Bridging the Gap between C++ and Relational Databases

Uwe Hohenstein

Corporate Research and Development, Siemens AG, ZFE T SE 4, D-81730 München (GERMANY)
E-mail: Uwe.Hohenstein@zfe.siemens.de

Abstract. This work presents a new approach to access existing relational databases from C++ programs in an easy and natural way. The coupling of both worlds makes use of data reverse engineering techniques. Semantics that is inherent to relational data is made explicit by using object-oriented concepts extensively. Relationships and subtypes are expressed directly in order to take great benefit of them. C++ application programs are thus given the ability to handle relational data as if they were C++ objects.
The key to our approach is a powerful specification language that allows for defining object-oriented views, i.e., describing how object types, relationships between them, and subtype hierarchies are derived from relational tables. Even complex relational situations can be remodelled in an intuitive and concise manner.
Given a concrete specification, a C++ database interface is generated preserving the object-oriented view for accessing relational data. Access methods are automatically implemented on top of the relational system.

1 Introduction

Nowadays, it is widely accepted that the object-oriented paradigm reduces the difficulty of developing and evolving complex software systems. Object-oriented programming languages encompass useful constructs such as inheritance and encapsulation that can be used to define complex objects and behavioural properties of objects. These pleasant characteristics make them more desirable for handling many kinds of new applications than conventional programming languages.

Applications written in object-oriented programming languages naturally want to store objects in a database and retrieve them. In fact, object-oriented DBSs (database systems) pick up this point and enhance object-oriented languages to support database capabilities like persistence, transactions, and queries in a homogeneous manner so that the programmer gets the illusion of just one language.

But on the other hand, enterprises are just advanced to gain confidence in relational DBSs since robustness and reliablity are gradually accepted. Storing data in relational databases, lots of applications have been developed on top of such systems recently. This data is a necessary input to many decision making

processes. New emerging applications will *still* need to access this relational data. Consequently, many companies will *not* replace their legacy system with object-oriented ones for the forseeable future [IEEE95, PeH95].

In fact, there is no principle problem to make relational data accessible from object-oriented programming languages. Database applications can be written using embedded SQL statements. But this approach suffers from the need to manage two languages with absolutely different paradigms and to interface them with extra programming effort ("*impedance mismatch*"). Furthermore, the "*semantic gap*" is coming to light: The application maintains complexly structured objects, while the relational DBS provides simple tuples. Retrieved tuples must be converted to objects, and objects must be broken down to tuples. The handling is cumbersome and makes application programs difficult to write and hard to read.

In this paper, we accomodate ourselves to the significance of legacy data existing in relational DBSs and the programming language C++ [Str91]. The main contribution consists of proposing a flexible and homogeneous coupling of both worlds, solving the problems of impedance mismatch and semantic gap in an elegant way. The impedance mismatch is avoided by staying completely in C++. Database features are encapsulated in predefined C++ classes and methods, thus hiding the specific coupling mechanisms of relational systems.

We bridge the gap between C++ and relational databases by translating the relational definitions of data to equivalent object-oriented class definitions. Principally, tables can be represented by C++ classes that get the same attributes. In spite of being able to conceal the cursor concept by means of methods, the application still handles tuples instead of objects. Tuples can be manipulated in a C++ way, but tuples are isolated, as there are no relationships. Our solution to the semantic gap is *semantic enrichment*. The semantics of tables, being hidden in foreign keys etc., is made explicit. Relationships, subtypes, and embedded structures are expressed explicitly in object-oriented terms. By using the C++ type system extensively, applications are able to benefit directly from the support for inheritance and polymorphism already available in C++.

In sum, C++ application developers see an object-oriented representation of relational data. Passing on the modelling power of C++ to the operational level retains the higher degree of abstraction. Manipulating and accessing data is completely done on a more abstract level in terms of object-oriented concepts, handling objects and relationships. In addition to features for navigating through the database, powerful associative queries are supported in an object-oriented way. Software development productivity is increased by eliminating the need for programmers to code the mapping between the data structures of the programming language and the database.

There are some commercial C++ class libraries such as RogueWave's DBtools that attempt to ease the access of relational databases for C++ applications. They only encapsulate database functionality and essentially hide the embedding of SQL in a programming language. The handling of relational databases gets a *C++-like appearing*, but the real concepts of object-orientation like inheritance

are not applicable. Other work such as [HoO93] addressed some but not all of the SQL/C++ issues. They proceed in a *bottom-up* manner and store C++ classes in relational databases by breaking down objects into tuples. To use existing relational databases, this implicit mapping must be inverted in order to find a schema that maps onto the existing tables. Some other tools such as Persistence and UniSQL [Kim92] behave similarly. Closer to our work comes the approach of O-R-Gateway [AlT92]. Generating a C++ view of relational data *automatically*, their approach suffers from not treating all relational situations correctly. The interface provides only a rudimentary object-oriented view. Other proposals like [ABV92] make things easier as they do not rely on C++ and existing databases, but design an *object-oriented database programming language* from scratch.

In the following, we present our approach. Section 2 is concerned with the database interface for C++. The interface we provide is that defined by ODMG-93 [Cat94], the future *standard* for object-oriented DBSs. ODMG-93 proposes an object-oriented data manipulation facility that corresponds to the C++ type system and provides a C++ conforming way to handle data. Most vendors of object-oriented DBSs are committed to support this standard soon.

Afterwards, we present the basis for semantic enrichment, a logic-based specification language (Section 3). This language allows for remodelling tables in the ODMG-93 object model. Object-oriented views are specified in an intuitive and easy to understand way. The semantics of tables is made explicit, it is "re-engineered" in object-oriented terms in the sense of data reverse engineering [HTJC93, CACM94, CBS94, PrB94, PeH95].

The specification of semantic enrichment must be done manually, but the C++ database interface is provided automatically due to a *generative* approach. Given a specification, a generator produces the C++ interface. This interface implements an ODMG-compliant access to the relational database. Section 4 presents the overall architecture of the generative approach and elucidates the most important parts of the implementation.

The work we present here is part of a project called "**F**lexible **I**ntegration of **H**eterogeneous **D**atabase Systems" (FIHD) which is concerned with database interoperability. Section 5 outlines some further aspects of FIHD. The overall goal is to provide applications with one single ODMG-compliant interface to operate on several database systems. The generative approach builds the first step to incorporate relational systems in such an interoperability approach.

2 The ODMG-93 Database Standard

2.1 Object Model and Object Definition Language

The database standard ODMG-93 is principally independent of programming languages. An object model provides concepts to define objects in a neutral form. We briefly summarize the essential terms and concepts used throughout the paper. Figure 1 presents a simple example modelling a company database.

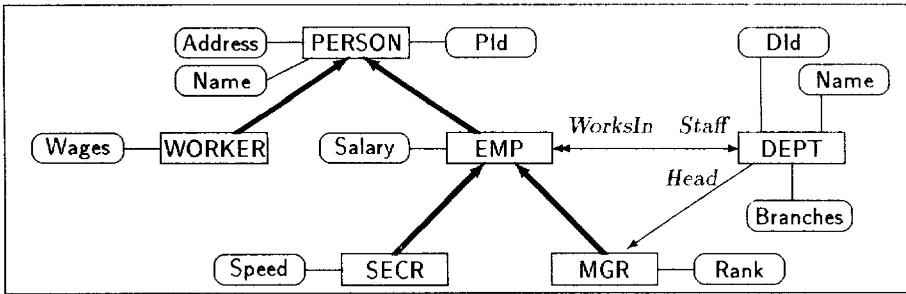


Fig. 1. Sample ODMG Schema

There are *object types* (in the sense of C++ classes) like PERSON and DEPT (department) that possess *attributes*. Every person (of type PERSON) has a number (PId), a Name, and an Address. Attributes are associated with a domain, which may be primitive like Long and Float or predefined like Date, Time and String. Object types are also valid domains. For instance, attribute Address may be of domain ADDRESS, which is an object type structured as ZIP code, City, Street, and Houseno. However, this does not represent a relationship between PERSON and ADDRESS; a person's address is embedded in PERSON, it is not an object and cannot be referenced from other objects. Furthermore, there are predefined templates for Set's, Bag's, List's, and Varray's which can be applied to domains, e.g., Set<String>. Bags are multisets, which retain duplicates. Lists possess an order so that the i-th element can directly be accessed by its position number.

