Adding View Support to ODMG-93 [1]

M. Dobrovnik, J. Eder

Institut für Informatik
Universität Klagenfurt
Universitätsstr. 65
A-9020 Klagenfurt, Austria
e-mail: {michi,eder}@ifi.uni-klu.ac.at

Abstract

A concept to introduce external models in object oriented databases is presented, such that application programs do no longer interface directly the whole conceptual schema, but work against external schemas specifically designed for the applications requirements. There are virtually no restrictions for such applications, since the interaction with the database takes place via updateable views.

The data model is a somewhat simplified form of ODMG-93 [4], where we incorporated the additional constructs we need for the external schema definition. The approach makes a clear distinction between types and classes, and also separates the type and class hierarchies of the conceptual schema from the external type and class hierarchies. With type derivation, we provide a powerful type restructuring mechanism, which allows to define an external type which is based on a conceptual type. In the derivation process, one can omit conceptual components and methods or redefine their types. Additional methods can be defined for external types as well.

By defining well formed external schemas via constraints and schema invariants, we are able to guarantee unambiguous method resolution, steadiness of method resolution and compliance with the covariant subtyping principle. The full semantics of the conceptual schema are preserved. The designer of the external schema can make use of all information contained in the conceptual schema, in particular conceptual methods can be called from externally defined ones.

In this paper, we concentrate on the area of type derivation and method resolution.

1 Introduction

External models [13] in database systems are used for a number of different purposes. One key aspect of such a model is the representation of a specific view of a user or an application on the conceptual schema of the database whereas the external model has to provide the mechanisms to map terms and concepts of the conceptual model to those of a user or application system. Another important feature of external models is their interface character. They are the interface specification between the conceptual schema and the external clients. Such external models can also serve as a security mechanism by restricting certain operations on the conceptual schema. Yet another aspect is the possibility to predefine queries which can be used later on.

This external layer results in logical data independence and reduces the amount of necessary maintenance in case of changes of the conceptual schema or changes in the applications. Except for the explicitly desired restrictions of the external schema, the application should not be restricted in any further

[1] To appear in: "Advances in Databases and Information Systems: ADBIS '94" , Proc. of the Intl. Workshop of the Moscow ACM SIGMOD Chapter

way by the system. So the external schema should be as tight as needed, but also as transparent as permissible and possible for the application. Updateability of the views is a crucial point in this context.

In the recent past, there have been quite a lot of proposals for view systems in OODBMS. These approaches differ with respect to paradigms and aims.

Some of the approaches [2, 1, 3, 11, 12] see views mainly as named query expressions and are primarily concerned with the integration of the type of query results into the type lattice of the conceptual schema.

Others treat the derived types as separate entities [9], or attach them directly to a root class [10].

In [8], an approach similar to that presented here is proposed, but it is less powerful and discussed at a rather informal level.

We do not intend to determine the behavior (the set of applicable methods) of a type automatically, as it is proposed in [2], we rather want the designer of the external schema to explicitly specify the desired behavioral aspects of the derived types.

1.1 General Ideas

Our approach is based on the following points:

- Intensional and extensional information are treated differently (types and classes).
- Provide updateable views by object preserving queries.
- No explicit mapping between conceptual and external object.
- No introduction of new external object types that are not based on corresponding conceptual types, as the specific application type should not be in the scope of the view in order to keep good cohesion and low coupling.
- Decouple the external and conceptual level by introduction of separate type and class hierarchies for each external schema.
- Let the view designer specify the desired behavior of the objects.
- Preserve type incompatibilities of the conceptual schema (objects that are not compatible at the conceptual level, are also incompatible at the external level).
- Provide different external (possible incompatible) perspectives of one conceptual type in the external schema.
- Provide well formed and closed conceptual and external schemas.
- Incorporation of new behavior into external objects.
- Possibility to make use of existing conceptual methods.
- Generate new conceptual object instances from the external model.

1.2 First Sketch of the Data Model

As already outlined in [6, 5], the data model will be closely tied to a schema definition language, a procedural language to write the method bodies and a declarative query language. In this paper we will not elaborate on these languages but concentrate on key aspects of the type system.

A schema consists of type and class definitions. Types represent the intensional information and describe the structure of objects and values together with the behavior of objects. Types are defined in a covariant inheritance lattice for structural (top-down) specification inheritance. In the future, we want to consider multiple inheritance, but for now we settle with single inheritance.

Classes are object containers structured in a inheritance lattice for (bottom-up) instance inheritance. Whenever an object is member of a class, it is also member of all its superclasses. Classes can contain objects which are compatible with a ground type, but there can be any number of classes for a given object type, including none. An object may be member in several (unrelated) classes which are compatible with the objects type.

Methods can be implemented in a Turing complete programming language which may also contain expressions of the query language. The declarative query language offers generic operations for projection, selection, extension and set operations on classes. It is beyond the scope of this paper but presented in somewhat more detail in [7].

An external schema provides the definition context and name scope for the derived types and classes. In an external schema we can construct derived types by a type derivation operator which allows to define an external type based on a conceptual type. In the derivation, we can apply type restriction, where we can virtually remove properties (components and methods) from the type definition, and type extension where we can add new methods to a type.

A derived type that is as well a projection as an extension of a conceptual type, can't be inserted into the conceptual type hierarchy in a straightforward manner, without either loosing the covariant subtyping property or coming up with a form of cumbersome upward schema inheritance.

So we provide for a different type lattice for each external schema. Also, a separate external class lattice is constructed out of derived classes (views). Views are based on the conceptual classes and can be built using the generic query operations mentioned above. To provide updateable views, the queries must be object preserving. As already mentioned, classes are not discussed in detail in this paper, where we will concentrate on the types and type derivation and on properties of conceptual and external schemas.

