An Object-Oriented Database Programming Environment for Oberon

Jacques Supcik[1], Moira C. Norrie[2]

[1] Institute for Computer Systems, Department of Computer Science, ETH Zürich
[2] Institute for Information Systems, Department of Computer Science, ETH Zürich

Abstract. We describe a system designed to provide database programming support for Oberon programmers. The system is based on a generic object-oriented data model which supports rich classification structures and an algebra over collections of objects. We describe how support for the constructs and operations of this model is provided to the programmer without changes to the Oberon language and with minimal changes to the run-time system. In particular, we consider issues of support for object evolution, constraint maintenance and query optimisation.

1 Introduction

Database functionality is necessary for a large category of modern computer applications. This is the case for not only traditional commercial applications such as payroll systems or personnel record management systems, but also the complex management of documents or computer aided engineering. In most of these applications, the relational data model is not appropriate because of the complex structure and processing of the information that has to be managed.

The more recently developed object-oriented data models are much more appropriate for these sorts of applications and the goal of our project is to provide Oberon programmers with the constructs and functions needed to support such a model. In order to realize this goal, we adapted the *OM* generic object data model [7] and made it available to the programmer through a set of Oberon modules. Since our primary goal is to provide Oberon programmers with database programming support, it was a basic requirement that no change be made to the Oberon-2 language [12] and minimum changes to the run-time system. This we achieved and our experiences show that the extensibility of the Oberon System [11] provides an elegant solution to the problem of integrating the underlying database engine in the operating system.

Two important aspects of database systems are the classification of objects and the ability of objects to evolve over time. Classification is important because it helps organize the database as collections of similar objects. In our model, we support multiple simultaneous classifications; that is, that the same object can be, at the same time, in many collections. Further, in our model classification has another fundamental role, namely the attribution of properties to objects. The second important aspect is object evolution. This is important because, usually, the objects will persist for a long time and, as we cannot predict the future, we cannot know in advance all the roles that the object may play during its life time. Even if we could predict the future, we still would like to have the evolution mechanism because we may not want to always store the whole history of all objects. This paper discusses the problems of supplying constructs and operations to support classification and evolution in the Oberon environment, and how these were solved.

Persistence in Oberon has already been dealt with in other research works. In [5], Templ shows an implementation of persistence in Oberon using meta-programming. In Oberon System-3, the concept of objects and persistence is a central part of the system. Knasmüller, in his *Oberon-D* project also added persistence to the Oberon System [8]. What makes our system different is that persistence itself was not the main goal, but rather a secondary goal required to support our data model. Consequently, contrary to Oberon-D, our persistent store does not need to be orthogonal to the Oberon types since, in our model, we make an explicit distinction between *database objects* that persist and *transient objects* that do not.

The next section of this paper motivates our approach and describes our object data model and the solutions found for the evolution problem. It also explains the idea of *"Typing by Classification"* which is a central concept in our model. The third section describes the operations on the model in terms of an algebra. Section four gives an overview of the query mechanism. Section five describes briefly the implementation and some key aspects of the persistent store. Concluding remarks and a discussion of further work are given in the last section. Throughout the paper, an example of a document management system shows how the constructs and operations needed to implement a database application are made accessible to the Oberon programmer.

2 The Data Model

The role of a database is to store and manipulate information about the application domain. To specify how this information is mapped in a database, we need to specify the constructs and the operations of the database system. These constructs and operations are specified in terms of a *data model* and this section describes the data model underlying our system.

The model is a generic, object-oriented model, offering a rich classification scheme. The advantage of such a generic model is that we can use it for both conceptual modeling of the application domain and for the implementation of the database system. In the following subsections, we describe in what ways the model is "object-oriented" and what the concept of *classification* means. We also describe our support for constraints. Many existing object-oriented database management systems, for example O_2 [13], lack any support for constraints.

2.1 An Object-Oriented Model

The term "object-oriented" is still used in several ways, so we start by stating what it means in the context of the model presented here. Let us begin with the definition of "Object". An object, in our model, is an abstract representation of a thing (concrete or abstract) or of a being. This representation is abstract because outside any context, an object has no visible attribute. It has no name, no colour, no size. In fact, the only global attribute of an object is an internal unique identifier used by the system to find the object and to differentiate it from the others. Such an object with no attributes does not seem very useful in a database but, in the next subsection, we show how to place the object in a context and how this context will be used to associate attributes with the object.

Contrary to a relational database management system, an object-oriented database management system should first be able to manage objects that are more complex than simple records. Such complex objects are found, for example, in computer aided engineering databases. Second, the system must be able to deal with *inheritance* between classes. With inheritance, we mean a mechanism to allow a class to be defined as an extension of a previously defined one. The system described in this paper meets these two requirements.

