Object Views of Relations

C. Delobel
LRI-Univ. Paris-Sud, INRIA

C. Souza dos Santos
INRIA-Rocquencourt

D. Tallot
O_2 Technology

February, 1995

Abstract

This paper investigates the problem of integration between relational and object-oriented databases. We discuss an approach based on *class* and *attribute mappings* and show how OQL, the ODMG query language, can be embedded in a set of primitives of a mapping language to serve as a basis for object-relational data integration. We then describe the technique used in the implementation of a prototype built on top of an object-oriented view mechanism, which allows the construction of an object view of a relational database.

1 Introduction

Many new applications have to manipulate *legacy data* (data existing in traditional database systems) in addition to new data manipulated with the support of the new generation of object database systems (ODBMSs). ODBMSs provide advanced technology for complex applications but their use is not yet so widespread. Therefore, the problem is to migrate or integrate these legacy data into object based environments without being forced to redevelop existing applications from scratch. Relational database products have been used for more than a decade, specially in business applications. Interoperability between ODBMS and RDBMS is thus essential.

There are basically two approaches: *integration* and *migration*. In the former approach, the object and relational databases co-exist. Integration is provided by gateways from relational to object data and vice versa, or by coupling heterogeneous database systems in a federated database system. In the migration approach, relational data are moved into the object environment. Once the migration is accomplished, user applications interact with the ODBMS only.

Conceptually, database migration and integration must address the four major components of any database application: schema information (metadata), data, applications and interfaces. In this paper, we focus on the first two (metadata and data) since data are probably the most critical resource for economical and organizational reasons.

When migrating or integrating data, we must consider the source and the target model. In our framework, the source model is the standard relational model where the basic concepts are: relation, attribute, atomic type and integrity constraints (functional and inclusion dependencies), whereas the target model is the ODMG object model [10] where the basic concepts are: object, class, type, inheritance and method. The problem consists in defining correspondences between the primitive concepts of both models.

Intuitively, we must generate the class and the composition hierarchies (i.e. attribute and inheritance links) for an object schema given a relational schema. Thus, in our approach, there are two basic mappings: the *class mapping* and the *attribute mapping*. A class mapping expresses how an entity set (i.e. a relation) is related to a class. An attribute mapping expresses how a relationship or association is mapped to an attribute of a class. We will see in the following sections a more detailed presentation of these concepts, where it will become clear that these mappings must not only express the association between a relational and an object schema (metadata mapping), but also give the way to compute the instances of the mapped classes and their attribute values (data mapping).

In general, migrating/integrating a relational to/with an object database is very complex. Therefore, we want to provide a set of flexible and loosely coupled tools for (i) helping the designer to extract and capture the semantics of the relational application, (ii) checking the consistency of the possible transformations, (iii) generating an object schema and loading an object database. As part of that goal, we have designed and developed a prototype that connects the O_2 ODBMS [4] to any relational system and constructs an object view of a relational database.

We show that OQL, the object query language of ODMG, embedded in the primitives of a given view definition language, provides a full specification of the mappings, with a complete restructuring of the relational database. We provide an implementation based on an object view mechanism where many different imported schemas can be federated to form a unified view. In addition, the system architecture can be used both for migration and integration, depending on the way view data are managed and on the ability to materialize views.

The rest of the paper is organized as follows. Section 2 describes the migration process through a running example. Section 3 defines class and attribute mappings. Section 4 presents the architecture of the migration system and describes the current prototype implementation. Section 5 shows some related work and section 6 concludes and points to some future work.

2 Mapping Methodology

To illustrate the mapping process, we use an example, borrowed from [24], and we analyze the different steps of the process. This methodological aspect is needed to understand mapping definition and it is closely related to the problem of data model transformations [26, 18, 19]. This problem has been largely addressed for many years in the database community. In the relational model, dependency theory has provided a formal framework for proving relational schema equivalence. Results about schema equivalence [12, 25, 20, 28] for different data models are more limited and this is still an open problem. Furthermore, these limited results have not proven to be useful in practical situations. One can propose some simple transformations which guarantee a natural correspondence between schemas.

Our methodology is organized around four steps:

1. Description of the relational schema information.
2. Semantic description of the relational universe.
3. Enrichment of the relational universe.
4. Mapping definitions.

These steps are explained in the sequel.

Description of the relational schema information

At first, we consider that all information concerning the relational schema is mapped to a uniform representation where all the meta information is recorded. The meta data concerning the relational schema can be obtained by analyzing the source code or, in the case of a connection to a relational system, by querying directly the relational dictionaries with the SQL language. This step is purely syntactic.