2 Conceptual Schema

Our definition of types and schemas does not take into account the full richness of ODMG's ODL, because for the time being, we strived for a more formal discussion, which is founded on the essential aspects of the ODMG data model. So, let us define a schema:

Definition 1 (Conceptual Schema) *A conceptual schema $S = schema(O, A, C)$ consists of a set of named object types O, a set of anonymous non-object types A, and a set of class definitions C.*

Types are used to describe the structure of objects and values. They define the components of objects and the methods which can be applied to them. We provide several predefined types (e.g. *int*, *bool*, *string*, *obj*, ...) which we call *atomic types*. Other types can be constructed by application of type construction operators (*complex types*).

Definition 2 (Types) *Can be defined as follows*

- *Atomic types are types.*

- *If T is a type, then set(T) is a set type, the domain of which is the sets of values of type T.*

- *If $T_1, \ldots, T_n$ are types, then tuple(c_1, $\ldots$,c_n) is a tuple type, the domain of which is the tuples of n components, where each component $c_i = l_i/T_i$ has a name name(c_i) $= l_i$ and has a domain of T_i.*

- *If N is a Name, S is the name of an object type or type obj, C is a set of components (name/type pairs), M is a set of method signatures $m_i(p_{i1}, \ldots, p_{in_i})/T_i$ and wfo(S, C, M) holds, then T_O = object(N, S, C, M) is an object type. S may be obj, which means that T_O inherits directly from the root object type. N = name(T_O) is the name of the newly created object type T_O. S = super(T_O) is the supertype from which T_O inherits. Please note, that in the sequel we will not explicitly distinguish between the name and the definition of a type, except where necessary.*

The function wfo() (well formed object) in the object type definition asserts that only valid object types can be in the schema. It will be defined later on.

Definition 3 (Signature) *A signature $s(l_1/T_1, \ldots, l_n/T_n)/T$ is the description of the interface of a method named s, which is also the name of the signature. The method takes n input parameters and has a result type of T.*

Two signatures s, t are covariantly signature compatible if they have the same name, have the same arity (number of parameters) all their parameters have the same name, the type of each parameter of s is a subtype of the type of the corresponding parameter of t, and the result type of s also is a subtype of the result type of t; formally:

Definition 4 (Covariant Signature)
$$
\begin{aligned}
&covar(s,t) \Leftrightarrow \\
&\quad s = f_s(l_{s1}/T_{s1}, \ldots, l_{sn}/T_{sn})/T_s \wedge \\
&\quad t = f_t(l_{t1}/T_{t1}, \ldots, l_{tm}/T_{tm})/T_t \wedge \\
&\quad f_s = f_t \wedge n = m \wedge \\
&\quad \forall (l_{si}/T_{si}, l_{ti}/T_{ti})\, l_{si} = l_{ti} \wedge T_{si} \preceq T_{ti} \wedge \\
&\qquad T_s \preceq T_t
\end{aligned}
$$

The following theorem states that the covar predicate is transitive which follows directly from the definition.

Theorem 1 (Transitivity of Covariance)
$$
\begin{aligned}
&covar(S,T) \wedge covar(T,U) \Rightarrow \\
&\quad covar(S,U).
\end{aligned}
$$

We define the subtype relationship in the usual way:

Definition 5 (Subtype Relationship) *Let S, T be Types then S is a subtype of T ($S \preceq T$) and T is a supertype of S ($T \succeq S$) if:*

- $S = T$, *or*

- $S = set(S_E), T = set(T_E), S_E \preceq T_E$, *or*

- $S = tuple(l_{S_1}/T_{S_1}, \ldots, l_{S_n}/T_{S_n})$,
  $T = tuple(l_{T_1}/T_{T_1}, \ldots, l_{T_n}/T_{T_m})$,
  $n \geq m, \forall (i = 1, \ldots, m) T_{S_i} \preceq T_{T_i} \wedge$
  $l_{S_i} = l_{T_i}$, *or*

- $\exists S' | S = object(N, S', C, M) \wedge S' \preceq T$

Since all object types inherit either directly or transitively from type obj, there is a single object type hierarchy in the conceptual schema.

Let $S = \text{object}(N, T, C, M)$ where $\text{wfo}(T, C, M)$ holds, then $\text{meth}(S)$ and $\text{meth}(T)$ are the set of method signatures which are defined for S and T respectively. The defined method set of an object type consists of the directly defined method set and of the methods which are inherited from the supertype. The set of inherited methods of S is noted as $\text{imeth}(S) = \{m_i \in \text{meth}(T) | \text{name}(m_i) \notin \text{names}(M)\}$. The set of defined methods for an object type is defined as $\text{meth}(S) = M \cup \text{imeth}(S)$. M is the set of explicit methods of S, also denoted by $\text{emeth}(S)$.

Similar definitions apply for the components. $\text{comp}(S)$ and $\text{comp}(C)$ are the sets of defined components for S and T respectively. The set of inherited components of S is noted as $\text{icomp}(S) = \{c_i \in \text{comp}(T) | \text{name}(c_i) \notin \text{names}(C)\}$. The set of defined components for an object type is defined as $\text{comp}(S) = C \cup \text{icomp}(S)$. C is the set of explicit components of S, also denoted by $\text{ecomp}(S)$.

For $S = \text{object}(N, T, C, M)$ to be a valid object type definition, $\text{wfo}(T, C, M)$ must hold. We require (1) that each component in C that has the same name as one of the components of the supertype T has a type which is a subtype of the type of the inherited component (redefined components types must be subtypes of the original components types). The second requirement is (2) that that the method redefinitions in M are covariantly signature compatible with the corresponding methods of T.