ODMG-93 provides an explicit *relationship* concept. WorksIn is a *single-valued* relationship, it points to exactly one object of type DEPT. In contrast, Staff is *multi-valued* (denoted as a double-headed arrow), referring to a collection (set or list) of objects. WorksIn and Staff specify different directions of the same semantic relationship. Such a relationship is called *bidirectional*. Referential integrity is automatically guaranteed, there is no danger of 'dangling pointers'. Furthermore, both directions are kept consistent in contrast to simple pointers. If a relationship between two objects is newly established, then it is visible from both sides. Relationships can be *unidirectional*, too: The relationship Head is directed, from one object type to another. The direction is important for access. Referential integrity is not controlled and must be maintained manually.

Object types can be organized in *subtype hierarchies*. Object type PERSON has two subtypes, (blue-collar) WORKER and EMPloyee, and EMP in turn has subtypes SECRetary and MGR (manager), graphically indicated by broad arrows. As in C++, each subtype inherits all the properties of its supertype(s), attributes as well as methods and relationships. Multiple inheritance is possible and especially useful to express non-disjoint subtypes.

As usual, methods can be defined and attached to object types in order to define the behaviour of objects.

An Object Definition Language (ODL) reflects the concepts of the object model and allows for the specification of types and relationships among them in a syntactic form. The ODL is still independent of language-specific concerns.

2.2 Object Manipulation and Querying in the C++ Binding

ODL specifications can be transformed into C++ and Smalltalk, obtaining equivalent class representations. Corresponding *C++* and *Smalltalk binding* define their appearings. In case of C++, every object type is transformed into a C++ class. Attributes and relationships are mapped to data members of corresponding C++ domains or predefined classes. Indeed, relationships result in pointer-like shapes. All those C++ classes are used by programs to invoke database functionality. They do not reflect only the structure of object types, but also provide generic methods for manipulating data in C++ by means of an Object Manipulation Language (OML). Each object type possesses predefined methods like a new operator to create new objects, a delete_object method to delete an object, methods for navigating along relationships, for transaction management, querying, etc. These C++ methods enable application programs to access databases. The following piece of code presents a short C++ application program.

```
Database db;    db.open("myDB");
Transaction t;  t.start();
Ref<EMP>        e = new(db) EMP  (3, "Lucky Luke", 3000);
Ref<DEPT>       d = new(db) DEPT (10, "Cowboys");
e->WorksIn = d;   cout << e->WorksIn->Name << endl;
...
d->Staff.delete_element(e);
Set<Ref<EMP>> empSet = d->Staff;
t.commit();
db.close();
```

Database and Transaction are predefined classes that manage database and transaction handling. After opening a database "myDB", a transaction is started. The basis for handling objects are so-called *references* given by a Ref template. They behave like C++ pointers in a certain sense, however, they are able to refer to transient *and* persistent objects. Particularly, attribute access and method invocation is done via '->'. Here, an object of type EMP with a PId 3, a name "Lucky Luke" and a salary of 3000 is created by applying operator new (provided a corresponding constructor exists for EMP). The operator new is overloaded as it requires a parameter db, the database the object is to be stored in. Similarly, a department "Cowboys" is created. The employee e is hired by this department: d is assigned to the WorksIn relationship. Since the WorksIn/Staff relationship is bidirectional, the employee is implicitly inserted into the staff of department d. This can explicitly be done by d->Staff.insert_element(e), too. The employee is fired by d->Staff.delete_element(e). Additional methods are available to process the staff as a set of employees (Set<REF<EMP>>), e.g., iterators can be created to handle a collection element by element. All modifications to data are temporary, until a commit is made to the transaction. Any changes to objects and relationships are then stored persistently in the database.

All these methods are predefined for performing database operations. It is important to note that *user-defined* methods can be defined in C++ and attached to classes in order to provide object behaviour. These methods can of course invoke database functionality by using those predefined functions.

A special method oql(result, predicate) allows invoking associative queries. The parameter predicate of type char* contains a string that defines the query specified in an Object Query Language (OQL); result obtains the query result. For example, the query "select d.Name from d in Depts where d.Head.Name = 'Lucky Luke' " computes the names of those departments d (in the extent Depts) that are headed by 'Lucky Luke'. OQL is an object-oriented extension of SQL designed to work on the constructs of the object model. It enhances SQL in an orthogonal manner with object-oriented features like inheritance and traversal along relationships. Due to space limitations, the reader is referred to [Cat94] for further details about the OQL.

3 Specification Language

In order to bridge the semantic gap between C++ and relational databases, the basic principle of our approach consists of remodelling relational schemas in the ODMG model in a semantic enrichment process [CaS91, MaM90, NNJ93, HoK95]. Applications are given "real" object-oriented views of the relational data including relationships and subtype hierarchies. It is just now that applications reap the full benefits of object-orientation, as they are no longer responsible for managing relationships and inheritance by their own.

It is very important that semantic enrichment is obliged to make explicit the correct and precise semantics because object-oriented operations will get a wrong semantics otherwise. There is the necessity of expressing any kind of semantics in relational data. Hence, our approach stresses *expressiveness*. The price we pay for comprehensive remodelling capabilities is a *manual* specification of enrichment. We consider this matter not so bad due to the following reasons:

- The information in demand is often available in form of (object-oriented) design documents, which provide a good basis for semantic enrichment.
- There has been a flurry of activities in the field of data reverse engineering to propose algorithms, methodologies, and heuristics [HTJC93, CBS94, PrB94]. This work as well as knowledge acquisition approaches [CaS91, MaM90], which analyze the contents of databases in order to detect semantics, do a valuable job. Hence, our approach is complementary and can benefit from this work already done.
- Automatic types of reverse engineering and knowledge detection do not always produce satisfactory results. For example, earlier approaches simply do not attempt to rebuild subtypes, or are only able to rebuild subtypes created by just one strategy (e.g., [CaS91, AlT92, YaL92]). Multi-level subtype hierarchies are rarely managed properly.

We propose an approach that is capable of remodelling any relational situations in object-oriented terms by extensively using all the concepts of the ODMG-93 object model. In particular, the general case of of multi-level hierarchies can be handled. A powerful *specification language* is used to this end, taking into account several enrichment concepts in an orthogonal manner. This language allows one to precisely describe how tables in the relational database schema can be combined to object types. Nevertheless, we do not want to over-

shoot the mark. We do put emphasis on powerful mechanisms to derive object types form tables in various ways. But we have to avoid problems with *view updates*. This is important because we automatically generate object-oriented operations (see later) the effect of which must be unambiguous when operating on tables. Hence, no schematic discrepancies [SCG92], which restructure table and attribute names to attribute *values*, are expressible. Such aspects are a matter of taste how to see data, and consequently less necessary to express real semantics.

The syntax of the specification language bridles the horse from the back. It is specified what object types are the outcome and how they correspond to tables. This is advantageous because an object type is generally made up of several tables. The syntax remains intuitive and easily understandable, and the object types are immediately visible. The language adopts the ODL of ODMG-93 and introduces some amendments in order to express connections between object-oriented and relational schemas.

We are now discussing the specification language in more detail. The discussion is based on some relational representations of the schema in Figure 1. The examples will give a feeling about the underlying principles, i.e., how to cluster several tables into one object type, how to rebuild relationships, and how to regain complete subtype hierarchies.