Figure 1 shows a relational schema composed of seven relations about conference organization. We have omitted the type of each attribute for the sake of simplicity. Underlined attributes represent the keys of each relation.

```
conference(conf_id, title, opening_day, closing_day)
conference_info_address(conf_id, name, mail_address)
session(conf_id,session_id, chair_name)
lecture(conf_id,session_id,lecturer_na me)
hotel(hname, fax, city, country)
reservation(conf_id, hname)
person(name, address, title, affiliation)
```

Figure 1: Relational schema `Conference`

Most of the relations are self explanatory. The relation `conference_info_address` gives information about the person to contact. A conference consists of sessions, the `session` relation expresses the relationship with the session chairman. A session of a conference is composed of lectures, this is the role of the relation `lecture`. Conferences take place in an hotel site, the relation `hotel` gives the list of hotels with their name and fax. The `reservation` relation represents the relationship between a conference and a hotel.

Semantic description of the relational universe

The second step is a semantic description of the relational universe and adds more information on the entities manipulated in the application. Representations based on extensions of the Entity-Relationship model like, for example, the *Schema Intension Graph* (SIG) whose formalism has been defined in [27] or the *Partial Universal Relation* (PUR) [31] approach, are adequate tools for tasks such as schema integration, model transformation, and proof of schema equivalence. These representations are powerful enough to express a relational model with functional and inclusion dependencies. Key dependencies are useful when dealing with the object identity concept and inclusion dependencies are the basis for the derivation of inheritance relationship. Furthermore, key and inclusion dependencies interact in many ways and algorithms for studying this interaction and the generation of relational schema under these dependencies have been developed in [8, 31].

Nevertheless, it should be pointed that extracting knowledge from a relational application is very often biased. A recent paper [32] about the reverse engineering of relational databases proposes a list of sensitive aspects during the re-implementation phase. It gives guidelines for discovering candidate keys, generalizations and associations in a relational schema. The authors also point, based on their experiences, that in many real relational applications the schema violates most of good design rules, therefore making properties discovering harder.

A PUR representation of the relational schema is given in Figure 2. It is a graphical representation where the nodes represent complex structures. There are two PURs in Figure 2, which partition the set of relations in the schema into semantical clusters. Inside a PUR, the nodes denote relations and the directed edges a partial order relation based on the inclusion of sets of functional dependencies. In our example, a simple interpretation of this partial order is: if the relation `lecture` is connected to relation `session` the closure of functional dependencies associated with `lecture` is included into the closure of the functional dependencies associated with `session`. This means that a `lecture` tuple uniquely determines a `session` tuple.

The edges between two PURs express in turn the inclusion dependencies. For example, the edge between the relations `conf_info` and `person` means that the values associated with the attribute `name` are included in the values of the attribute `name` of the relation `person`. Algorithms for building PUR representations are given in [31].

Enrichment of the relational universe

The third step is an enrichment of the second step based on the graphical representation. For example, we can create new entity sets, we can partition entity sets or we can add new attributes. The enrichment process is based mainly on two basic concepts of semantic modeling: *inheritance* and *aggregation*. Guidelines for developing inheritance relationships or building aggregate objects will follow the normal procedures existing in Object Oriented Analysis and Design Methods [7, 35]. To discover inheritance relationships, we can consider a wide range of strategies derived from different conceptual approaches. Since the relational data model does not offer inheritance modeling as a primitive concept, various decisions and optimization strategies can be used to implement an inheritance hierarchy within the scope of the relational model. To generate the hierarchy, we can be assisted by the PUR