Definition 6 (Well Formed Object) $\mathit{wfo}(T, C, M) \Leftrightarrow$

1. $\forall (c_T \in \mathit{comp}(T)) \forall (c_S \in C)$
$\mathit{name}(c_S) = \mathit{name}(c_T) : T_S \preceq T_T \land$

2. $\forall (m_T \in \mathit{meth}(T)) \forall (m_S \in M)$
$\mathit{name}(m_S) = \mathit{name}(m_T) :$
$\mathit{covar}(m_S, m_T)$

If all object definitions in a schema are well formed then the schema obeys the covariant subtyping principle.

Let us note, that the method object generating method `new()` is treated somewhat special. The execution of `T.new()` on a type `T` results in the creation of a new object with type `T`. The actual parameters supplied in the call are used to initialize the new object properly. Since it is permitted to define new components on an object type, it should be possible to initialize those new components. This implies that the length of the parameter list of `new()` is variable. The definition of covariance required the length of a methods parameter list to remain constant in the inheritance hierarchy. To resolve this conflict, we allow `new()` to have a variable parameter list, thereby exempting it from the covariance rule. This poses no problem, since there is another aspect in which `new()` differs from the other methods defined for an object type; `new()` is a method executed on a type, rather than on an object. It is not allowed to send `new()` to an object. When one writes down a call of an ordinary method, the most specific type of the object it will be executed on is not known. At each call of `new()`, one exactly knows the type on which it is applied, so the parameters can be statically type checked.

For type equivalence and type compatibility, we use the same rules as in ODMG. Two types S, T are equivalent, iff S is a subtype of T and vice versa. More explicitly, two types are equivalent, if they

- are named types (predefined types or object types) and have the same name, or

- they are anonymous types (set or tuple types) and if their structure is identical. For set types, the element type must be identical, and for record types, components must be identical in numer, name and type.

So basically, we use name equivalence for object types and structural equivalence for anonymous types. We do not use structural object equivalence, since we do not want to introduce additional type equivalences in the external schema which do not have any correspondence in the conceptual schema.

Such an unwanted equivalence would arise externally, if two external types have the same externally visible structure, but could be based on incompatible conceptual types.

As an example, let `Song` and `Book` be two conceptual types with different structure and let us define two external types `ESong` and `EBook` via a projection on just the components `Title` and `Authors`. The application systems perception of the external types is just their external structure, there are no visible connections to the conceptual schema. For the application it is not possible to make a decision, whether the two types are equivalent, based on the externally available structural type information. Using name equivalence, we address and avoid this problem.

An assignment `a:=b` is defined, if the type of `a` is a supertype of the type of the expression `b`. The type of an actual parameter must be a subtype of the formal parameter it substitutes.

The following example is a small and simple part of the conceptual schema of a database. The `conceptual schema` consists of three type definitions (`interfaces`), where `Advisor` and `Student` inherit from `Person`.

An Example

```
conceptual schema university {

 interface Person {
    attribute string name;
    attribute date birthdate;
    int age();
    boolean older(p:Person);
  }

  interface Advisor:Person {
    attribute set(Student)
              candidates;
    void add_candidate(s:Student);
    Advisor new(aname:string,
                cands:set(Student));
  }

  interface Student:Person {
    attribute Advisor the_advisor;
    attribute date advised_since;
  }
}

int Person::age() {
  return years(birthdate-today());
}

boolean Person::older(p:Person) {
  return self.age() >= p.age();
}

Advisor::new(aname:string,
             cands:set(Student)) {
  name=aname;  candidates=cands;
}

void Advisor::
     add_candidate(s:Student)
{
  candidates+=s;
  s.advised_since=today;
  s.the_advisor=self;
}
```

The conceptual schema presented here is not complete, since no class definitions are given. As we already mentioned, this paper concentrates on types, we do not elaborate on classes here. Some preliminary ideas can be found in [6, 5, 7].

3 External Schema

An external schema defined on top of a conceptual schema consists of external type and view definitions.

Definition 7 (External Schema) *An external schema $E = eschema(S, O, A, V)$ is based on one conceptual schema S and consists of a set of derived object type definitions O, a set of anonymous non-object type definitions A, and a set of derived class definitions (views) V.*

As we did for the conceptual schema, we will concentrate on the area of types, in the external schema, in particular on type derivation. In an external schema we construct derived types by the type derivation operator derive().

An external derived type is based on exactly one conceptual object type. The external appearance of the object (the external type) can be changed from the conceptual definition.

- Properties (components and methods) of the conceptual type can be virtually removed from the external type definition. Since the external type is confined in narrower bounds, we call this aspect *type restriction*.
- Via *type extension* is also possible to add new methods (but no new components) to an external type. So arbitrarily complex additions to the behavior of an object can be constructed.
- It is allowed to change the definition of the types of components and method signatures (*type redefinition*).

One of the basic ideas of our concept is the separation of the type and class hierarchy of the conceptual schema from the type and class hierarchies of the external schema. Therefore, type derivation must also take into account the placement of the external types in the external type hierarchies.

Now let us define the derive() operator:

Definition 8 (Derived Type) *With*

- *N is a Name,*
- *$\overline{T_S}$ is an optional external type; called the external supertype of $\overline{T_E}$*
- *T_B is a conceptual object type on which $\overline{T_E}$ is based; it is called the base type of $\overline{T_E}$,*
- *C_P is a set of component definitions; called the projected components of $\overline{T_E}$,*
- *M_P is a set of method signatures; it is called the method projection set and states how the signature of the methods of the base type should be interpreted in the external schema.*
- *M is the set of direct methods of $\overline{T_E}$*

then $\overline{T_E} = derive(\overline{N}, [\overline{T_S}], T_B, C_P, M_P, M)$ is a derived type, if $wfexo(\overline{T_S}, T_B, C_P, M_P, M)$ holds. For convenience, we further define
$pmeth(\overline{T_E}) = M_P$, $emeth(\overline{T_E}) = M$,
$pcomp(\overline{T_E}) = C_P$, $name(\overline{T_E}) = N$,
$base(\overline{T_E}) = T_B$, $super(\overline{T_E}) = \overline{T_S}$.