3.1 Deriving Object Types and Relationships from Tables

We consider the relational schema given in Figure 2. Table M represents managers (MGR), while table D contains departments (DEPT). The branches of departments are stored in table B (DId, Loc, No) the tuples of which contain the branches for each department DId value by value; each branch receives a number enumerating the branches of a department. The headquarter is located in AA (No=1), BB is the second place, and so on. The Mgr column in table D is a foreign key, it refers to the manager in M who is the *Head* of that department.

The enrichment specification in Figure 2 combines the tables D and B to one object type DEPT with a multi-valued attribute Branches. Object types are defined as interface declarations as in ODL. The extent clause defines a variable to access the objects of a type: Depts is necessary to constitute an entry point in DEPT for querying data; only then can objects of a type be queried in an associative manner. The key-clause contains (object-oriented) key attributes that care for uniqueness. For example, the DId-values of departments are requested to be unique. The part in curly brackets specifies attributes and relationships. This is the usual way to define object types in the ODL.

Those interface declarations form the basis for logic-based extensions that express connections between object-oriented and relational schemas. The clause from relation relates the specified object type to a table. It specifies in what table the objects of a type are found. DEPT from relation D[DId] means that type DEPT is directly built from table D. DId is the relational key of D. Each tuple, which is uniquely identified by its key value, refers to one object. We presuppose a key for each table, because it is necessary to constitute object identifiers in the runtime system (see later on). Composite keys are possible and denoted as $(a_1, a_2, a_3, \ldots)$.

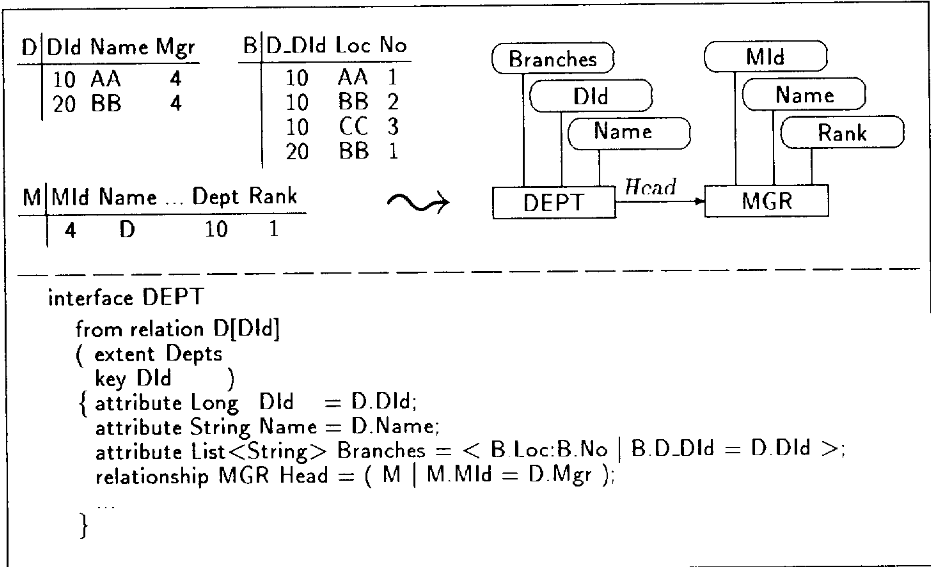


Fig. 2. Sample Enrichment

Equations '=' occurring behind the attributes relate object type attributes to relational attributes. The simplest form is Long DId = D.DId and directly connects an object type attribute DId with domain Long to a relational attribute DId. Attributes are renamed by specifying different names on the left hand side of the equations. Renaming is useful to choose intuitive and self-explanatory names.

The list-valued attribute Branches is made explicit in DEPT by using a list constructor <...> in Branches = < B.Loc:B.No | B.D_DId = D.DId >: Compute the Loc values for each tuple in B that possesses a D_DId equal to the DId of the department. In this case, the attribute No is used to determine the order in the list. Similarly, a set constructor could have been applied to build a *set* of branches: Branches = { B.Loc | ... }. In the same way, the effect of normalization can be inverted. It is possible to 'cluster' any tables related by attributes to one object type [YaL92].

Similarly, relationships can be expressed. The Head of a department (represented by a foreign key Mgr) is made explicit in a specification by relationship MGR Head = (M | M.MId = D.Mgr): The head consists of that tuple in M (identified by the respective key) that possesses an MId equal to the Mgr-value of the department. Round brackets convert a tuple in M into a corresponding object of type MGR.

If the relationship Head were represented by a table H(DId,MId), containing the Id's of the participating tables, then the specification would look like relationship MGR Head = (M | M.MId = H.MId, H.DId = D.DId) .

In both cases, composite attributes for expressing the relationships can also be handled and specified.

3.2 Subtype Hierarchies

Remodelling subtype hierarchies requires advanced concepts, since several relational representations exist. Some approaches to semantic enrichment [MaM90, CaS91, YaL92] are able to make subtype relationships explicit. However, they generally take into account only one strategy (the vertical one below) and lack the handling of multi-level subtype hierarchies. But it is just the "implementation" of subtypes offers a wide spectrum of possibilities. It is important to detect them correctly, even if they have been applied in a mixed manner.

We refer to the subtype hierarchy given in Figure 1, however, we disregard relationships for a moment. Let us assume in the following that PId 1 is a real person, 2 an employee, 3 a secretary, 4 a manager, and 5 and 6 are workers.

Vertical Partitioning. Possibly the most common way of representing subtype hierarchies is a *vertical* partitioning. Let us organize the tables P, E, M, S, and W in Figure 3 into a subtype hierarchy. Each table refers to one object type in the hierarchy, as usual, and contains all the *elements* of the corresponding type, i.e., the instances of the type and all its subtypes. Only the specific attributes of the type are found in the table; attributes inherited from supertypes are available in the tables associated with that supertype. The following inclusions then hold between supertype and subtype tables: $P.Id \supseteq W.WId$, $P.Id \supseteq E.EId$, $E.EId \supseteq S.SId$, and $E.EId \supseteq M.MId$. To access the attributes of supertypes, tables must be joined over the Id attributes: Take the MId-value of a manager and look in E and P for tuples that have the same value as EId and Id, resp., to get the properties inherited from PERSON and EMP. Please note that those Id attributes do not need to be keys, but they must guarantee uniqueness of attribute values.

P

Id	Name	Addr
1	A	A_town
2	B	B_city
3	C	C_village
4	D	D_village
5	E	E_city
6	F	F_town

E

EId	Salary
2	2000
3	3000
4	4000

S

SId	Speed
3	133

M

MId	Rank
4	1

W

WId	Wages
5	555
6	666

```
interface PERSON from relation P[Id]
   { attribute PId = P.Id;
     attribute Name = P.Name;
     ... }
interface EMP    : PERSON from relation E [EId =P.Id]  ...
interface WORKER : PERSON from relation W[WId=P.Id]    ...
interface SECR   : EMP    from relation S [SId =E.EId] ...
interface MGR    : EMP    from relation M[MId=E.EId]   ...
```

Fig. 3. Vertical Partitioning of Subtypes

The enrichment specification in Figure 3 should be understood as follows.

The elements of PERSON are found in table P, as usual (from relation), elements of subtype EMP in E, and so on. Since each table contains only the specific attributes of that object type, the connection to the supertype table must be established. This is done by EMP : PERSON from relation E[EId = P.Id]: Tables E and P are related by attributes EId (of relation E) and Id (of P): A person is an employee if its Id occurs in E as EId. Composite attributes are again possible.

Horizontal Partitioning. Horizontal partitioning of types into tables is another subtype representation. As shown in Figure 4, one table again holds the information of one object type in the hierarchy. The structure of each table comprises the specific information of its corresponding type and, unlike vertical partitioning, the attributes of supertypes, too. Hence, the attributes inherited from supertypes are directly available in each table. But each table contains only the *instances* of the type itself. Hence, a manager is stored in M only, however, including the EMP and PERSON information. On instance level, exclusion conditions are fulfilled between super-and subtype tables: P.Id $\cap$ E.EId $= \emptyset$, P.Id $\cap$ W.WId $= \emptyset$, E.EId $\cap$ S.SId $= \emptyset$, and E.EId $\cap$ M.MId $= \emptyset$.

