Implementing an Object Model on Top of Commercial Database Systems

(Extended Abstract)

Markus Tresch and Marc H. Scholl

Department of Computer Science, Information Systems – Databases
ETH Zürich, CH-8092 Zürich, Switzerland
E-mail: tresch@inf.ethz.ch

1 Introduction

The aim of the COCOON[1] project is to investigate the design of an object-oriented data model with main emphasis on efficient implementation of queries and updates. While the nested relational kernel of DASDBS is the target physical storage system for COCOON, the model is also implemented on top of commercial database systems, as first prototypes and to analyze feasibility and compare performance.

This report summarizes the experiences of implementing the COCOON object model and its object algebra COOL on top of two commercially available database systems: a relational one (Oracle) and an object-oriented one (ONTOS).

We have chosen a relational one, because the state of the art database systems for productive use are still relational, and an object-oriented one, because a couple of companies started to sell object-oriented database system products. Initially, we expected a semantic gap between our object model and relational databases, and assumed that the mapping to object-oriented databases is much more straightforward.

First we describe the mapping of COCOON type and class declarations into Oracle tables and ONTOS C++ classes. Then we investigate the transformation of object retrieval and manipulation by comparing COOL with Oracle's relational SQL and the SQL-like query interface of ONTOS.

2 The COCOON Model

Essentially, the COCOON model as described in [SS90, SLT91] is an object-function-model in the sense of [Bee89]. It is a *core* object model, meaning that it is restricted to the essential ingredients: *objects, functions, types,* and *classes*. Functions are the generalized abstraction of attributes (stored or computed) and relationships. They can be single- or set-valued. Types are arranged in a *type-lattice*, and classes in a *subclass-hierarchy*.

When the object model has been developed, the following *key properties* were identified, that appear to be essential for a model and a language for object-oriented databases:

- *object-preservation* is the central concept, meaning that the results of queries are some of the already existing objects from the database. Object preservation is crucial for many of the subsequent requirements; for example a straightforward view definition facility.
- *type/class separation* is a consequence of object preservation. If both projection and selection are to preserve objects, and if composite select/project queries are permitted, we need type/class separation in order to give a proper description of the result of such a query.
- *multiple type instantiation and multiple class membership* are again immediate consequences of object preservation: since projection, for example, changes the type, all objects in the result 'acquire' a new type, in addition to their former one.
- *dynamic reclassification of objects during updates* becomes necessary when we take into account, that objects can dynamically gain and loose types and class membership during their life time. Update operations may change type and classification of objects. (In COCOON, classes may be defined by a class predicate, so objects of the superclasses are automatically classified depending on their properties – cf. AI classification.)

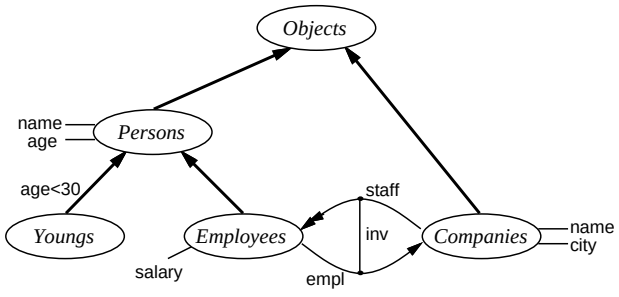
[1] Recursive acronym standing for "COcoon ... Complex Object-Orientation based on Nested relations".

The set-oriented object algebra COOL[2] includes generic query (*select, project, extent, union, intersect, difference, extract*) and update operators (*update, insert, delete, add, remove, set*). All query operators (except for extract, which returns tuples) result in a set of preexisting objects ($\rightarrow$ object preservation). Therefore queries can be used to define virtual classes (views) and as the input to update operations [SLT91, HS91]. The principle idea behind COOL's query operators is to provide a functionality similar to (nested) relational algebra.