The well formed external object predicate wfexo() ensures the the validity of the derivation operation. We will define it later on. Note, that the external supertype $\overline{T_S}$ is optional. If no supertype is given, then the new type $\overline{T_E}$ will have no external supertype. By allowing this omission, we facilitate the definition of multiple unrelated type hierarchies in a single external schema. It is also allowed to derive several different external types from one conceptual type. By combining this multiple external definitions with the unrelated type hierarchies, the view designer can restrict type compatibility in the external schema. But there is no way to loosen up the type incompatibilities of the conceptual schema. Conceptually incompatible types cannot be made compatible in the external schema.

Before we define the wfexo() (well formed external object) predicate, we need some auxiliary definitions:

The subtype relationship in the external schema is defined similar to the conceptual subtype relationship; the object() constructor is substituted by the type derivation derive().

Definition 9 (External Subtyping) *Let S, T be external types, then S is a subtype of T ($S \preceq T$) and T is a supertype of S ($T \succeq S$) if:*

- $S = T$, *or*
- $S = set(S_E), T = set(T_E), S_E \preceq T_E$, *or*
- $S = tuple(l_{S_1}/T_{S_1}, \ldots, l_{S_n}/T_{S_n})$,
  $T = tuple(l_{T_1}/T_{T_1}, \ldots, l_{T_n}/T_{T_m})$,
  $n \geq m, \forall (i = 1, \ldots, m) T_{S_i} \preceq T_{T_i} \wedge$
  $l_{S_i} = l_{T_i}$, *or*
- $\exists S' | S = derive(N, S', T_B, C_P, M_P, M) \wedge S' \preceq T$

Similar to a conceptual object type definition, we compute the defined components and methods of an external type.

Definition 10 (Derived Properties) *Let $\overline{T_E} = derive(\overline{N}, \overline{T_S}, T_B, C_P, M_P, M)$ where $wfexo(\overline{T_S}, T_B, C_P, M_P, M)$ holds, then the set of defined methods for the external type $meth(\overline{T_E})$ is defined as:*

$$meth(\overline{T_E}) = emeth(\overline{T_E}) \cup pmeth(\overline{T_E}) \cup \{m \in meth(\overline{T_S}) | name(m) \notin names(\{emeth(\overline{T_E}) \cup pmeth(\overline{T_E})\})\}$$

The set of components $comp(\overline{T_E})$ of the external type is:

$$comp(\overline{T_E}) = pcomp(\overline{T_E}) \cup \{c \in comp(\overline{T_S}) | name(c) \notin names(pcomp(\overline{T_E}))\}$$

An external or conceptual signature s is externally signature compatible with an external signature t if they have the same name, have the same arity (number of parameters) and all types of t are subtypes or derived from the types of the corresponding parameter of s. This external covariance is also defined for comparing method signatures from conceptual and external schema:

Definition 11 (External Covariance)
$excovar(s, t) :\Leftrightarrow$

$$\begin{aligned} & s = f_s(l_{s1}/T_{s1}, \ldots, l_{sn}/T_{sn})/T_s \wedge \\ & t = f_t(l_{t1}/T_{t1}, \ldots, l_{tm}/T_{tm})/T_t \wedge \\ & f_s = f_t \wedge n = m \wedge \\ & (T_s \preceq T_t \vee T_s \preceq base(T_t)) \wedge \\ & \forall (l_{si}/T_{si}, l_{ti}/T_{ti})\, l_{si} = l_{ti} \wedge \\ & \quad (T_{si} \preceq T_{ti} \vee T_{si} \preceq base(T_{ti})) \end{aligned}$$

The motivation for this definition is that the 'real type' of an object with external type T is a subtype of base(T), i.e. it holds, that each instance of T is also an instance of base(T).

The excovar() predicate is defined for signatures of methods from external and conceptual types. Theorem 2 follows immediately from the definitions. Furthermore, it is easy to show, that the covar predicate is also transitive, which is stated in the theorem 3.

Theorem 2 *$covar(s,t) \Rightarrow excovar(s,t)$.*

Theorem 3 (Transitivity of excovar())
$excovar(s,t) \wedge excovar(t,u) \Rightarrow$
$\quad excovar(s,u)$.

Now we define a well formed object derivation.

We require that the set of projected component names is a subset of the names of the components of the conceptual base type(1).

The type of a component can be redefined to a derived type based on the original type, but subtype compatibility with the corresponding components of the external supertype must still hold (2).

The set of projected method names must be a subset of the names of the defined methods of the conceptual base type (3).

Methods from the projection list can't be redefined (4).

Methods that are explicitly defined for the external supertype or that are projected from the conceptual type, must be covariantly signature compatible with the corresponding methods in the external supertype (5).

Methods that are projected from the conceptual base type, must be externally covariant signature compatible with the corresponding methods in the base type (6).

If an external supertype was defined in the type derivation, then the base type of the newly defined external type must be a subtype of the base type of the external supertype (7).