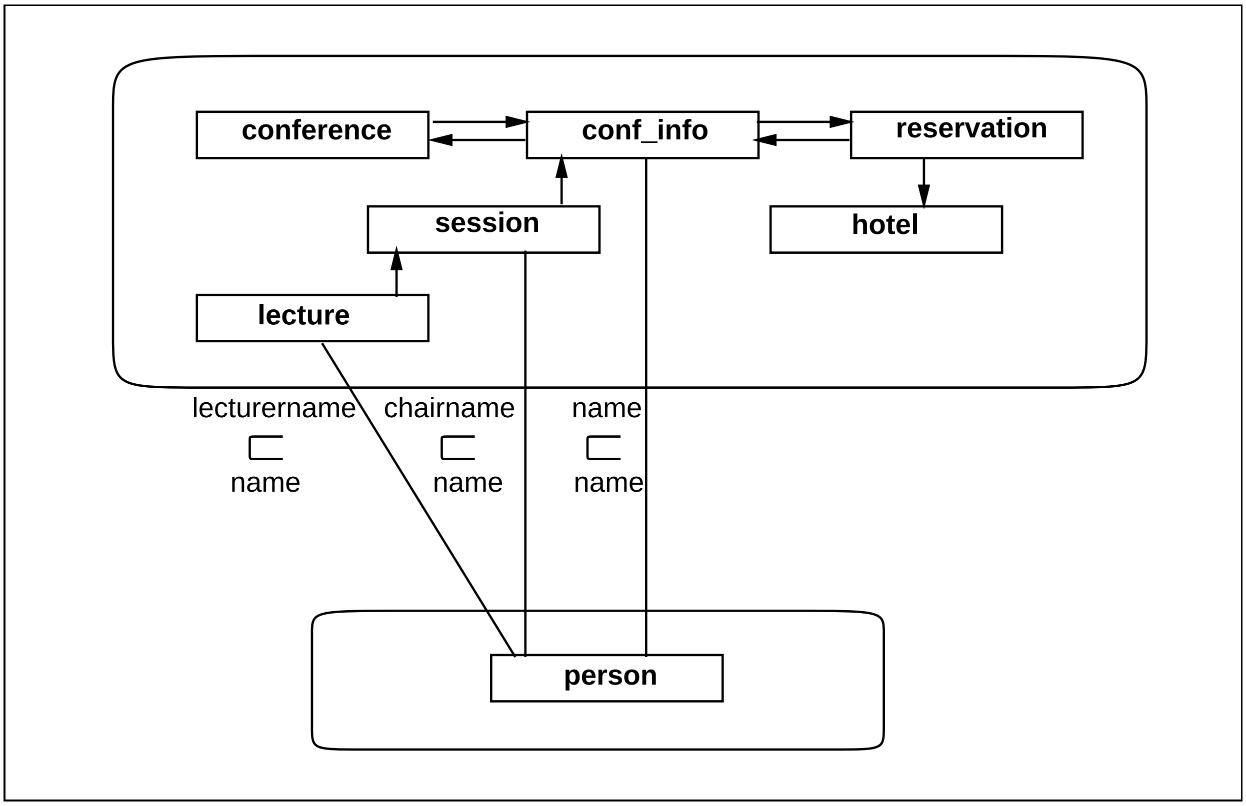

Figure 2: A PUR representation of the relational schema

representation where the inclusion dependencies are given. A complete study of this problem is out of the scope of the paper.

As far as data capacity is concerned, the object schema can be greater, smaller or incomparable with respect to the relational schema. Data capacity augmentation corresponds to data completion, i.e. adding new information which is not part of and cannot be derived from the original relational database. Adding inheritance and creating inverse links, in turn, are data conservative semantic enrichments.

In the example, the schema designer may propose the following modifications:

- A new entity set `CITY` is generated from the atomic value extracted from the attribute `city` in the relation `hotel`. This transformation is an example where a lightweight attribute is promoted to an entity set because we are adding information that is not contained in the relational universe and that we want to provide a new interface management.

- The attributes `photo` and `cv` have been added as multimedia data type as properties about person.

- The concept of person is specialized into chairman and lecturer. Such decisions are under the responsibility of the designer.

- The class `MAP` has been added to integrate a multi-media dimension. The designer wants that when the user starts the object application he/she sees a map on the screen with the hotels holding a conference. These hotels are active objects from which we can navigate inside the database.

Mapping definitions

The fourth step is the description of the final mapping between the relational representation and the object model. In our proposal, the mapping consists of two parts: the *class mapping* and the *attribute mapping*. The class mapping specifies relationships between entity sets and object classes. The attribute mapping maps a relationship to an attribute of a class. These mappings do not only express the association between the relational and the object model, but also how to compute the instances of a class and the attribute values. The mapping is specified with a mapping language, which is presented in the next section. When the designer specifies the mapping, he must have in mind the target object model. Compiling the mapping statements will produce automatically the object schema and an application program for loading the relational data into an object space.

For example, we can consider the O_2 schema shown in Figure 3. The class behaviors (methods) are not specified and are added at a later stage.

This schema integrates most of the transformations that a designer would like to define when building a new database:

- The instances of the class `CONF_EVENT` are the image of the tuples in the relation `conference`. Nevertheless, the key attribute `conf_id` does not exist because the instances of class `CONF_EVENT` have their own object identity and that new composite links were created to relate other entities to a conference (e.g. attribute `conf` in class `SESSION`)[1].

- The structure of the class `CONF_EVENT` includes a tuple value for the attribute `conf_info`, a set of objects for the attribute `sessions` and a reference to an object for the attribute `hotel`. In fact, the relational schema defines a vertical partitioning [30] of the entity conference into two relations `conference` and `conf_info_address`. In the object schema the transformation preserves only a unique class conference but with a nested structure for the conference information.

- The instances of classes `SESSION`, `LECTURE`, and `HOTEL` are respectively the image of the tuples of the relations `session`, `lecture` and `hotel`. This is an easy and frequent transformation where each relation is mapped to a class. Nevertheless, the class structure is not always strictly equivalent to the corresponding relation structure and a relation is not always mapped to a class.

- The `reservation` relation has disappeared. This relationship has been captured by object references between instances of the class `HOTEL` and `CONF_EVENT`.

- The `PERSON` class has been specialized into `CHAIRMAN` and `LECTURER`.