P	Id	Name	Addr
	1	A	A_town

E	EId	Name	Addr	Salary
	2	B	B_city	2000

W	WId	Name	Addr	Wages
	5	E	E_city	555
	6	F	F_town	666

S	SId	Name	Addr	Salary	Speed
	3	C	C_village	3000	133

M	MId	Name	Addr	Salary	Rank
	4	D	D_village	4000	1

```
interface PERSON from relation P[Id] + W[WId] + E[EId] + S[SId] + M[MId]
  { attribute Long PId = P.Id + W.WId + E.EId + S.SId + M.MId;
    attribute String Name = P.Name + W.Name + E.Name + S.Name + M.Name; ... }
interface EMP : PERSON from relation E[EId] + S[SId] + M[MId]
  { attribute Float Salary = E.Salary + S.Salary + M.Salary; }
interface WORKER : PERSON from relation W[WId]
  { attribute Float Wages = W.Wages; }
interface MGR : EMP from relation M[MId]       interface SECR : EMP from relation S[SId]
  { attribute Long Rank = M.Rank; }              { attribute Long Speed = S.Speed; }
```

Fig. 4. Horizontal Partitioning of Subtypes

Figure 4 demonstrates how to rebuild the subtype hierarchy. Each from relation clause defines how to compute all the elements of a type. The elements of PERSON, spread over the tables P, W, E, S and M, are obtained by computing the union ('+') of tuples. As objects are made of tuples of different tables, the key values of those tuples must be used to build object identifiers. The parts [...] indicate the corresponding key attributes. Attribute correspondences are specified in the same way: Id = P.Id + W.WId + E.EId + S.SId + M.MId identifies semantically equivalent attributes in tables; the Ids of persons are stored in all of those tables, however, in differently named columns!

Flag Approach. In contrast to the first strategies, one single table can represent the whole hierarchy as well. One table P contains all the information about all the object types. Flags like Emp?, etc. determine the specific subtype. Flags can denote elements or instances. They represent elements in Figure 5: Tuples having Emp?=true correspond to elements of EMP. Naturally, only sensible flag combinations must occur. For example, Emp?=false and Secr?=true is not valid, a secretary must also be an employee. Furthermore, non-applicable attributes must be NULL so that Emp?=false implies Salary=NULL.

P	Id	Name	Addr	Emp?	Salary	Mgr?	Rank	Secr?	Speed	Worker?	Wages
	1	A	A_town	false	NULL	false	NULL	false	NULL	false	NULL
	2	B	B_city	true	2000	false	NULL	false	NULL	false	NULL
	3	C	C_village	true	3000	false	NULL	true	133	false	NULL
	4	D	D_village	true	4000	true	1	false	NULL	false	NULL
	5	E	E_city	false	NULL	false	NULL	false	NULL	true	555
	6	F	F_town	false	NULL	false	NULL	false	NULL	true	666

```
interface PERSON from relation P[Id] ...

interface EMP : PERSON from relation P[Emp? = true]
    { attribute Float Salary = P.Salary; }

interface WORKER : PERSON from relation P[Worker? = true] ...

interface SECR : EMP from relation P[Secr? = true] ...

interface MGR  : EMP from relation P[Mgr? = true] ...
```

Fig. 5. Flag Approach

If flags denote instances instead, then Emp?=true holds for real instances of EMP only. SECR instances still have Secr?=true, but now Emp?=false. Then, at most one flag can be true for each tuple in P.

Some variants of the flag approach are conceivable. In place of flags, an enumeration type type of domain { Emp, Secr, Mgr, Worker } can serve the same purpose. For example, the instances of EMP get a type-value Emp. The subtype specification can also be done by condition, e.g., Salary!=NULL could detect EMP instances. Even more general expressions could be used. However, there will be no possibility to distinguish between "value is unknown" and "value is inapplicable (no subtype attribute)", because both are represented by NULL.

Flag approaches can again be handled in the from relation clause. As usual, the part in [...] defines how to compute elements. The specification given in Figure 5 defines that any tuple in table P with Emp? = true refers to an object of type EMP. These objects are identified by PId, as arranged in PERSON.

Other flag approaches are handled by different forms of conditions. For example, if flags denote instances, the conditions look like

```
interface EMP : PERSON from relation P[Emp? = true or Secr? = true or Mgr? = true]
```

Those tuples of P are elements of EMP which have one of the (exclusive) flags Emp?, Secr? or Mgr? true. Naturally, this discriminant form allows for arbitrary conditions like [Salary!=NULL] and [type=Emp], too.

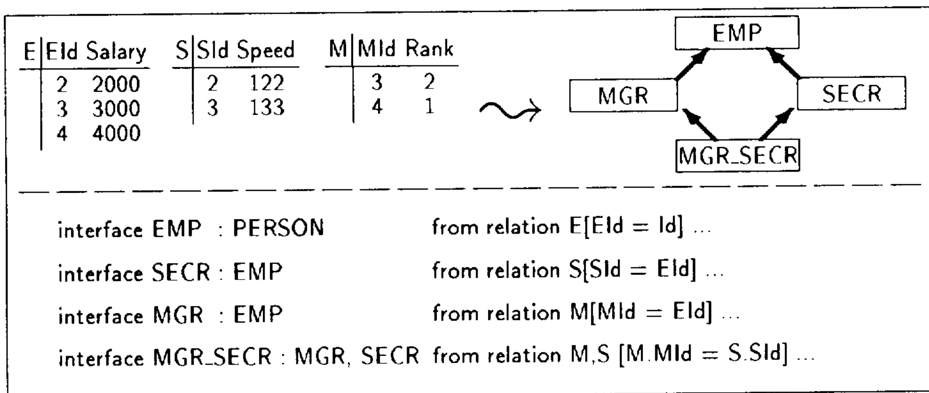
Complete Materialization. Another relational representation uses the schemas of horizontal, but the instances of vertical partitioning. For example, type EMP is represented by a table E(EId, Name, Addr, Salary) and contains three tuples with EIds 2, 3 and 4. This leads to redundancy, as the table P(Id, Name, Addr) contains the Name and Addr information for all tuples. Consequently, the name 'B' of Id 2 is stored in P and E. Since each table contains the whole information of a type, all the elements and all the attributes, we call it *complete materialization.* Rebuilding the hierarchy from these tables is done in the following way:

```
interface EMP : PERSON from relation E[EId =P.Id]
    attribute Float Salary = E.Salary = S.Salary = M.Salary;
```

The form of from relation is similar to vertical partitioning, as each table contains elements. Hence, the correlation to supertype tables is expressed by EId=P.Id. But in contrast to vertical partitioning, redundancies must be reflected for the attributes: Employees' Salaries are stored in E, M and S.

3.3 Multiple Inheritance

Subtypes are disjoint w.r.t. instances in C++ and the ODMG object model. On the other hand, this is not true for tables, since they can contain tuples with the same Id. For instance in Figure 6, an employee with EId 3 occurs in S and E, (s)he is a secretary and a manager at the same time.



E	EId	Salary
	2	2000
	3	3000
	4	4000

S	SId	Speed
	2	122
	3	133

M	MId	Rank
	3	2
	4	1

Fig. 6. Non-disjoint Subtype Tables

Non-disjoint types can be modelled by means of multiple inheritance. An artificial subtype MGR_SECR represents the intersection of MGR and SECR. MGR_SECR is necessary to be able to insert objects like 3 that are both manager and secretary. The attributes inherited from EMP via MGR and SECR are virtual and occur only once in MGR_SECR. The type EMP enables accessing all the elements, no matter whether managers, secretaries, or both. Please be conscious of telling apart elements and instances: 2 and 3 are elements of SECR, however, 2 is the only instance, as 3 has become an instance of SECR_MGR.