As a sample COCOON database, we will use the following type and class definitions (see figure below) in the sequel.

```
type person isa object
      = name: string, age: integer ;
type employee isa person
      = salary: integer,
        empl: company inverse staff ;
type company isa object
      = name, city: string,
        staff: set of employee inverse empl;

class Persons : person some Objects ;
class Youngs : person all Persons where age<30 ;
class Employees : employee some Persons ;
class Companies : company some Objects ;
```



Furthermore, based on the above schema, we apply the following example queries and updates:

Q_1 : **select** [city(empl(e)) = 'Zuerich'] (e:Employees);

Q_2 : **select** [age>65]
(**select** [name(empl(e)) = 'IBM'](e: Employees));

Q_3 : **project** [name, salary, city(empl)] (Employees);

Q_4 : **extract** [name, salary, city(empl)]
(**select** [age>65](Employees));

U_1 : john:= **insert** [name:= 'JS', age:= 29] (Persons);

U_2 : **set**[salary:= 1.1*salary]
(**select** [city(empl(e)) = 'Bern'](e:Employees));

U_3 : **set**[age:= 30] (john);

U_4 : **add**[john] (Employees);

[2] Acronym for "Complex Object-Oriented Language".

3 The Two Implementations

The target system for the implementation of COCOON is the nested relational DASDBS system, for it provides very flexible (hierarchical) clustering options [SPSW90]. Therefore, many of the numerous joins that are necessary in a relational implementation of an object model can be materialized in DASDBS storage clusters [SPS87]. One reason for a relational competitor implementation is to evaluate quantitatively these possible performance gains and trade them off against the discrepancy between a commercial, highly tuned system and a university prototype.

The second reason, and the one for choosing the **Oracle** relational DBMS (Version 6.0, Oracle Corp.) [Ora88], is that this system provides (i) clusters, that can mimic 2-level nested relations, and (ii) *ROWIDs*, that is, tuple addresses are available in the query language. This allows an implementation of *join-indexes* or *link-fields* on top of the system. For example, an employee relation may, in addition to the foreign key, have an attribute *compID*, that holds, for each employee, the address of his/her company tuple. Thus, a join via the *ROWID* field requires no searching at all, rather a direct access from employee to company tuples is possible.

As a second competitive prototype we chose **ONTOS** (Release 2.0 Beta, Ontologic Inc.): an early object-oriented DBMS [AH87, ONT90]. The programmatic interface to ONTOS is AT&T's C++ 2.0 compiler. Thus C++ serves for schema definition, object manipulation, and retrieval. An additional query interface is ONTOS SQL, an extension of SQL. *SELECT–FROM–WHERE* clauses are applied to classes, and C++ expressions can be used in the formulation of queries.

Now, why implement another object model on top of an object-oriented DBMS? The answer is, that the query language of that ONTOS is not sufficiently powerful. Because ONTOS SQL returns data tuples, not objects[3], composition of

[3] Consequently, there was no need to separate types and classes in ONTOS – another deficiency for the mapping of COCOON to ONTOS.

queries, views, and queries as the input to set-oriented updates are not possible. Hence, our objectives are twofold: (i) to investigate whether a more powerful query language can be implemented on top of ONTOS SQL or whether one has to redo the whole language processor in plain C++, and (ii) what the necessary extensions to ONTOS SQL would be to make it better suited to our requirements.

3.1 Mapping the Schema Definition

... to Oracle

Our current Oracle implementation does not yet provide necessary flexibility to run experiments with different relational designs for a given COCOON schema. The only option currently included is whether or not to include *ROWIDs* together with foreign keys. Extensions are underway, here we sketch the current default realization.

Essentially, every COCOON type maps into an Oracle table (relation), and every class is represented as a view, derived from some of these tables. Thus every instance of a type is a row in at least one table, and the rows of a table constitute the 'active domain' of a type.

