presented an approach to introduce external schemas in OODBMS that takes into account all this aspects and offers logical data independence and a greater degree of modularity in information systems built on top of OODBMS. Better coupling characteristics can be achieved and a a high degree of change transparency can be gained. Extensions of the system and maintenance activities can therefore be carried out with much less effort than in a monolithic system. A high degree of security can be implemented without diminishing the usability or power of the applications to an unwanted degree when views are used to specify and enforce access control. We also discussed to some extend, how views could be implemented in an existing system and how they could be used as starting points for application packaging.

References

Abiteboul, S. and Bonner, A. (1991). Objects and Views. In Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data, pages 238-247, Denver, Colorado.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The Object-Oriented Database System Manifesto. In Proc. First International Conference on Deductive and Object-Oriented Databases, pages 40-57, Kyoto, Japan. Elsevier Science.

Bancilhon, F. and Kim, W. (1990). Object Oriented Database Systems: In Transition. SIGMOD RECORD, 19(4):49-53.

Barclay, P. J. and Kennedy, J. B. (1993). Viewing Objects. In Worboys, M. and Grundy, A., editors, Advances in Databases, Proc. 11th British National Conf. on Databases (BNCOD), number 696 in LNCS, pages 93-110, Keele, UK. Springer.

Beeri, C. (1990). A Formal Approach to Object-Oriented Databases. Data & Knowledge Engineering, 5(4):353-382.

Beeri, C. (1992). New Data Models and Languages - The Challenge. In Proc. 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 1-15, San Diego.

Bertino, E. (1992). A View Mechanism for Object-Oriented Databases. In Pirotte, A., Delobel, C., and Gottlob, G., editors, Advances in Database Technology - EDBT'92, number 580 in LNCS, pages 136-151, Vienna, Austria. Springer.

Busse, R., Frankhauser, P., and Neuhold, E. J. (1995). Federated Schema in ODMG. In Eder, J. and Kalinichenko, L., editors, Proc. 2nd Intl. East/West Database Workshop, pages 356-379, Klagenfurt, Austria. Springer.

Cattell, R., Atwood, T., Duhl, J., Ferran, G., Loomis, M., and Wade, D. (1993). The Object Database Standard ODMG-93. Morgan-Kaufmann.

Cattell, R., Atwood, T., Duhl, J., Ferran, G., Loomis, M., and Wade, D. (1994). The Object Database Standard ODMG-93 Release 1.1. Morgan-Kaufmann.

Deux, O. et al. (1991). The O_2 system. Communications of the ACM, 34(10):34-48.

Dobrovnik, M. (1995). Externe Schemata in objekt-orientierten Datenbankmanagementsystemen; Logische Datenunabhängigkeit durch Änderungen über Sichten. Dissertation, Institut für Informatik, Universität Klagenfurt.

Dobrovnik, M. and Eder, J. (1993a). A Concept of Type Derivation for Object Oriented Database Systems. In Gün, L., Onurval, R., and Gelenbe, E., editors, 8th Intl. Symposium on Computer and Information Sciences (ISCIS VIII), pages 113-120.

Dobrovnik, M. and Eder, J. (1993b). View Concepts for Object-Oriented Databases. In Lasker, G. E., editor, Proc. Intl. Symposium on System Sciences, Informatics and Cybernetics, Baden-Baden, pages 163-167.

Dobrovnik, M. and Eder, J. (1994). Adding View Support to ODMG-93. In Mizin, I. A., Kalinichenko, L. A., and Zhuralev, Y. I., editors, Proc. Intl. Workshop on Advances in Databases and Information Systems, pages 74-81, Moscow, Russia. Moscow ACM SIGMOD Chapter.

dos Santos, C. S. (1995). Design and Implementation of Object-Oriented Views. Extended Version of: dos Santos, C.S. (1995) Design and Implementation of Object-Oriented Views. In: Revell, N., Tjoa, A M., editors, Proc. 6th Intl. Conf on Database and Expert Systems Applications DEXA'95, number 978 in LNCS, pages 91-102, London, UK, Springer.

dos Santos, C. S., Abiteboul, S., and Delobel, C. (1994). Virtual Schemas and Bases. In Jarke, M., Bubenko, J., and Jeffery, K., editors, Advances in Database Technology - EDBT'94, pages 81-94, Cambridge, UK. Springer.

Geppert, A., Scherrer, S., and Dittrich, K. (1992). A View Mechanism for NO2. Technical Report TR 93.X.5#1, Institut für Informatik, Universität Zürich.

Geppert, A., Scherrer, S., and Dittrich, K. (1993). Derived Types and Subschemas: Towards better Support for Logical Data Independence in Objectoriented Data Models. Technical Report TR 93.27, Institut für Informatik, Universität Zürich.

Heiler, S. and Zdonik, S. (1988). Views, Data Abstraction, and Inheritance in the Fugue Data Model. In Dittrich, K., editor, Advances in Object-Oriented Database Systems, number 334 in LNCS, pages 225-241. Springer.