Multiple inheritance is denoted as in C++, specifying several supertypes behind a colon: MGR_SECR : MGR, SECR. According to the semantics of from relation, elements of MGR_SECR are characterized: The set of elements is computed by intersecting tables M and S by a condition M.MId = S.SId.

3.4 Additional Concepts

Additional forms are available to handle further aspects which are important for building object-oriented views of tables. We briefly summarize them.

Multivalued attributes like Branches are sometimes available as a constant number of relational attributes, if the collection has a fixed size or an upper bound. This is particularly useful for small collections. Hence table D may look like D (DId, Name, Branch1, Branch2, Branch3), if there will be at most 3 branches in a department. In order to handle this, constant sets can be built over columns: Set<String> Branches = { Branch1, Branch2, Branch3 } .

Several relational attributes may correspond to a predefined ODMG data type such as Time and Date. For example, three Long-valued attributes Day, Month, and Year, assume that they occurred in table D, could be combined to form a date of foundation. A corresponding attribute equation then makes use of a tuple constructor (...) and looks like Date Foundation = (D.Day, D.Month, D.Year) .

Sometimes it is useful to structure several relational attributes in a similar manner, even if no predefined domains are applicable. Relational attributes ZIP, City, Street, and Houseno obviously represent addresses. It is useful to define an embedded structure Address that contains these components. The address of a person could be made explicit by defining an embedded type address that is used as domain for Address. A corresponding equation is similar to above.

New object types may be introduced, e.g., to concentrate common attributes into a *generalized* object type. Suppose tables A (AId, a, c, d) and B (BId, b, c, d) are given. Both tables can be generalized to a newly defined supertype C that receives c and d. Supertype C does typically not contain instances of its own. Indeed, creating instances, it is not clear in which table to put them.

Owing to *optimization* reasons, tables are often merged after design, in order to avoid costly join operations. Combined with previously discussed concepts, the specification language allows for splitting up tables into several object types.

One important point has been neglected so far. Relational DBSs possess modelling constructs such as not null that are provided neither in C++ nor in ODL. In order to reflect the relational semantics entirely, we introduced corresponding restrictions. Keywords like not null can be specified for attributes (in the relational sense), and relationships can be defined as mandatory: Any object must participate in a relationship of that type.

3.5 Complex Example

In order to demonstrate the power of the specification language, we are now presenting a complex relational schema that comprehends several of the concepts discussed previously in combination. Particularly, we use a relational representation of Figure 1 that incorporates different subtype strategies within one hierarchy. Sometimes, it is quite useful to having applied different strategies. Reasons

for that might be to speed up access for specific applications, which have different preferences for each level of the hierarchy. We apply a vertical strategy to PERSON–EMP and EMP–SECR, a horizontal one to EMP–MGR, and a flag approach to PERSON–WORKER. We obtain the tables given in Figure 7.

P

Id	Name	Addr	Worker?	Wages
1	A	A_town	false	NULL
2	B	B_city	false	NULL
3	C	C_village	false	NULL
5	E	E_city	true	555
6	F	F_town	true	666

E

EId	Salary	Dept
2	2000	10
3	3000	20

S

SId	Speed
3	133

B

D_DId	Loc
10	AA
10	BB
10	CC
20	BB

D

DId	Name	Mgr
10	AA	4
20	BB	4

M

MId	Name	Addr	Salary	Dept	Rank
4	D	D_village	4000	10	1

```
interface PERSON    // no supertype
  from relation P[Id] + M[MId]
  ( extent Persons
    key PersId      )
  { attribute Long PersId = P.Id+M.MId;
    attribute String Name = P.Name+M.Name;
    attribute String Address = P.Addr+M.Addr;
  }

interface EMP : PERSON
  from relation E[EId=P.Id][EId]+M[MId];
  ( extent Emps )
  { attribute Float Salary = E.Salary+M.Salary;
    relationship DEPT WorksIn inverse Staff
        = (D | D.DId = E.Dept+M.Dept); }

interface MGR : EMP
  from relation M[MId]
  { attribute Long Rank = M.Rank;
  }

interface DEPT    // no supertype
  from relation D[DId]
  ( extent Depts
    key DId       )
  { attribute Long DId = D.DId;
    attribute String Name = D.Name;
    attribute String Address = D.Addr;
    attribute Set<String> Branches =
        { B.Loc | B.D_DId = D.DId }
    relationship MGR Head =
        (M | MId = D.DId);
    relationship Set<EMP> Staff
        inverse WorksIn =
        { E+M | E.Dept+M.Dept=D.DId };
  }

interface SECR : EMP         // vertical
from relation S[SId=E.EId]
  { attribute Long Speed = S.Speed; }

interface WORKER : PERSON   // flag
  from relation P[Worker?=true]
  { attribute Float Wages = P.Wages; }
```

Fig. 7. Complex Specification of Semantic Enrichment

The tables reflect the characteristic inclusions E.EId $\subseteq$ P.Id and S.SId $\subseteq$ E.EId of vertical partitioning. According to horizontal partitioning, an exclusion condition E.EId $\cap$M.MId = $\emptyset$ holds. M contains additional employees who possess different Ids and have the complete PERSON and EMP attributes. Workers, finally, do not have a table of their own, but are part of P with a discriminant flag Worker?.

The WorksIn and Head relationships are represented by foreign keys Dept and Mgr, respectively. The Dept column in E refers to the department's DId, the employee works in, and similar for Mgr. Please note that horizontal subtypes receive all the properties of their supertypes. Hence, M also has an attribute Dept, since EMP's relationship to DEPT is valid for managers, too. Even the attributes of in-

direct supertypes are repeated. M obtains the attributes of P, although it is not a direct horizontal subtype of P. This is necessary, because M cannot "inherit" the attributes of PERSON otherwise by means of joins!

Figure 7 presents a specification that regains the schema given in Figure 1. PERSON from relation P[Id] + M[MId] specifies a horizontal strategy. The elements of PERSON are obtained by computing the union ('+') of tuples in P and M. Horizontal strategy is reflected in attribute equations: Name = P.Name + M.Name specifies that people's names occur in P and M.

The form EMP : PERSON from relation E[EId = P.Id][EId] + M[MId] represents a vertical strategy first of all: E[EId = P.Id] means that E is a vertical subtype of P. The tables E and P are related by attributes EId (of table E) and Id (of P). E[...][EId] + M[MId] specifies that EMP objects are stored in the tables E and M due to horizontal partitioning. Please note that different attributes could have been used for vertical and horizontal strategy!

WORKER : PERSON from relation P[Worker? = true] indicates a flag approach: The type WORKER is subtype of PERSON represented by a flag Worker? in table P.

Set<EMP> Staff = { E + M | E.Dept + M.Dept = D.DId } expresses a set-valued relationship to EMP for interface DEPT. Staff consists of those tuples in E and M (identified by respective keys) that have the department's DId as value of Dept. Keyword inverse marks a relationship as bidirectional, thus relating WorksIn of EMP to Staff of DEPT. The inverse relationship WorksIn is analogously computed by (D | D.DId = E.Dept + M.Dept). Head and Branches are specified as in Figure 2.

4 Generative Approach

4.1 Principle

The specification of object-oriented views is done manually by means of a specification language. Nevertheless, the database interface is produced automatically due to a generative principle: Given as input any specification of semantic enrichment, a generator produces a pile of C++ classes according to the language-specific ODMG C++ binding (cf. Subsection 2.2). The generated output provides a C++ database interface implemented on top of the relational system. Each interface declaration results in exactly one C++ class that defines generic methods for manipulation and navigation according to the ODMG standard. User-defined methods are added to these classes. C++ applications that want to access the relational database need only compile and link these classes into application programs. Figure 8 illustrates the process of generation. Software components are represented by boxes and require input data and produce output, both shown as parallelograms. Closed lines denote data flow, while broken lines define function calls.

Let us discuss the information flow between the basic components. Starting point is a *specification* defining semantic enrichment for a relational database.