Transformation of functions depends on whether the they have a primitive, abstract, single-, or set-valued range. Functions with primitive range are stored attributes, and can therefore be realized as columns, using Oracle standard types (number, float, char). Set-valued functions and functions with abstract range are implemented as (connection-) tables.

```
CREATE TABLE person ( OID NUMBER(10),
            name CHAR(30), age NUMBER(10) );
CREATE INDEX Iperson ON person (OID);

CREATE TABLE employee ( OID NUMBER(10),
            salary NUMBER(10) );
CREATE INDEX Iemployee ON employee (OID);
CREATE TABLE employee_empl ( employee
            NUMBER(10), company NUMBER(10) );
CREATE UNIQUE INDEX Iemployee_empl
       ON employee_empl (employee,company);

CREATE TABLE company ( OID NUMBER(10),
            name CHAR(30), city CHAR(30) );
CREATE INDEX Icompany ON company (OID);
CREATE SYNONYM company_staff FOR employee_empl;
```

The data-definition statements above show the transformation of the COCOON example into SQL tables. Every relation holds an additional attribute to store the object-identifier. Connection-tables are generated for set-valued functions and / or functions with abstract domain. Inverse functions can be implemented by creating a synonym for the connection table. In the schema definitions shown, no *ROWIDs* are utilized. The figure below illustrates, how class-extents are calculated as select-project-joins views.

```
CREATE VIEW Persons AS
SELECT name, age FROM person;

CREATE VIEW Employees AS
SELECT t0.OID, t0.name, t0.age,
       t1.salary, t2.company empl
FROM   person t0, employee t1,
       employee_empl t2
WHERE  t0.OID = t1.OID AND t1.OID = t2.OID;

CREATE VIEW Youngs AS
SELECT * FROM person WHERE age < 30;

CREATE VIEW Companies AS
SELECT t0.OID, t0.name, t0.city,
       t1.employee staff
FROM   company t0, company_staff t1
WHERE  t0.OID = t1.OID;
```

Subtype-hierarchy, that is the relationship between supertype A and subtype B, where B inherits all functions from A, and additional functions can be defined at B, can be mapped to tables RA, RB in several ways:

- *horizontal partitioning* between tables for super- and subtype: relations RA and RB are created such that

  $A \leftarrow RA \cup \pi_{attr(A)}RB$ and $B \leftarrow RB$

- *vertical partitioning* between tables for super- and subtype: relations RA and RB are created such that

  $A \leftarrow RA$ and $B \leftarrow RB \bowtie RA$

- *no partitioning* between tables for super- and subtype: one relation R is created such that

  $A \leftarrow \pi_{attr(A)}R$ and $B \leftarrow \sigma_{attr(B)\neq NULL}R$

- *redundant, direct materialization* of tables for super- and subtype: relations RA and RB are created such that

  $A \leftarrow RA$ and $B \leftarrow RB$

The problem of physical database design is finding the best representation. Currently, we have implemented only vertical partitioning: there are

no null-value problems, and it is a reasonably good compromise between fast access to objects and easy updating tables.

Furthermore, the design can already be improved by defining clusters (e.g., for set-valued functions), or additional indexes (e.g., for the selection predicate of a view).

... to ONTOS

The schema of an ONTOS database is a collection of C++ class definitions. Internally, ONTOS uses a meta-schema to store the schema data. While C++ does not allow for dynamic class definitions at runtime, the ONTOS meta-database is fully transparent to the user and can be directly updated and queried. So, there are two ways to define an ONTOS database schema:

- *indirect,* by generating code files with C++ class definitions. They must be compiled to the database using the *classify* utility, which extends the ONTOS kernel by the definitions of the application classes.
- *direct,* by updating the ONTOS meta classes. This is mainly done by instantiating the ONTOS class *type*, the runtime-usable form of C++ class-definitions.

Of course, direct meta-schema manipulation is much more flexible. As long as there are no new C++ procedures to be added to the schema definition, no reclassification, recompilation, and linking is necessary. The direct method also offers additional possibilities of defining semantic constraints in a schema. The advantage of the indirect method is of practical nature, because this way is much more stable and secure[4], the generated C++ code is understandable, and debugging is easier. The example below shows indirect mapping of the COCOON type *company*:

```
class company : public Object {
    char *_name, *_city;
    TRef *_staff;
public:
    ...
    Set  *get_staff();
    void  set_staff(Set*,int);
    void  add_staff(employee*,int)
    void  rem_staff(employee*,int);
}
```

[4]The direct method did not work with ONTOS 1.5. Furthermore, the additional constraints are not checked and can even be conflicting.