Definition 12 (Well Formed Object) *For $\overline{T_E} = derive(\overline{N}, [\overline{T_S}], T_B, C_P, M_P, M)$ to be a valid derived type definition, $wfexo(\overline{T_S}, T_B, C_P, M_P, M)$ must hold.*

$wfexo(\overline{T_S}, T_B, C_P, M_P, M))$
$\Leftrightarrow$

1. $names(C_P) \subseteq names(comp(T_B))$

2. $\forall(c \in ecomp(\overline{T_E})), c = l_c/T_c$
$$\forall(d \in comp(T_B)), d = l_d/T_d, l_c = l_d \Rightarrow T_d = base(T_c)$$
$$\wedge \forall(l_e/T_e \in comp(\overline{T_S})) \wedge l_e = l_c \Rightarrow T_c \preceq T_e$$

3. $names(M_P) \subseteq names(meth(T_B))$

4. $names(M_P) \cap names(M) = \emptyset$

5. $\forall(\overline{m} \in M \cup M_P)$,
$$\forall(\overline{m_S} \in meth(\overline{T_S})), \quad name(\overline{m}) = name(\overline{m_S}) \Rightarrow covar(\overline{m}, \overline{m_S})$$

6. $\forall(\overline{m} \in M_P)$
$$\forall(m \in meth(T_B)), \quad name(\overline{m}) = name(m) \Rightarrow excovar(m, \overline{m})$$

7. *if* $\exists \overline{T_S} \Rightarrow T_B \preceq base(\overline{T_S})$

Requirement (7) in the well formed object definition asserts, that when there is an external subtype relationship between two types, then the base types of the external types are in a conceptual subtype relationship. So, there are no possibilities to externally reverse a conceptually defined inheritance relationship. This property is formulated in the following theorem.

Theorem 4 (Subtype Morphism) *Let S, T be external types, then*
$S \preceq T \Rightarrow base(S) \preceq base(T)$.

An important consequence of this theorem is that all assignments and all substitutions of formal parameters in the methods of the external schema and in the application programs do not violate the type compatibility rules of the conceptual schema.

The following theorem formulates, that in a path of the external type hierarchy all methods with the same name are external covariant compatible, independent, whether they are projected from the conceptual model or newly defined in the external model. The theorem follows from the definitions and the transitivity of covar().

Theorem 5 *Let S, T be external types, then*
$S \preceq T, m_S \in meth(S), m_T \in meth(T),$
$name(m_S) = name(m_T) \Rightarrow$
$excovar(m_S, m_T) \wedge$
$covar(m_S, m_T)$.

External schemas are imported into application programs in a similar way as e.g schemas can be imported in O_2. The application programmer can use only the types and methods of the external schema but not the types and classes of the conceptual schema.

The type compatibility for application programs is defined in the usual way. An assignment `a := b` is valid, if the (external) type of `a` is a supertype of the (external) type of `b` (`b` can be a variable or an expression). In analogy, for passing parameters in method calls, the type of the formal parameter is a supertype of the type of an actual parameter.

For new defined methods in external types ($m \in$ emeth(T)) method bodies have to be defined. These methods can use external as well as conceptual types and their methods. Thus, programming external methods can be seen as programming of the interface between application programs and the database. As types in the conceptual and the external schema may have the same name, we establish the rule that external goes before conceptual. If C is both the name of a conceptual type and of an external type, then C refers to the external type and conceptual(C) to the type C in the conceptual schema. The call of methods of conceptual types can be performed in a similar way using the @ notation. For external methods we require the following type compatibility rule: Let T_a be the type of variable `a`, and let T_b be the type of the expression `b`. An assignment `a := b` is valid, if base$(T_b) \preceq$ base(T_a). Substitution of parameters is treated in analogy. This means that the type compatibility of the conceptual schema is relevant for the programmer of methods in the external schemas. So restrictions defined for application programs do not apply for methods in the external schemas.

With this design choice we provide great power and flexibility for external schemas. For example it is possible to define a method new() for the external schema, calling the conceptual method new().

Method Resolution We have several possible ways to define method resolution in the external schema. Here, we will present just one of them. Let us first describe how a method is found in the conceptual schema.

If a conceptual method is explicitly defined for a type, then its origin is the type, else the origin is the origin of the method of the supertype.

Definition 13 (Conceptual Resolution) *Let T be a valid object type definition and $m \in meth(T)$ be a method defined for T, then*
$origin(m, T) = T$, *if* $m \in emeth(T)$

origin$(m, super(T))$*, else.*

One approach to define external method resolution for a method m, is to stay in the external schema, and only to extend the search to the conceptual schema, whenever an explicit reference to a conceptual method is made, i.e. when the method is included in the projection list:

1. If m is defined directly in E, then E is the origin of m.
2. If m is in the projection list of E, then the origin of m is the conceptual origin of m with the base type of the object as a starting point.
3. If m is neither in emeth(E), nor in pmeth(E), then the origin is the origin of the method in the external supertype.

Definition 14 (External Resolution) *Let* $derive(E, S, B, C_P, M_P, M)$ *be a valid object type definition, and* $m \in meth(E)$ *be a method defined for* E*, then*

$exorigin(m, E) =$
E, *if* $m \in emeth(E)$,
$origin(m, base(E))$, *if* $m \in pmeth(E)$
$exorigin(m, S)$*, else.*

Schema Invariants The possibility to derive several external types from one conceptual type poses some problems with respect to proper method resolution. Let two external types T and T' be derived from the same base type B, and the external type S be a common supertype of T and T'. Then a variable o of type S can also contain references to objects whose external type is T or T'. When one sends a message m to o, it is not clear which method body has to be executed.