A *Parser* first takes the name of a relational database and then reads the information about the database *schema*: Table and attribute names are found in the dictionary of the relational system. This information is stored in a meta

database and forms the relational part of *meta-information*. Parsing the enrichment specification, information about the derived object-oriented schema and its connection to relational tables is added to the meta information.

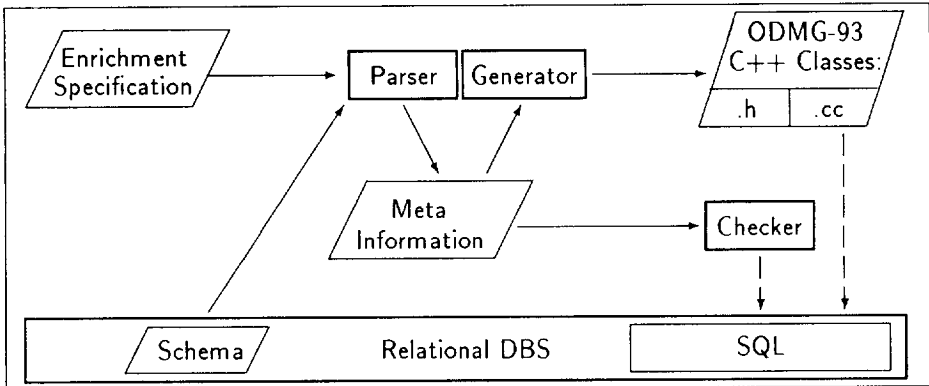


Fig. 8. Generative Approach

The implementation of the parser uses a comfortable compiler-compiler. The syntax of the specification language is defined in a yacc-like format, and semantic actions such as filling the meta database are implemented in C++. In addition to that, a lot of ***context-sensitive syntax rules*** are supervised. For example, all table and attribute names occuring in a specification must exist in the relational schema, and attributes must belong to the tables. The various constructs to remodel subtype hierarchies and relationships have additional demands.

A *Generator* takes the meta information as input and generates the ODMG-93 conforming classes. The output consists of C++ header files (".h"), which contain the C++ class definitions including the signatures of methods, and implementation files (".cc") implementing those methods. Naturally, the implementation of methods must call SQL in order to access the database.

As mentioned before, the meta database comprises the information about the relational schema, the outcoming object-oriented schema, and interrelations between them. Figure 9 gives a simplified view of the meta information. The left side consists of the relational part: R_TABLE contains the table names of a given schema; a table consists of several R_COLUMNs, and each R_COLUMN possesses a relational R_DOMAIN. One or more columns build the *key* of a table. Analogously, the right side contains the object-oriented counterpart, O_TYPEs with several O_ATTRibutes and associated O_DOMAINs. Relationships between object types are kept in O_RELSHIP; each relationship has a *source* type and a *destination*. Subtype hierarchies are reflected by *subtypes/supertypes* relationships. In the middle, information about semantic enrichment is placed. Each O_TYPE is related to several tables depending on the subtype strategy. In general, one table is the *base* table of an object type. Consequently, each O_TYPE refers to an ENRICHMENT object (via *enriches*) that determines the *base* table and its *key* columns. Subtype strategies are reflected by special subtypes of ENRICHMENT. For instance, HORIZONTAL keeps a list of pairs (R_TABLE, R_COLUMN) according

to the '+' (plus) form of from relation. Similarly, attribute and relationship equations are handled by ATTR_ENRICHMENT and R_ENRICHMENT. This simplified view illustrates that all the information about a specification is stored.

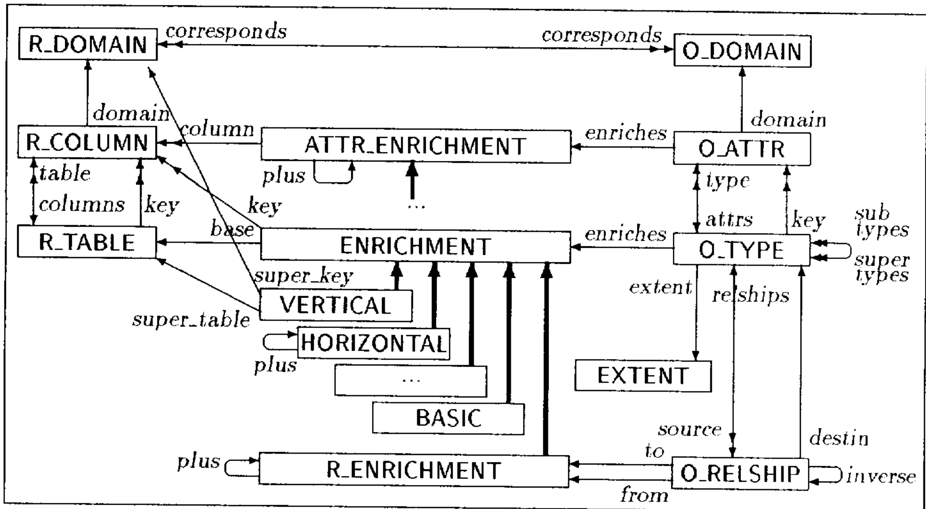


Fig. 9. Meta-Schema

Specifying semantic enrichment explicitly, obviously, not every enrichment specification makes sense. In principle, the user *supposes* that the database comprises the semantics specified, but the database does not know what semantics it has to satisfy. An *Integrity Checker* has the task to prevent users from specifying nonsense by checking the relational data against a specification. Subsection 4.3 discusses this point in more detail.

4.2 Architecture of the Runtime System

The output produced by the generator is an object-oriented runtime system that provides an ODMG-93 conforming access to relational data. C++ classes represent the relational data, however, in a less simple form than just tables. Moreover, methods define means for manipulation. These methods work on objects and are automatically implemented on relational SQL. In fact, the implementation is done in direct correspondence to the semantic enrichment. For example, referring to Figure 7, we consider the case of creating a new employee:

```
Ref<EMP> emp = new(db) EMP (3, "Lucky Luke", 3000);     (1)
emp->WorksIn = d;                                        (2)
```

The first C++ statement implies an SQL insert into tables E and P due to a vertical subtype strategy: Employees are stored in both tables. Assigning a department to the employee requires an update of the foreign key attribute Dept in E (Dept represents the WorksIn relationship).

Let us discuss the architecture of the generated runtime system. The implementation files (.cc) do not directly use the relational database system due to

portability, efficiency, and reduced amount of generated code. Hence, the runtime system is layered in order to bridge the gap between the ODMG interface and relational operations. Figure 10 gives a brief survey about the layering.

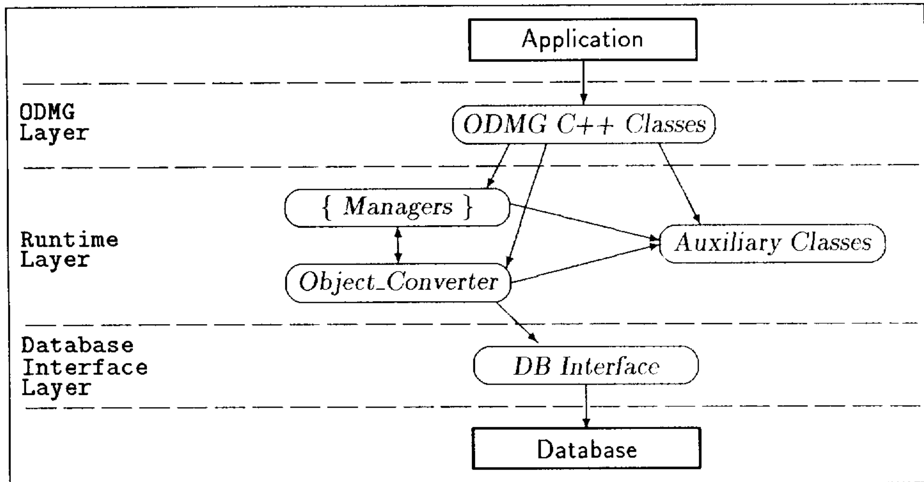


Fig. 10. Runtime System

The upper `ODMG Layer` consists of ODMG-93 conforming C++ classes that provides database access. Some classes like Transaction, Database and templates like Iterator<T>, Ref<T> and Set<T> are independent of the enrichment specification. Others indeed are dependent, in particular those classes that represent interface definitions.