Parts of the generated C++ class are given, as it is to be classified afterwards. COCOON functions with primitive domain are always mapped into private properties. They can only be accessed by the automatically generated `get...` and `set...` methods. Functions with abstract domain (*person, employee, ...*) realize relationships between objects and are implemented as pointers.[5] For each set-valued function, the additional procedures `add..., rem...` are produced. The generated procedures maintain consistency, such as minimum and maximum cardinalities, or inverse relationship of functions.

The next figure shows the transformation by direct manipulation of the meta-schema. New ONTOS types and properties (attributes) are constructed. Afterwards a constructor is generated for every new type, and the type is compiled to create associated structures required by the system.

```
ct = new type("company",OC_Object);
c0 = new PropertyType(0,"_staff","company",
                      staffPtr,0,private)
c0->Inverse(e0);
...
ct->generateConstructor(...);
ct->compile();
```

Distinction between types and classes prevents from a very simple one-to-one mapping of COCOON into ONTOS. COCOON types are mapped into C++ classes as shown above. There is an additional construct needed to implement COCOON classes. This is different from Oracle, because ONTOS lacks a view mechanism. Thus, the meta-class *Class* is introduced, which is mainly a persistent homogeneous set of objects. It stores references to the member-objects of the class.

Table 1 gives a summary of mapping COCOON schema definitions into Oracle and ONTOS.

3.2 Mapping Queries and Updates

... to Oracle

Mapping COOL queries to relational SQL is quite intuitive. So simple selection of objects, as for example in query Q_1, is translated into Oracle as:

[5]Instead of direct virtual memory pointers, *TRefs* (= transparent references) are used. There is an ONTOS library class, implementing these persistent references to database objects.

COCOON	*ONTOS*		*Oracle*
	indirect	*direct*	
types	C++ classes	instances of *Type*	Oracle tables (relations)
functions	C++ properties with access-functions	instances of *PropertyType*	column of table or separate connection-tables
classes	instances of *Class*		views on tables
objects	persistent C++ objects		rows of Oracle tables

Table 1: Summary of schema definition mappings

Q_1 :
```
SELECT c0.OID
FROM   Employees c0, Companies c1
WHERE  c0.empl = c1.OID
AND    c1.city = 'Zuerich';
```

However, to guarantee object-preservation and to enable nesting of operations, the identity of an object (its surrogate `OID`) must never be lost. Omitting the `OID` attribute from the projection list would perform extraction of the values from objects, such that tuples of data are produced and duplicates are eliminated. This would exactly implement COOL's **extract**-operator, as in query Q_4.

Q_4 :
```
SELECT DISTINCT c0.name,c0.salary,c1.city
FROM   Employees c0, Companies c1
WHERE  c0.empl = c1.OID;
```

Access to non-primitive functions (object- or set-valued) requires joins with the corresponding connection-tables. There we can benefit, if *ROWIDs* are included in these relations' schema.

According to COCOON's type/class separation, objects may be instances of multiple types and/or member of several classes at the same time. Thus updates may

- locally change the value of a function (e.g., raise the salary of an employee, cf. U_2);
- change class membership (e.g., change the age of a person, may remove it from the class *Youngs*, cf. U_3);
- change class-membership and type (e.g., add a person to the class *Employees*, cf. U_4).

COCOON update operations are always mapped into a sequence of SQL data manipulation statements. They directly manipulate the tables, representing types. Extensive code has to be implemented for consistency checks, since Oracle cannot express them directly.

... to ONTOS

Queries of the COCOON algebra can be transformed into ONTOS retrievals in two different ways:

- *ONTOS SQL:* An extension of SQL, where queries of the from `SELECT <expr> FROM <class|set|...> WHERE <logic expr>` can be formulated. A query-iterator is used to iterate the SQL-query over a given database in a programmatic way. The output over every iteration step is either a character string, or a list of the values.
- *COOL Interpreter:* An interpreter can be implemented in C++, traversing a syntax tree produced by a COOL parser, and thus evaluating queries on the database.