In order to achieve an unambiguous method resolution and method-steadiness, we define the following schema invariants:

1. If the external type S is a common supertype of T and T' that are derived from the same conceptual type, then all the methods which are defined for S must have an unambiguous origin with respect to T and T'.

$$\begin{aligned}&\forall(T, T'), \mathrm{base}(T) = \mathrm{base}(T'),\\ &\quad \forall(S) T \preceq S, T' \preceq (S),\\ &\quad \forall(m \in \mathrm{names}(\mathrm{meth}(S)) :\\ &\qquad \mathrm{exorigin}(m, T) = \mathrm{exorigin}(m, T')\end{aligned}$$

2. If the external type S is a common supertype of T and S', where T is derived from a subtype of the base type of S', then there must exist a subtype T' of S' which was derived from the same type as T.

$$\begin{aligned}&\forall(S, T, S'), T \preceq S \wedge S' \preceq S \wedge\\ &\quad \mathrm{base}(T) \preceq \mathrm{base}(S') \Rightarrow\\ &\qquad \exists(T'), T' \preceq S', \mathrm{base}(T) = \mathrm{base}(T')\end{aligned}$$

As one can see immediately, schema invariant 1 holds for all external schemas, where there are no two different external types that are based on the same conceptual type. We assume that this class of schemas will be a fairly large one.

Schema invariant 2 holds, if all or none of the subtypes of a conceptual type get mapped into corresponding derived subtypes of its external types.

In the sequel, all schemas adhere to the schema invariants.

Now we can draw some conclusions from schema invariant 1 and summarize them in the following theorem. The theorem could immediatly be used to derive algorithms for a stricter version of the schema invariants.

Again, let two external types T and T' be derived from the same base type B, and the external type S be a common supertype of T

and T'. Then a variable o of type S can also contain references to objects whose external type is T or T'. We distinguish between two different cases according to the origin of m in T:

1. If exorigin(m, T) is in the conceptual schema (so there was an external supertype in which m was projected), then also exorigin(m, T') must be in the conceptual schema according to schema invariant 1. So there must be a supertype D of T', where m was projected in D, and in no type D' between T' and D, the method m was redefined or projected. D is the lowest supertype of T', where m was projected, and no redefinitions occur below D. From the definition of exorigin() we see, that then exorigin(m, T') = exorigin(m, D) = origin$(m, \text{base}(D))$. From schema invariant 1 follows origin$(m, \text{base}(D))$ = exorigin(m, T).

2. If exorigin$(m, T) = D$ is in the external schema (so there is an external supertype D of T in which m was redefined), then also exorigin$(m, T') = D$ must hold according to schema invariant 1. So, D must also be a supertype of T', and in no type D' between T' and D, the method m was redefined or projected.

Theorem 6 (Common Based Types) $\forall (T, T')$ *with* $base(T) = base(T') = B$,
$\quad \forall (S)$ *with* $S \succeq T, S \succeq T', T \neq T'$
$\quad \forall (m \in names(meth(S))$:

1. $exorigin(m, T)$ *in the conceptual schema* $\Rightarrow$

$$\exists D, D \succeq T' \wedge$$
$$m \in names(pmeth(D)) \wedge$$
$$\forall D', T' \preceq D' \preceq D :$$
$$m \notin emeth(D'),$$
$$m \notin pmeth(D'),$$
$$exorigin(m, T') =$$
$$origin(m, base(D)) =$$
$$exorigin(m, T) \wedge$$

2. $exorigin(m, T)$ *in the external schema* $\Rightarrow$

$$\exists D, D \succeq T' \wedge$$
$$m \in names(emeth(D)) \wedge$$
$$D \succeq T' \wedge$$
$$\forall D', T' \preceq D' \preceq D :$$
$$m \notin emeth(D'),$$
$$m \notin pmeth(D')$$

The following special cases of the theorem are worth noting:

1. If method m is projected immediately in T, so $D = T$ then $T' \preceq T$ or $m \in$ names(pmeth(T')).

2. If method m is explicitly defined immediately in T, so again, $D = T$, then $T' \preceq T$ must hold. Also, m must not be redefined or projected in T'.

Now we define the method resolution for the following situation. Let o be a variable of type E, and o contains an object of the conceptual type C. This is only permitted by the type compatibility rule, if C is a subtype of base(E). We now resolve the method invocation $o.m$ with the function fetch. The result of fetch(m, C, E) is a type, where m is directly defined and this method will be executed. We intend to find the most special applicable method, therefore the searches start at an external type T under E, which was derived from the most special type between C and base(E).

Definition 15 (Fetch) *Let D, T be such that $C \preceq D \preceq base(E)$ and $base(T) = D$ and $\forall C \preceq D' \preceq D, D' \neq D : \neg\exists T'$ with $base(T') = D', T' \preceq E$.*

$fetch(m, C, E) = origin(m, C)$, *if*
$exorigin(m, T)$ *in the conceptual schema*
$exorigin(m, T)$, *else.*

In the definition of fetch, T is the external type under E which was derived from the most special conceptual type above C. For convenience we provide the function pfetch which is defined in a more procedural way and can be implemented in a straight forward manner. It is easy to see, that pfetch and fetch are equivalent.

Definition 16 (Pfetch)
$m \in names(meth(E)), C \preceq base(E)$:

$$
\begin{aligned}
&fetch(m, C, E) = \\
&\quad fetch1(m, C, C, E) \\
&fetch1(m, C, D, E) = \\
&\quad origin(m, C), \text{ if } \exists T \preceq E \text{ with} \\
&\qquad base(T) = D, m \in pmeth(T) \\
&\quad T, \text{ if } \exists T \preceq E \text{ with} \\
&\qquad base(T) = D, \\
&\qquad m \in names(emeth(T)) \\
&\quad fetch1(m, C, super(D), E), \text{ if} \\
&\qquad D \preceq base(E) \\
&\quad fetch1(m, C, super(D), super(E)), else.
\end{aligned}
$$

Furthermore, we define the origin of methods for the @ notation. The expression $o.m@C$ has the following semantics: Let E be the type of the variable o, $o.m@T$ is well defined, iff T and E are both external or conceptual types and $E \preceq T$, or T is a conceptual type and E is an external type and base$(E) \preceq T$.

Definition 17 (Fetch at a type)
$fetch(m@, C, E) = origin(m, C)$.