[1]The key may be useful in some applications so it is a choice to keep it or not. For instance, it may appear in some official forms.

```
class CONF_EVENT type struct(
   title : string,
   opening_day : Date,
   closing_day : Date,
   conf_info : struct(
           name : Person,
           mail_address : string),
   sessions : set(SESSION),
   hotel : HOTEL)
end;

class SESSION type struct(
   conf : CONF_EVENT,
   chairman : CHAIRMAN,
   lectures : set(LECTURE))
end;

class LECTURE type struct(
   session : SESSION,
   lecturer : LECTURER)
end;

class HOTEL type struct(
   hname : string,
   conf : set(CONF_EVENT),
   city : CITY)
end;
```

```
class CITY type struct(
   cname : string,
   country : string,
   map : Map)
end;

class PERSON type struct(
   name : string,
   photo : bitmap, cv : Text)
end;

class LECTURER inherit PERSON
   type struct(affiliation:string)
end;

class CHAIRMAN inherit PERSON
   type struct(title:string)
end;

class MAP type struct(
   background : bitmap,
   foreground : list(struct(
             xhotel : real,
             yhotel : real,
             hotel : Hotel)))
end;
```

Figure 3: O_2 schema for a relational schema

- The class `MAP` has been added.

Intuitively, one can see and understand, in this example, that the object schema is a correct and consistent representation of the relational schema. Nevertheless, the equivalence must be based on formal derivations and/or proofs.

For example, we can derive the fact that in the class `CONF_EVENT` the attribute `hotel` has the type `HOTEL`. This is valid because we can infer the following assertions: (i) the image of the relation `conference` is the class `CONF_EVENT`, (ii) the key of the relation `conference` is `conf_id`, (iii) the attribute `conf_id` determines an hotel tuple, (iv) the relation `hotel` is mapped to a class `HOTEL`.

More work is needed to provide tools to automatically establish the correctness of the target schema.

3 Schema Mapping Definitions

The mapping between the relational and the object schema defines the relationship between basic elements of the two data models. For example, a relation or part of a relation may be mapped to a class and a column of a relational table may be mapped to an attribute. Therefore, we consider two basic mappings: *class mapping* and *attribute mapping*. This approach is very similar to the definition of an object view mechanism [1, 6, 38].

A class mapping consists of four elements:

1. The name of the class associated with the mapping.
2. The name of the extension of the class (if this is not provided by the model).
3. The external key and the base relations from which the class is derived.
4. A query expression to generate the class extension from relations.

```
create class mapping CONF_EVENT
   extension The_CONF_EVENTs
   external key conf_id: integer from conf_id in conference
   import attributes(title: string,
                     opening_day: Date,
                     closing_day: Date)
   includes( define The_CONF_EVENTs as
                    select CONF_EVENT(conf_id: c.conf_id,
                                      title: c.title,
                                      opening_day: Odb_date(c.opening_day),
                                      closing_day: Odb_date(c.closing_day))
                    from c in conference)
```

Figure 4: A Class Mapping Definition

The mapping is purely declarative and not procedural. It is mainly based on an OQL query introduced in the `includes` clause. The scope of an OQL expression may contain some relational or object names. This is possible if we consider the object schema as composed of both the object class declaration and the relational names (recall that a relation can be considered in the object world as a set of tuples).

In the example, the class `CONF_EVENT` is the image of the `conference` relation. The name of the extension of the class is defined. The external key establishes, for system implementation purposes, a link between the internal oid of each instance of the class `CONF_EVENT` and the associated key. The `import` clause defines the attributes whose values are taken directly from the corresponding attributes in the `conference` relation. The `includes` clause defines how the instances of the class are computed from the relational tuples: for each conference tuple an object is constructed with the `CONF_EVENT` constructor. In this constructor, a function for converting a relational `Date` type into an object `Date` type is applied.

An attribute mapping is a procedure defining how to compute an attribute value. In the simple case of the `title` attribute, it is not necessary to define an explicit attribute mapping since this procedure is specified with the imported attribute list in the class `CONF_EVENT`. For the other attributes the designer should define an attribute mappings. Examples are given in Figure 5 for the class `CONF_EVENT`.

```
create attribute mapping conf_info :struct(name: Person, mail_address: string)
   in class CONF_EVENT
   with value ( define info as element ( select struct(name: c.name,
                                                        mail: c.mail_address)
                                         from c in conf_info_address
                                         where c.conf_id = self.conf_id ) in
              struct(name: element( select p
                                    from p in The_PERSONs
                                    where p.name = info.name)
                     mail_address: info.mail))

create attribute mapping sessions: set(SESSION) in class CONF_EVENT
   with value select s
              from s in The_SESSIONs
              where s.conf_id = self.conf_id

create attribute mapping hotel: HOTEL in class CONF_EVENT
   with value element(select h
                      from h in The_HOTELs, r in reservation
                      where r.conf_id = self.conf_id and
                            r.hname = h. hname)
```

Figure 5: Attribute mappings

It is interesting to note:

- The cores of these specifications are OQL expressions. Some query expressions use the parameter `self` to denote the current element. We chose here the OQL language because it is the standard ODMG query language, but other alternatives could be used. For example, the IQL language [2] provides a deeper insight for the interpretation of the object creation. It has also a clear semantics but has no product implementation.