Although mapping to ONTOS SQL would be a quite elegant way, it is only feasible for very simple queries. In fact, ONTOS SQL is not object-preserving, but value-generating. This disables the mapping of nested COOL queries like for example query Q_2 to ONTOS SQL.

However, an ONTOS SQL statement is appropriate as for the COOL data extraction operator. The ONTOS SQL example below is equivalent to query Q_4:

Q_4 :
```
select name, salary, empl.city
from employee where age > 65;
```

Thus, for the object-preserving COOL operators, we have to write our own query processor. The COOL interpreter traverses the syntax tree, such that for every COOL operator, there is a corresponding C++ procedure, evaluating this operator. For example, the select-operator is implemented as a method `select`, with two arguments: a set of objects, and pointer to the node in the syntax tree, representing the selection-predicate.

```
Set *Interpreter::select (Set*, node*) { ... };
```

Such an interpreter is in the most part a reimplementation of the ONTOS SQL query facility. This is a pity, because ONTOS SQL could be easily improved to fulfill most of our requirements: if the inputs and outputs were sets of TRefs (objects) then closure, view definition, and composition would be possible. So an 'extended' ONTOS SQL statement, realizing query Q_2, could look like this:

```
Q2 : select f.name, f.salary, f.empl.city
     from f in ( select e
                 from e in employee
                 where e.empl.name == 'IBM'; )
     where f.age > 65; );
```

In the inner select–from–where block, the 'tuple' variable e itself is listed in the selection clause. This could indicate an object-preserving query.

Data-manipulation statements are completely missing in ONTOS SQL. So, even if some simple queries could be transformed to ONTOS SQL, at least all update operations must be implemented in C++. For this purpose, the generated C++-methods `get_`, `set_`, `add_`, and `rem_` are used. So, the descriptive and generic object-manipulation operators are implemented as calls of procedural methods. In order to provide set-oriented update capabilities, ONTOS SQL has to be extended with a descriptive iterator over sets (of TRefs, see above) that takes methods, or generic update operations, if any, as argument (cf. update U_2):

```
U2 : update e in employee
     set e.salary = 1.1*e.salary
     where e.empl.city == 'Bern';
```

In more generality, we would like to have the possibility to use arbitrary (object-preserving) queries in the update statement (`update o in <query>...`).

4 Conclusion

We showed that implementing a state-of-the-art object model on top of commercial database systems results in a semantic gap between the concepts needed and those offered by relational as well as early object-oriented database management systems. Particularly, we experienced that the implementation is not much more straightforward, if we take the object-oriented DBMS product instead of a standard relational one. Even though the mapping of the structured part is more direct, the operations still require much coding, since its query language and object manipulation facilities are not yet satisfactory.

In the future, we will complete these two implementations and the DASDBS realization of COCOON and then run extensive performance comparisons.

Thanks to our students Matthias Boos, Marcel Fritsch, Felix Meschberger, and Paul Wickli for doing the implementation work [BFMW91].

References

[AH87] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proc. Int'l Conf. OOPSLA*. ACM Press, October 1987.

[Bee89] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, December 1989. North-Holland.

[BFMW91] M. Boos, M. Fritsch, F. Meschberger, and P. Wickli. COOL projects. Master's thesis, Dept. of Computer Science, ETH Zurich, March 1991.

[HS91] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In *Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Kaiserslautern, Germany, March 1991. Springer, IFB 270.

[ONT90] ONTOS Inc., Burlington, MA. *ONTOS 2.0 – Reference Manual*, 1990.

[Ora88] Oracle Corp. *ORACLE - Relational Database Management System, Version 6.0*, November 1988.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. Springer, LNCS 566.

[SPS87] M.H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by nested relational kernel. In *Proc. 13th Int'l Conf. on Very Large Data Bases (VLDB)*, Brighton, Great Britan, September 1987. Morgan Kaufmann.

[SPSW90] H.-J. Schek, H.-B. Paul, M.H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), 1990.

[SS90] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT)*, Paris, France, December 1990. Springer, LNCS 470.