Heiler, S. and Zdonik, S. (1990). Object Views: Extending the Vision. In Proc. 6th Intl. Conf. on Data Engineering, pages 86-93.

Heuer, A. and Sander, P. (1990). Preserving and Generating Objects in the Living in a Lattice Rule Language. In Göers, J. and Heuer, A., editors, Proc. Second Workshop on Foundations of Models and Languages for Data and Objects, Informatik-Bericht 90?3, pages 1-36, Aigen (Austria). Institut für Informatik, TU Clausthal.

Heuer, A. and Sander, P. (1991). Perserving and Generating Objects in the Living in a Lattice Rule Language. In Proc. Intl. Conf. on Data Engineering, pages 562-569, Kobe, Japan.

Heuer, A. and Sander, P. (1993). The Living in a Lattice Rule Language. Data & Knowledge Engineering, 9(3):249-286.

Ishikawa, H., Izumida, Y., Kawato, N., and Hayashi, T. (1992). An Object-Oriented Database System and its View Mechanism for Schema Integration. In Chen, Q., Kambayashi, Y., and Sacks-Davis, R., editors, Future Databases'92, Proc. 2nd Far-East Workshop on Future Database Systems, volume 3 of Advanced Database Research and Development Series, pages 194-200, Kyoto, Japan.

Jungclaus, R., Saake, G., and Hartmann, T. (1991). Language Features for Object-Oriented Conceptual Modeling. In Teory, T., editor, Proc. 10th Intl. Conf. on the ER-approach, pages 309-324, San Mateo.

Kifer, M., Kim, W., and Sagiv, Y. (1992). Querying Object-Oriented Databases. In Stonebraker, M., editor, Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 393-402, San Diego.

Kim, W. (1994). Observations on the ODMG-93 Proposal for an Object-Oriented Database Language. SIGMOD RECORD, 23(1):4-9.

Kim, W. and Kelley, W. (1995). On View Support in Object-Oriented Database Systems. In Kim, W., editor, Modern Database Systems, The Object Model, Interoperability, and Beyond, chapter 6, pages 108-129. ACM Press.

Kotz-Dittrich, A. and Dittrich, K. R. (1995). Where Object-Oriented DBMSs should do better: A Critique based on early Experiences. In Kim, W., editor, Modern Database Systems, The Object Model, Interoperability, and Beyond, chapter 12, pages 238-254. ACM Press.

Kuno, H. and Rundensteiner, E. A. (1993). Developing an Object-Oriented View Management System. IBM CASCON. TR r-93-2.

Meier, A. and Wüst, T. (1995). Objektorientierte Datenbanksysteme - ein Produktvergleich. Theorie und Praxis der Wirschaftsinformatik, 32(183):24-40.

Motschnig-Pitrik, R. (1995). Requirements and Comparision of View Mechanisms for Object-Oriented Databases. Submitted to Information Systems.

Neuhold, E. J. and Schrefl, M. (1988). Dynamic Derivation of Personalized Views. In Proceedings 14th VLDB Conference, pages 183-194, Los Angeles.

Rundensteiner, E. (1992). Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In Proc. 18th VLDB Conference, pages 187-198, Vancouver, Canada.

Schiefer, B. (1993a). Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen. Dissertation, Universität Karlsruhe.

Schiefer, B. (1993b). Supporting Integration & Evolution with Objectoriented Views. Technical Report FZI.063.3, Forschungszentrum Informatik (FZI), Haid- und-Neu-Strasse 10-14, D-76131 Karlsruhe, Germany.

Scholl, M. H., Laasch, C., and Tresch, M. (1990). Views in Objectoriented Databases. In Göers, J. and Heuer, A., editors, Proc. Second Workshop on Foundations of Models and Languages for Data and Objects, Informatik-Bericht 90?3, pages 37-58, Aigen (Austria). Institut für Informatik, TU Clausthal.

Scholl, M. H., Laasch, C., and Tresch, M. (1991). Updateable Views in Object-Oriented Databases. In Delobel, C., Kifer, M., and Masunaga, Y., editors, Proc. 2nd DOOD Conf., number 566 in LNCS, pages 189-207, Munich, Germany. Springer.

Shilling, J. J. and Sweeney, P. (1989). Three Steps to Views: Extending the object-oriented paradigm. In Proc. OOPSLA '89, pages 353-361, New Orleans.

[Tanaka et al., 1988] Tanaka, K., Yoshikawa, M., and Ishihara, K. (1988). Schema Virtualization in Object-Oriented Databases. In Proc. 4th Intl. Conf. on Data Engineering, pages 23-30. IEEE.

Zand, M., Collins, V., and Caviness, D. (1995). A Survey of Current Object Oriented Databases. The DATA BASE for Advances in Information Systems, 26(1):14-29.