- When an OQL expression contains a relational name, we must convert part of the expression to a pure SQL expression for accessing the relation. The simplest strategy is to load the relation in the object space but this can be very time consuming. For example, to execute the instance population of the class `CONF_EVENT`, we would like to avoid loading the `conference` relation. Each time we fetch a tuple from the relational system, we convert the tuple to an object. Another example will illustrate this implementation aspect. The specification of the attribute mapping `hotel` of type

`HOTEL` in the class `CONF_EVENT` contains a reference to the `reservation` relation. One strategy for evaluating the value of the attribute is to load the `reservation` relation in the object space, this could be a good access strategy if the relation is small. Another strategy is to load in the object space only the part of the relation `reservation` needed by the query. These examples show that we need a query evaluator that minimizes transfer from the relational to the object space.

- Specifications of class and attribute mappings are given in a declarative mode and are completely independent of any ordering. The order of evaluation is defined according to a graph of dependency derived from the mapping declarations.

4 An Integration System Architecture

In this section, we present the architecture of the object-relational integration system illustrated in Figure 6.

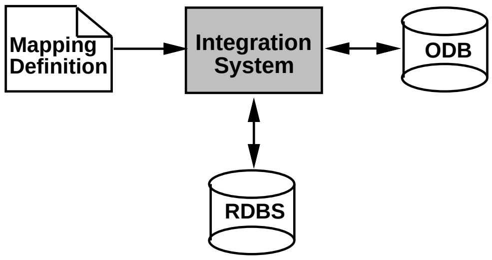

Figure 6: The general architecture

The gray box in Figure 6 represents the integration system, which interacts with a relational and an object-oriented database system and takes a mapping definition as input. The architecture of such a system will be depicted below.

The integration system can be seen as composed of the following main components:

- A *Driver* is a low-level communication link between data servers which performs basic data format conversion and allows data transfer to be performed.

- An *Object-Relational Interface* (*ORI*) is built on top of a *Driver*. The role of this interface is:

  1. to initialize a communication between an object database client process and a Relational Database Server (RDBS);
  2. to pass SQL queries to the relational server;
  3. to retrieve the tuples qualified by the query and convert them into object values or pure objects.

The *ORI* is thus a higher level interface to the *Driver* which encapsulates low level functions in a library specific to a given target object model.

- A *Mapping Compiler* transforms the *Mapping Definition* into an *Object-Oriented Schema* and generates a set of *API Functions* to be used by the *Loader*.
- The *API functions* are a more specific set of functions on top of the *ORI* which implement a given mapping definition language. They consist mainly of functions for building class extensions and for defining the values of attributes (class and attribute mapping implementations).
- A *Loader* is an application program used to create the object database. It uses only the *Object Schema* and the *API Functions* generated by the *Mapping Compiler*.

We believe the components above implement the main tasks involved in the integration process in a modular way, as it will become more clear in the following description of our current prototype implementation.

Current Implementation

Our current prototype is built on top of the O_2Views mechanism [37, 36]. We have extended this mechanism to provide an object view of a relational database. The architecture of the system is shown in Figure 7.

O_2Views is an object-oriented view mechanism built on top of O_2. It implements a view definition language whose main primitives are *virtual classes*, *imaginary classes*, *virtual attributes* and *virtual names*.

A virtual class is used for redefining the structure and behavior of existing objects, which are put into a new class (a virtual class) where their structure and behavior can be redefined. An imaginary class allows the construction of new objects in the view from existing objects and values. Virtual attributes are used to enrich the structure of virtual and imaginary objects with attributes whose values are computed (given by a query). Finally, virtual names are temporary global data in the view.

The mapping definition is implemented by a view definition using an extension of the O_2Views language in which some primitives have been adapted to allow the construction of an object-oriented view from a relational database.

In the current prototype, class mappings are defined through *imaginary classes* and attribute mappings correspond to *virtual attributes*. They are both defined using the source and the target query language (SQL and OQL respectively), in a declarative style.

In addition, virtual names (also defined through a query) are used to define temporary data. For instance, virtual names can be used to import relations into sets of tuples in O_2 rather than into classes. Virtual classes were not used since there are no objects in the source (root) relational base to be redefined in the view, all objects being constructed from relational data through the imaginary class primitive. Nevertheless, virtual classes can be used in a later stage to build different views from the object view of a relational database.