The ODMG layer classes use functionality supplied by a `Runtime Layer`. This layer consists of several forms of *Managers*, a Transaction_Manager, Database_Manager, Extent_Manager, Object_Manager, and Query_Manager. The *Query_Manager* handles OQL queries, by translating them into relational SQL queries [Hoh95]. An *Object_Manager* manages all the objects at runtime in a cache. All modifications to objects are first made in the cache. When a commit occurs, all changes are made persistent in the database, i.e., objects are taken from the cache and put into the relational database. Figure 11 illustrates the principle of the Object_Manager.

Ref objects, the substitutes for pointers, refer to temporary identifiers, so-called *tids*, in the cache. This is advantageous because several references e1, e2 can point to the same object, e.g., if this object is fetched several times into different references. Tids avoid synchronizing modifications via different references, as just one physical instance of the object exists:

The internal structure of the Object_Manager can be understood as a collection of triples (tid, key, object pointer). Key and Tid are *Auxiliary Classes*. Key maintains the key values of any tuple. Keys are used to build object identifiers in the runtime system.

We discuss the connection between the ODMG Layer and the Object_Manager by listing the actions for creating a new object emp in OML (cf. (1) above). Ope-

rator new is overloaded as it now yields a reference instead of a pointer. The implementation of new creates the storage for an EMP object; attributes remain unset firstly. Then a tid entry is requested from the Object_Manager. An entry in the cache is made, relating tid and object pointer.

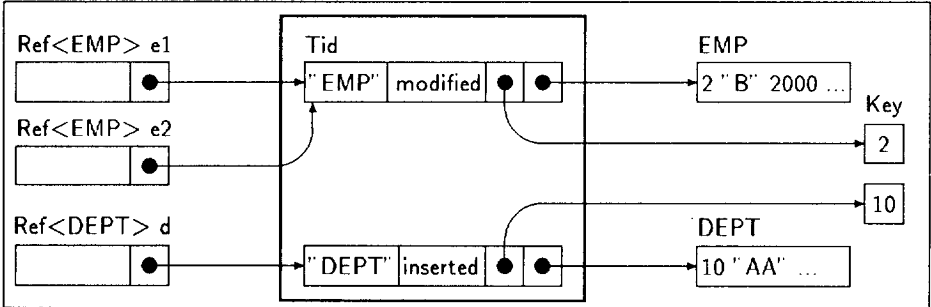


Fig. 11. Object_Manager

The implementation of the constructor EMP::EMP(Long k, ...), being implicitly invoked afterwards, reserves the key k in the relational database. If the key already exists, an error is returned. Furthermore, a Key instance is created from k. Tid and key are then associated in the cache. The Key instance serves as an object identifier: The pair (EMP, 3) uniquely identifies an object in the database, since 3 is unique in type EMP. The assignment emp = ... finally lets the reference of the newly created employee point to the entry in the cache.

The methods to allocate storage and to associate Key's, Tid's, and object pointers are part of the Object_Manager. Additional methods are available to look up objects by means of keys or tid's. The internal state of objects, i.e., whether they are deleted, inserted, or modified, is maintained and can be asked and set by corresponding methods. Any modifications are made in the cache, making topical the objects' states. Only if a commit is invoked at the ODMG layer, changes are made persistent. Indeed, the implementation of this method scans the cache and calls operations according to the objects' states, i.e., delete, insert, or update them. These basic operations are provided by the beneath Object_Converter.

The *Object_Converter* consists of several classes; each interface has a corresponding Object_Converter of its own. They support basic operations

- to load_objects into cache for a given key or tid,
- to load_extent (elements) and load_instance sets in order to materialize whole object types,
- to load_relationships of objects given by tid, and
- to store, modify, and delete objects.

The implementation of these methods is dependent on the specification of semantic enrichment as the effect varies from type to type due to specific mappings onto tables and columns. For example, inserting an object of type EMP

has an impact on table E and P; both contain information about employees. But storing a department is only done in D.

The Object_Converters incorporate several strategies to load objects from the database into the cache. Their goal is to find a good compromise between efficient access, main memory occupation, and necessity of information. In any case, if an object is demanded (calling load_object), all the information directly available in the corresponding base table is fetched into cache. Materializing an EMP object thus picks up all the information stored in E, i.e., EId, Salary, and Dept; the Dept-value is converted into a relationship WorksIn, but the related department is not always loaded. State information in the Object_Manager maintains what parts of an object have been fetched, what attributes, relationships, attributes of supertypes, and so on. Consequently, it is known what is available and what has to be fetched on demand. Sometimes, further information can be easily computed by joins in advance. For example, the department of an employee, which is referred by column Dept in E and M, can be materialized when fetching an employee by joining D, E, and M. Similarly attributes of supertypes can be pre-fetched by joins, e.g., a join between E and P to make available attributes inherited from PERSON. Nevertheless, too many joins in one SQL statement are rather inefficient. Global parameters like a maximal number of joins to be performed, maximal amount of storage for one object, total cache size, etc. can be tuned in order to control materialization. In a simple case, just a "lazy fetching", picking up elementary properties of an object, is possible.

The Object_Converters could in principle access the relational database directly. However, we put a `Database Interface Layer` in between due to portability; each relational DBS has its own call level interface. Exchanging the underlying relational DBS thus requires few modifications in only this layer.

4.3 Constraints Checking

Specifying semantic enrichment manually, there must be a monitor checking whether an enrichment specification makes sense, i.e., whether the database contains the semantics specified by a user. Comprising this semantics means for relational systems that certain conditions are satisfied by the relational data. Those integrity constraints are monitored by an integrity checker.

Given an enrichment specification, many constraints are derived by the generator automatically. For example, DEPT from relation D[DId] requires that DId is a candidate key for relation D: DId is obliged to uniquely identify tuples. The below SQL query Q_1 yields all tuples that violate this constraint:

```
select *
from  D
group by DId
having cnt(DId) > 1
```

<Query Q_1>

```
select * from S s
where not exists (select *
                  from  E e
                  where e.EId = s.SId )
```

<Query Q_2>

```
select *
from  E e , M m
where m.MId = e.EId
```

<Query Q_3>

Typical constraints claim for inclusion and exclusion conditions. Hence, as SECR is vertical subtype of EMP in Figure 7, the inclusion S.SId $\subseteq$ E.EId must hold; Q_2 detects violation. Horizontal strategy demands exclusive key values in

tables, e.g., $E.EId \cap M.MId = \emptyset$, respectively Q_3. Combining strategies produces constraints that are even more complex like $E.EId \subseteq \sigma_{Worker?=false}(P.Id)$.

These SQL queries are automatically generated and must yield an empty result; the constraints are violated otherwise, and the specified enrichment is not sensible. However, the queries supervise correctness only at the time of monitoring; constraints might not hold later on so that periodic checking is necessary. In fact, those queries are also useful in the reverse engineering process to indicate, e.g., subtype relationships. This is taken into account by the process described in [HoK95].

5 Conclusions

In this paper, we described an approach to interfacing existing relational databases from C++ programs. Handling the database is completely done in C++. In contrast to some commercial tools that encapsulate database access in special classes, we take a step further and eliminate the semantic gap between relational tuples and C++ objects. Our solution to this problem is *semantic enrichment* [MaM90, CaS91, HoK95]. Powerful object-oriented view of tables can be built by taking full benefit of object-oriented concepts. Hence, relational data is manipulated in an object-oriented manner, i.e., real objects are handled instead of tuples. Relationships can be traversed like pointers, and inheritance is applicable.