Let H be an external type.

$$
\begin{aligned}
&fetch(m@H, C, E) = \\
&\quad exorigin(m, H), \\
&\qquad \text{if } exorigin(m, H) \text{ is an external type}, \\
&\quad origin(m, C), \text{ otherwise}.
\end{aligned}
$$

Let H be a conceptual type.
$fetch(m@H, C, E) = origin(m, H)$.

Theorem 7 *Fetch is well defined.*

Proof: Fetch is well defined means: a) $\forall m, C, E$, with $m \in$ meth(E), and $C \preceq$ base(E) : $fetch(m, C, E)$ is defined, and b) it is unique.

For a) it is easy to see that there is a nonempty sequence of D_i with $C \preceq D_1 \preceq D_i \preceq D_n \preceq$ base(E), and $T_i \preceq E$, such that base$(T_i) = D_i$, and for all $D' \preceq D_i : \neg\exists T'$ with base$(T') = D', T' \preceq E$.

For b) let D be as in a). Let $T, T' \preceq E$ with base(T) = base(T') = D. Schema invariant 1 requires that exorigin(m, T) = exorigin(m, T'), therefore fetch(m, C, E) is unique independently which T is chosen. □

Theorem 8 $m \in emeth(fetch(m, C, E))$.

This theorem states that the resolved method is directly defined in fetch(m, C, E). It follows immediately from the definition of fetch, origin, and exorigin.

In the following theorem we claim, that the method m in fetch(m, C, E) is covariant compatible with the method m of E, irrespective whether it is a method from a conceptual type or from an external type.

Theorem 9 (Covariant Resolution)

$$
\begin{aligned}
&\forall E, \forall (m \in meth(E), \\
&\quad \forall C \preceq base(E), \\
&\quad \forall m' \in meth(fetch(m, C, E)), \\
&\quad name(m) = name(m') \\
&\quad \Rightarrow excovar(m', m)
\end{aligned}
$$

Proof:

1. fetch(m, C, E) is an external type T. If $T \preceq E$, then the theorem follows from Theorem 5, or $E \preceq T$, then exorigin$(m, E) = T$.

2. fetch(m, C, E) is a conceptual type, say A, and $m \in$ emeth(A).

   (a) If base$(E) \preceq A$, then the theorem follows from Definition 12.6.

   (b) With $B =$ base(E), suppose $A \preceq B$. Let us further denote the method definitions $m_E \in$ meth(E), $m_B \in$ meth(B), and $m_A \in$ meth(A), where all the methods m_i have the same name. With Theorem 6.1 we know that $\exists D$ and T with base$(T) = D$, $C \preceq D$, with $m_T \in$ pmeth(T), $m_D \in$ meth(D), where excovar(m_D, m_T).

   Now we have to show that excovar(m_A, m_E). Since $A \preceq B$, covar(m_A, m_B) holds.

   We can distinguish three cases, depending on the position of T and A in relation to E and D respectively:

      i. Let $E \preceq T$, then E inherits m_T from T ($m_E = m_T$). Also, $B \preceq D$ according to the subtype morphism. From covar(m_A, m_B) and the transitivity of covar(), we see that covar(m_A, m_D). Now with excovar(m_D, m_T) we conclude that excovar(m_A, m_T). Since $m_E = m_T$, the theorem follows.

      ii. Let $T \preceq E$, and $A \preceq D$. Then covar(m_A, m_D) and excovar(m_T, m_E) holds. Now, with excovar(m_D, m_T), the theorem follows.

      iii. Let $T \preceq E$, and $A \succeq D$. Then D inherits m_A from A ($m_D = m_A$) and trivially, excovar(m_T, m_E) holds. From excovar(m_D, m_T) we conclude that excovar(m_D, m_E). Since $m_D = m_A$, the theorem follows. □

Theorem 10 (Method Steadiness)

$$\begin{aligned}&\forall (E, E'), E \preceq E',\\ &\quad \forall (C \preceq \mathit{base}(E)),\\ &\quad \forall m \in \mathit{meth}(E')\\ &\quad \Rightarrow \mathit{fetch}(m, C, E) = \mathit{fetch}(m, C, E')\end{aligned}$$

Proof: Let m, C, E and E' be as above. Suppose that fetch$(m, C, E) = A$. Let D, and T be as in the definition of fetch. Since $D \preceq E \preceq E'$, fetch$(m, C, E') = A$.

Now suppose fetch$(m, C, E') = A$. Let D' and T' be as in the definition of fetch. Since $C \preceq$ base(E), $D' \preceq$ base(E). With schema invariant 2 we know $\exists T \preceq E$ with base$(T) = D$, and the theorem follows from schema invariant 1. □

The theorem has an important consequence which we call method-steadiness, i.e. if we assign an object to a variable of a supertype, the executed methods remain the same. To give an example:

Let e be a variable of external type E and e' a variable with type E' which is a supertype of E. Then for $e.m$ and $e'.m$ the same method body is executed, given they contain the same object. Therefore, an assignment $e' := e$ does not influence which method bodies are processed. Due to the type compatibility requirement this holds for all assignments which can be made on such variables in application programs.

However, for the methods defined in the external models, method-steadiness can only be guaranteed, if the programmer remains within the strict type compatibility. The looser type compatibility requirement for these methods may lead to method-unsteadiness, and the view programmer has to take care of the effects. Nevertheless, this looser type compatibility requirement is well founded and leads to greater flexibility for the definition of views.

The following example illustrates the principles and constructs presented above.