The O_2Views compiler generates, given a view definition, a standard O_2 schema implementing the view. This schema is provided with a set of auxiliary functions (called

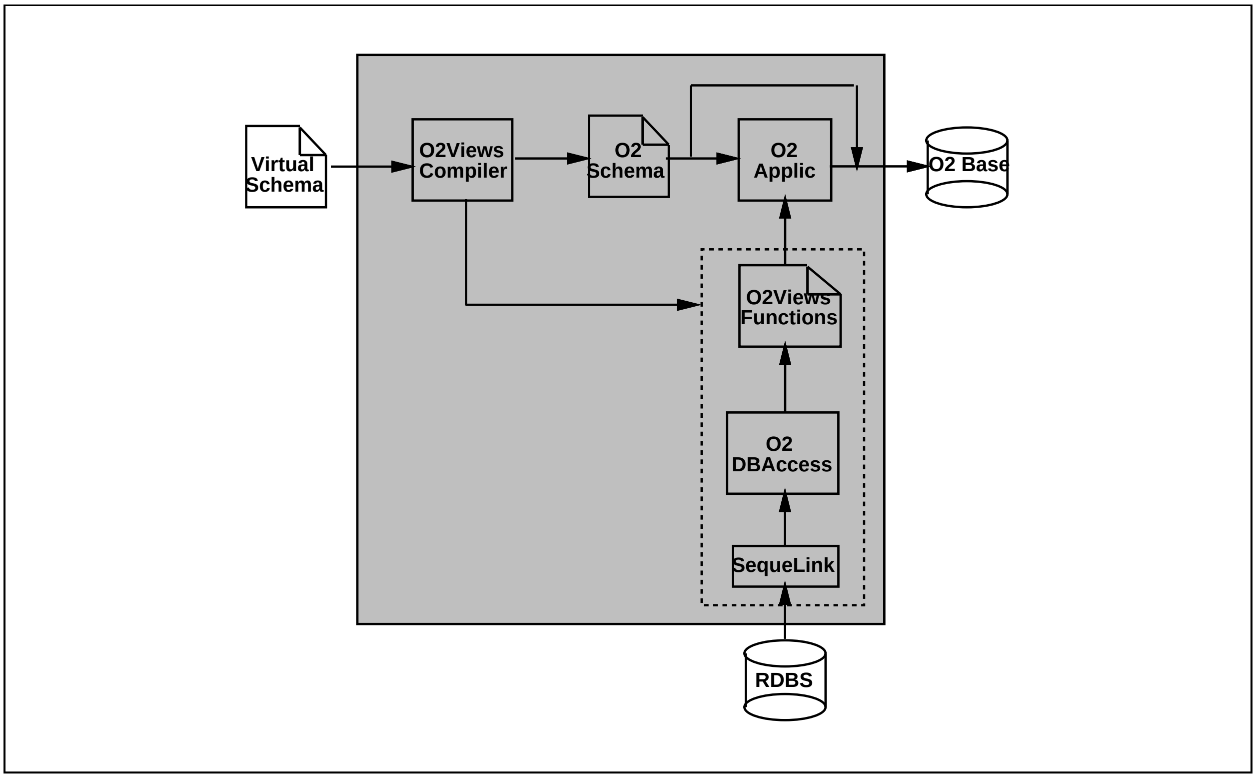

Figure 7: The O_2Views based prototype architecture

system-supplied or API functions) for view object construction, view consistency checking, data conversion, among others. These functions implement basic operations of the view mechanism, as described in [36].

In the current implementation, some small changes to the view definition language syntax were performed to allow the use of SQL in addition to OQL in view definitions and the specification of some properties of the relational schema (e.g. key information). An example of view definition is given in Figure 8 which corresponds to the class mapping given in Figure 4 and some of the attribute mappings given in Figure 5.

The view compiler generates functions that make the connection with *O_2DBAccess*, an *ORI* implemented on top of the *SequeLink Driver*[2]. This way, for computing the extension of imaginary classes, for instance, all communications with the relational database are performed from the query evaluation auxiliary function (named `Evaluate_Query_For_`*class_name*), through calls to O_2DBAccess services (e.g. passing the SQL query to the RDBS, informing the type of the result, executing the query and fetching qualified tuples). Such query evaluation function is in turn used by the imaginary class extension update function (named `Update_`*class_name*). This is illustrated in Figure 9.

Function `Update_CONF_EVENT` uses another compiler generated function, namely the conversion function `As_CONF_EVENT`, which, given the core attributes builds the corresponding imaginary object. This function can be used to construct imaginary data incrementally in-

[2] *SequeLink* is a registered trademark of TechGnosis Inc.

```
imaginary class CONF_EVENT
core attributes conf_id:integer
extension The_CONF_EVENTs
includes sql
   select conf_id from conference;
end;

virtual attribute sessions:set(SESSION) in class CONF_EVENT
has value
  select s
  from s in The_SESSIONs
  where s.conf_id=self->conf_id;
end;

virtual attribute hotel:HOTEL in class CONF_EVENT
has value
  element(
  select h
  from h in The_HOTELs,
       r in reservation
  where h.hname=r.hname and r.conf_id=self- >conf_id);
end;

virtual name reservation:set(tuple(conf_id:integer,hname:string))
has value sql
  select conf_id,hname from reservation ;
end;
```

Figure 8: Class and attribute mappings defined through a view

stead of computing the whole imaginary class extension. The evaluation of virtual attribute values is processed analogously from the body of conversion functions.