Our approach consists of specifying how tables are combined to object types by using a powerful *specification language*. This is advantageous because the approach is capable of remodelling complex relational situations as classes. Having defined a specification of semantic enrichment, a generator produces a corresponding object-oriented database interface. The methods, as well as their implementations on top of a relational system, are generated automatically. The overall result behaves like an object-oriented DBS. Nevertheless the goodies of relational systems, i.e., powerful query capabilities in the sense of SQL are still available, now in an object-oriented fashion. Deficiencies of querying, often recognized in object-oriented systems, are thus eliminated.

The generated interface complies with the future *standard* ODMG-93 [Cat94] for object-oriented DBSs. Hence, our approach facilitates replacing relational systems with object-oriented ones without affecting applications. Similarly, data can easily be migrated from relational to object-oriented systems, since migration programs can read objects from existing relational databases and then store them in an object-oriented DBS by handling just one interface. This is a first contribution to handle the hard problem of *legacy data* [IEEE95]. We are just extending our tool to ease the process of enrichment. [HoK95] presents a graphical approach to interactively design semantic enrichment.

The presented approach has been implemented on SUN workstations in AT&T C++ on top of the relational system INFORMIX. The implementation makes use of a compiler-compiler to produce the generator. The generative principle is designed to provide flexibility. Hence it is easily possible to exchange the underlying DBS.

Motivation for our work comes from a project concerned with database interoperability [HNS92]. A global interface, relying again on ODMG-93, should

provide database access to several kinds of database systems, relational, object-oriented ones, and others. Object-orientation provides a good basis for integrating heterogeneous systems [Har92, BNPS94]. The essential idea is to first translate schemas of component DBSs, expressed in the native data model of the system, into the ODMG-93 model. This leads to a *homogenization* of schemas. Thus, syntactic heterogeneity resulting from different data models and access interfaces of the component systems is eliminated. Our generator supports the homogenization of relational systems in an effective way, as it expresses implicit semantics directly in the ODMG object model and provides an ODMG database interface. Using ODMG-93 plays an important role: Soon or later, object-oriented DBSs will support this standard database interface. Then, no real homogenization will be necessary for them.

The main concern of the successive integration step is identifying conflicts between several homogenized schemas, to resolve them and to merge the schemas into global schema(s). Global schemas give a user the illusion of a homogeneous "database system", with a unified, database spanning, and transparent access to all the integrated data.

Future work will be directed to that integration step. We carry on applying our generative principle. Object classes that represent global schemas are automatically generated. They now provide an integrated database interface for all databases. An integration specification language is used to dissolve semantic heterogeneity between schemas. Syntactic constructs are necessary to handle typical problems of schema integration like different units of measurement ($ vs. £) and homonyms and synomyms. Means to give schemas a new structure are useful to overcome structural differences [ScN88], e.g., if some unit is modelled by an attribute in one schema, but as an object type elsewhere [SpP91, SCG92]. Generalization is an important concept to bring together objects of the same type, but from different databases [KDN90]. Vertical fragmentation has an orthogonal effect: Objects are built by "joining" objects from different databases. Hence logical links between thus far disjoint databases must be newly specified.

Up to now, a prototype that integrates INFORMIX and the object-oriented database systems Objectivity/DB and VERSANT has been implemented, but a lot of work is still necessary.

References

[ABV92] M. Aksit, L. Bergmans, S. Vural: *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach.* In [Mad92]

[AlT92] A. Alashqur, C. Thompson: *O-R Gateway: A System for Connecting C++ Application Programs and Relational Databases.* In: C++ Conference, Portland 1992, USENIX Association, Berkeley

[BNPS94] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella: *Applications of Object-Oriented Technology to the Integration of Heterogeneous Database Systems.* In: Distributed and Parallel Databases 1994, Vol. 2

[CACM94] *Reverse Engineering.* Special Issue of Comm. of the ACM 37(5), 1994

[CaS91] M. Castellanos, F. Saltor: *Semantic Enrichment of Database Schemas: An Object-Oriented Approach.* In: Proc. of 1st Int. Workshop on *Interoperability in Multidatabase Systems* Kyoto (Japan), 1991

[Cat94] R. Cattell (ed.): *The ODMG-93 Standard for Object Databases.* 2nd edition, Morgan-Kaufmann Publishers, San Mateo (CA) 1994

[CBS94] R. Chiang, T. Barron, V. Storey: *Reverse Engineering of Relational Databases: Extraction of an EER model from a Relational Database.* Data&Knowledge Engineering 12, 1994

[ERA93] Proc. 12th Int. Conf. on *Entity-Relationship Approach.* Karlsruhe 1993

[Har92] M. Härtig: *An Object-Oriented Integration Framework for Building Heterogeneous Database Systems.* In [HNS92]

[HNS92] D.K. Hsia, E.J. Neuhold, R. Sacks-Davis (eds.): Proc. of the 5th IFIP WG 2.6 Database Semantics Conference (DS-5) on *Interoperable Database Systems,* Lorne (Australia), 1992

[Hoh95] U. Hohenstein: *Query Processing in Semantically Enriched Relational Databases.* In: Basque Int. Workshop on Information Technology (BIWIT'95) "Data Management Systems", San Sebastian (Spain), 1995

[HoK95] U. Hohenstein, C. Körner: *A Graphical Tool for Specifying Semantic Enrichment of Relational Databases.* In: 6th IFIP WG 2.6 Work. Group on Data Semantics (DS-6) "Semantics of Database Applications" 1995

[HoO93] U. Hohenstein, E. Odberg: *A C++ Database Interface Based upon the Entity-Relationship Approach.* In: Proc. of 11th British National Conf. on Database Systems (BNCOD11), Keele (England) 1993

[HTJC93] J.-L. Hainault, C. Tonneau, M. Joris, M. Chandelon: *Schema Transformation Techniques for Database Reverse Engineering.* In [ERA93]

[IEEE95] *Legacy Systems.* Special Issue of IEEE Software 12(1), 1995

[Kim92] W. Kim: *On Unifying Relational and Object-Oriented Database Systems.* In [Mad92]

[KDN90] M. Kaul, K. Drosten, E. Neuhold: *ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views.* In: Proc. 6th Int. Conf. on Data Engineering, Los Angeles 1990

[Mad92] O.L. Madsen (ed.): European Conf. on Object-Oriented Programming (ECOOP92), Utrecht 1992

[MaM90] V. Markowitz, J. Makowsky: *Identifying Extended ER Object Structures in Relational Schemas.* IEEE Trans. on Software Engineering 16(8), 1990

[NNJ93] B. Narasimhan, S. Navathe, S. Jayaraman: *On Mapping ER and Relational Models into OO Schemas.* In [ERA93]

[PeH95] G. Pernul, H. Hasenauer: *Combining Reverse with Forward Engineering - A Step forward to Solve the Legacy System Problem.* In: Int. Conf. on Database and Expert Systems Applications, 1995

[PrB94] W. Premerlani, M. Blaha: *An Approach for Reverse Engineering of Relational Databases.* Communications of the ACM 37(5), May 1994

[SCG92] F. Saltor, M. Castellanos, M. Garcia-Solaco: *Overcoming Schematic Discrepancies in Interoperable Databases.* In [HNS92]

[ScN88] M. Schrefl, E. Neuhold: *A Knowledged-Based Approach to Overcome Structural Differences in Object-Oriented Database Integration.* In: The Role of Artifical Intelligence in Database & Information Systems. IFIP Working Conf., Canton (China) 1988

[SpP91] S. Spaccapietra, C. Parent: *Conflicts and Correspondence Assertions in Interoperable Databases.* ACM SIGMOD-RECORD 1991, 20(4)

[Str91] B. Stroustrup: *The C++ Programming Language.* 2nd edition, Addison-Wesley 1991

[YaL92] L.-L. Yan, T.-W. Ling: *Translating Relational Schema With Constraints Into OODB Schema.* In [HNS92]