An Example, Part 2 We derive three external types in the `external schema E` derived from the conceptual university schema; `EPerson` is derived from `Person`, `EAdvisor` is derived from `Advisor` and inherits from `EPerson` and `AnonStudent` is derived from `Student`. Note, that there is no external inheritance from `EPerson` to `AnonStudent`, the conceptual inheritance relationship is not visible in the external schema. From `Person`, only the `name` is in the projection list, all other properties are hidden from the applications using the external schema. In type `EAdvisor`, two additional methods are defined, the `new()` method allows to create new conceptual objects at the external level. Derived type `AnonStudent` does not have any attributes defined for its base type `Student`, but redefines the conceptual method `older`, which `Student` inherited from `Person`. In the redefinition, the parameters type is redefined. The external method `set_advisor()` of `AnonStudent` allows one to define the advisor for an anonymous student. In the conceptual schema, we had no corresponding method, but there we defined `Advisor::add_candidate()`. This method is called in `AnonStudent::set_advisor()` via the `@` notation.

In the example, we defined an external inheritance hierarchy, which was different from the conceptual one. We left out some attributes and methods of the conceptual schema and defined new external methods, which make use of conceptual methods.

```
external schema E
of
university {

  derived interface EPerson
  of
  Person {
    project {
      attribute string name;
    }
  }

  derived interface EAdvisor:EPerson
  of
  Advisor {
    project {
     attribute date birthdate;
    }
    int more_than(e:EAdvisor);
    EAdvisor new(aname:string);
  }

  derived interface AnonStudent
  of
  Student {
    project {
      boolean older(s:AnonStudent);
    }
    void set_advisor(a:EAdvisor);
  }
}


int EAdvisor::more_than(e:EAdvisor) {
  return card(candidates) >=
         card(e.candidates);
}
```

```
EAdvisor::new(aname:string) {
  return new(aname, {})@;
}

void AnonStudent::
     set_advisor(a:EAdvisor) {
  a.add_candidate(self)@;
}
```

4 Conclusion

We presented a concept to integrate external schemas in object oriented databases. This additional level of abstraction offers logical data independence and a greeter degree of modularity in information systems.

A clean separation of the conceptual schema and the external schemas was achieved by type derivation in combination with updateable views.

The covariant subtyping of the conceptual schema is fully preserved in the external schema. The external schema adheres to the principal type compatibility that the conceptual schema defines. It is not possible to make conceptual incompatible things externally compatible. On the other hand, the external level can introduce additional compatibility constraints, making conceptually compatible things incompatible in the external context.

Through the introduction of schema invariants, we were able to provide covariant method resolution and method steadiness.

Since we allow to define methods in the external schema, which can call other external as well as conceptual methods, the designer of the external model can implement methods which overlap the schemas. Such inter schema methods can be used to implement additional powerful mappings and thereby provide better schema derivation and integration facilities.

In particular, object creation at an external level is perfectly feasible. The external schema just has to provide a new() method, which takes care of the necessary mapping (calls the corresponding conceptual new() method) and provides default values for components that are not visible to the application program.

A crucial point of a schema is that it is closed, which means for types, that all types that are needed in the schema are also defined there. As an aid for the schema designer, we will provide a type covering operator which will define a derived type for an external type similar as the derive() operator does. But the covering type will have the same name as the covered type. A covered conceptual type will automatically be substituted by the external covering type in the whole schema at all places where no explicit type substitution occurs.

Presently we excluded multiple inheritance on the conceptual level as well as on the external level from our considerations for the sake of simplicity. But since multiple inheritance is supported in ODMG and also widely offered by almost all of the current object systems, we are planning to incorporate it into the model. Since renaming is an adequately conflict resolution strategy, we will use it, and as a byproduct we will be able to do general renaming between the conceptual and external schema.

So far, we concentrated on the intensional mechanisms for type derivation. Further on, we will investigate class construction and derivation concepts. A suitable approach for this seems to be to define a derived class (view) by means of a query over conceptual classes (extents). Besides updateable views via object preserving queries, we will provide non-updateable views via object generating and value generating queries. Restructuring mechanisms for the class hierarchy will be incorporated in the approach along with corresponding schema invariants to ensure well formed external schemas.

References

[1] R. Agrawal and L. DeMichiel. Type derivation using the projection operation. Extended Version of [2], by personal communication with R. Agrawal, 1994.

[2] R. Agrawal and L.G. DeMichiel. Type derivation using the projection operation. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Advances in Database Technology - EDBT'94*, pages 7–14. Springer, 1994.

[3] A. Borgida. Modeling class hierarchies with contradictions. Technical report, Rutgers University, New Brunswick, 1988.

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

[5] M. Dobrovnik and J. Eder. A concept of type derivation for object oriented database systems. In L. Gün, R. Onurval, and E. Gelenbe, editors, *8th Intl. Symposium on Computer and Information Sciences (ISCIS VIII)*, 1993.

[6] M. Dobrovnik and J. Eder. View concepts for object-oriented databases. In *Proc. Intl. Symposium on System Sciences, Informatics and Cybernetics, Baden-Baden*, 1993.

[7] M. Dobrovnik, K.-H. Eder, L. Böször mén yi, and J. Eder. An updateable view system for oodbms. Technical report, Institut für Informatik, Universität Klagenfurt, May 1994.

[8] C. S. dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Advances in Database Technology - EDBT'94*, pages 81–94, Cambridge, 1994. Springer.

[9] S. Heiler and S. Zdonik. Object views: Extending the vision. In *6th International Conference on Data Engineering*, pages 86–93, 1990.

[10] W. Kim. A model of queries in object oriented databases. In *Proc. of 15th VLDB Conference*, pages 423–432, Amsterdam, August 1989.

[11] E. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. 18th VLDB Conference, Vancouver*, 1992.

[12] M. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The cocoon object model. Technical Report TR 192, Institut für Informatik, ETH Zürich, 1992.

[13] D. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS framework. Report of the Study Group on Database Management Systems, Information Systems 3, 1978.