Once the view is compiled, the standard O_2 schema and the auxiliary generated functions can be used to build applications that instantiate data in the view, i.e., that derive imaginary objects from the relational database.

In the O_2Views approach, data are not physically materialized in the object database and exist only during view activation. The loader, which corresponds to an application on the view, allows the construction of objects in the view (imaginary objects) which become the image of relational data stored in the root base. This is an advantage over a strict migration approach.

5 Related Work

Several projects and approaches address the problem of database integration [39, 17, 24], which is closely related to that of database migration [11, 33]. Views are used in many of

```
function body Evaluate_Query_For_CONF_EVENT:unique set(tuple(conf_id:integer))
{
 o2 unique set(tuple(conf_id:integer)) aux;
 context->associate(" select conf_id from conference");
 context->define_projection(aux,list("conf_id"));
 context->exec;
 context->fetch(0);
 return(aux);
};

function body Update_CONF_EVENT:unique set(CONF_EVENT) {
 o2 tuple(conf_id:integer) t;
 o2 unique set(CONF_EVENT) aux;
 for (t in Evaluate_Query_For_CONF_EVENT())
      aux += unique set(As_CONF_EVENT(t.conf_id));
 return(aux);
};
```

Figure 9: Extension function for the class CONF_EVENT

these approaches [29, 13, 34, 21, 16, 17]. They basically describe multi database models which are mapped to a uniform model. This requires the translation of a source to a target model through a data definition language, as well as the transformation of data formats (data conversion). Some approaches also deal with *semantic integration* associated with some *semantic enrichment* [5, 9].

In particular, the IQL model [2] integrates both relational and object-oriented features in a uniform data model and allows the derivation of objects with identifiers from relational data using an inflationary fixpoint semantics. A similar approach can be found in [23].

The idea of automatically derived class populations appeared first in the notion of virtual and imaginary classes [1]. A similar idea has been used in the Pegasus [3, 15] system, based on an import mechanism. Pegasus translates a relational schema into an Iris [14] database schema. It provides mapping facilities for generating imported schema but also translates Pegasus queries into the data manipulation languages of the external databases.

More recently, gateways between C++ application programs and data in relational database appeared in the market (e.g. Persistent [22]). These products basically try to overcome the impedance mismatch between relational and object domains: programmers developing C++ programs spend most of the time programming the interface to the relational database. To improve programmer productivity, these systems automatically generate relational database accesses embedded into C++ classes. These products appear more as providing a programming interface than a view or an imported schema.

Our approach is based on an existing object-oriented view definition language. Given a basic mapping between atomic types in the two different data models (implemented through some kind of driver), data integration can be specified by a view definition whose embedded queries are written in the source DML (e.g. SQL) so that data can be fetch in the root data base of the view (e.g a relational one). The idea can be generalized to allow the specification of federated heterogeneous databases through a unifying object data model where tasks such as oid management and query transformation and processing are carried out by the view mechanism.

6 Conclusion and Future Work

This paper presented a declarative specification language and a general framework for mapping relational data into objects. We have already validated the architecture by a prototype implementation based on an object view mechanism.

The prototype has some limitations: objects in the view are not materialized, they can be manipulated and displayed, but they cannot be extracted from nor loaded into a new object database. Also, we still rely on SQL for view specification, although mappings can be defined based entirely on OQL. Some compiler optimizations could derive the SQL query from the OQL queries embedded in the mappings.

In spite of its limitations, the current prototype implements a simple and efficient solution to the problem of accessing legacy relational data through an ODBMS. The next version (under implementation) of the prototype will be more independent of the view language and of the source model (logical independence) and fully ODMG compliant. In addition, we will fully check consistency between mapping declarations and the relational semantics.

A R&D effort is now going on within the context of the Syndamma project[3].

Acknowledgments: Karim Souak participated in the implementation of the current prototype on top of O_2Views. We would like to thank Serge Abiteboul, François Bancilhon, Sophie Cluet and Michel Scholl for their valuable comments on a preliminary version of this paper.

References

[1] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 238–247, 1991.

[2] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *Proc. ACM SIGMOD Conference on Management of Data*, 1989.

[3] R. Ahmed, P. DeSmedt, and W. Du et al. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, pages 19–26, December 91.

[4] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[3] Syndamma is an Esprit project funded by the European Community where the partners are Datamat (Italy), ITC (Ireland), Mari (Nothern Ireland), O_2Technology (France), Tecnopolis (Italy) and University of Ulster (U.K.)

[5] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4), December 1986.

[6] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. of Int'l Conference on Extending Data Base Technology*, pages 136–151, Vienna, March 1992.

[7] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[8] M.A. Casanova. Inclusion dependencies and their interaction with functional dependencies. In *First ACM SIGACT-SIGMOD Symposium on principles of Database Systems*, 1982.

[9] M. Castellanos. A methodology for semantically enriching interoperable databases. *Proc. of Int'l Conference on Extending Data Base Technology*, 1993.

[10] R. Catell. *The Object Database Standard: ODMG'93*. Morgan Kaufmann, 1994.

[11] R. Dale. Database migration: keeping a steady course. *Database Programming and Design*, 3(4), 1990.

[12] A. D'Atri and D. Saca. Equivalence and mapping of database schemes. In *Proc. of the tenth Int. conf. on VLDB*, Singapor, August 1984.

[13] U. Dayal. Queries and Views in an Object-Oriented Data Model. In D. Stemple R. Hull, R. Morrison, editor, *Database Programming Languages - The Second International Workshop*, pages 80–102. Morgan Kaufmann, 1990.

[14] D. Fishman et al. Iris: an object-oriented database management system. *ACM TOIS*, 5(1):46–69, 1987.

[15] J. Albert et al. Automatic importation of relational schemas in Pegasus. In *Proc. RIDE-IMS'93*, 1993.

[16] W. Gotthard, P.C. Lockermann, and A. Neufeld. System-Guided View Integration for Object-Oriented Databases. *IEEE Transaction on Knowledge and Data Engineering*, 4(1), Feb. 1991.

[17] M. Haertig and K. Dittrich. An object-oriented integation framework for building heterogeneous database systems. In *Proc. IFIP-DS5 Semantics of Interoperable Database Systems*, Lorne, Australia, 1992.

[18] R. Herzig and M. Gogolla. Transforming conceptual data models into an object model. In *Proc. of the int'l Conf. on Extending Database Technology*, pages 280–298, 1992.

[19] P. Johannesson. Schema Transformations as an Aid in View Integration. In *Proc. of the Int'l Conf. on Extending Database Technology*, 1993.

[20] L.A. Kalinichenko. Methods and tools for equivalent data model mapping construction. In *Proc. of the Int'l Conf. on Extending Database Technology*, pages 92–119, Venice, Italie, March 1990.

[21] M. Kaul, K. Drosten, and E.J. Neuhold. ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. *IEEE Data Engineering*, pages 2–10, July 1990.

[22] A. Keller, R. Jensen, and S. Agarwal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. In *ACM SIGMOD 93*, Washington DC, May 1993.

[23] W. Kim. On unifying relational and object-oriented database systems. *Lncs on Advanced Databases*, 1993.

[24] W. Klas, G. Fischer, and K. Aberer. Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept. *Systems Integration*, September 1993.

[25] I. Kobayashi. Losslessness and Semantic Correctness of Database Schema Transformation: Another Look of Schema Equivalence. *Information Systems*, 11(1):41–59, 1986.

[26] P. Lyngbaek and V.Vianu. Mapping a semantic data model to the relational model. In *Proc. ACM SIGMOD*, San Francisco,CA, 1986.

[27] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. of the 19th VLDB*, Dublin,Ireland, 1993.

[28] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. In *Proc. of the Int'l Conf. on Extending Database Technology*, Cambridge, England, 1994.

[29] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, July 1987.

[30] S. Navathe and C. Ceri. A vertical partitionning algorithm for database design. *ACM Transaction on Databases*, (4), 1984.

[31] E. Pichat and A. Roche. Extension de la normalisation aux dépendances d'inclusion. In *INFORSID*, Nancy, France, 1989.

[32] W. Premerlani and R. Blaha. An Approach for Reverse Engineering of Relational Databases. *Communications of ACM*, (5), May 1994.

[33] E. Radeke and M. Scholl. Framework for Object Migration in Federated Database Systems. In *Proc. Int'l Conf. on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[34] S. Hayneand S. Ram. Multi-User View Integration System (MUVIS): An Expert System for View Integration. *IEEE Data Engineering*, pages 402–409, July 1990.

[35] Rumbaugh, J. E. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[36] C. Santos. Design and Implementation of an Object-Oriented View Mechanism. In *10èmes Journées Bases de Données Avancées*, Clermont-Ferrand, France, September 1994.

[37] C. Santos, C. Delobel, and S. Abiteboul. Virtual Schemas and Bases. In *International Conference on Extending Data Base Technology*, Cambridge, March 1994.

[38] M. H. Scholl and H.-J. Schek. Supporting Views in Object-Oriented Databases. *Data Engineering Bulletin*, pages 43–47, June 1991.

[39] S. Sheth and J. Larson. Federated Database Systems for Managing Distributed Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.