2.2 The Classification

Classification is used to organize and to structure a database by placing objects in different collections. A collection represents a *semantic* grouping of objects, i.e. a set of objects with a common role in the application system. For example, a collection "Professors" could contain a set of persons giving a course in a university. In our model, a collection is itself an object. Thus, it is possible to make collections of collections and to build a nested structure in the database system. The set of all collections in a database, together with the rules that control these collections, is called the *schema* of the database. An interesting consequence of this notion of collection as a semantic grouping is that the collection now gives a *context* or a *role* to the objects it contains and, in such a role, the objects share some common attributes related to it. For example, all members of the "Professors" collection could have an attribute such as the name of their university or the number of courses they are giving. This implementation of roles using semantic grouping is an interesting alternative to the other solutions proposed for example in [1] where they have implemented the concept in a new language, namely Fibonacci, or in [2] where they implemented, in Smalltalk, a role hierarchy for evolving objects.

With this view, collections are somehow similar to Oberon *types*. The main difference is that, in Oberon, an object has only one type, and in our model, an object can be in many collections and in every collection it will exhibit attributes relative to the role represented by that collection. Our classification model is not only more powerful than the one of the Oberon language, but also than most commercial object-oriented database systems, such as O_2 [13]. Another difference is that the classification is not a rigid structure: that is, an existing object can be inserted in or removed from existing collections. An object can also be moved from one collection to another. This mechanism, called *object evolution*, allows the objects to *evolve* during their life in the database. For example, in a university database, an object representing a person could first be considered as a student, then later as an assistant and then as a professor. In each of these three phases, the object would have attributes related to the corresponding role.

Some basic attributes such as integers or strings are already implemented in the basic system, but the user is not restricted to use only these types. Actually, he can define any new attributes he wants, provided that he also implements the methods for loading and storing the attributes in the persistent store. This makes the model much more flexible than the relational model in which the first normal form [14] disallows multivalued attributes, composite attributes and their combinations.

The general idea can be summarized as "Typing by Classification", which is contrary to the usual idea of "Classification by Typing". It is the most novel concept in our system, and represents an interesting solution to the object-evolution problem. Besides this, the mechanism also offers a solution to the *Schema evolution* problem because it is always possible to add or to remove collections. We also are able to add, remove or change the attributes associated with a collection, without changing all the objects in the collection. The details concerning this feature are beyond the scope of this paper. Another interesting theme concerning collections is how to control object evolution in the database. For example, how to forbid an object to evolve from the "Professors" to the "Students" collection. This theme is covered in [9].

We can illustrate this concept of classification using the example of a document management system. Figure 1 is a graphical representation of the schema of the database. In this figure, the collections are represented by rectangles and the subcollection relation by arrows between rectangles.

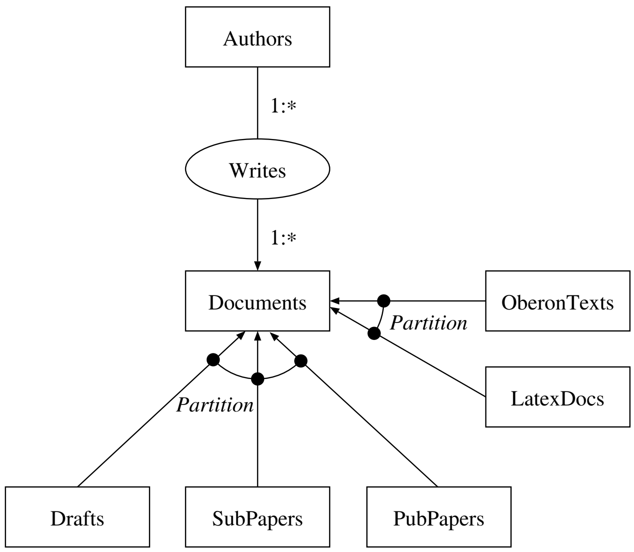

Fig. 1. Schema of the Document Management Database

In this schema, we see that "Documents" is a central collection. Actually, this collection contains all documents managed by the system. Then we have a first classification on the status of the document. In this classification, a document can be a draft, a submitted paper or a published one. We represent this classification using the three subcollections "Drafts", "SubPapers" and "PubPapers". This classification is a partition in that a document must belong to exactly one of these collections. The second classification over the documents is based on their form, or their file format. In our system, the documents we manage can be written in LaTeX or in Oberon Text Format. We represent this classification using the two collections “LatexDocs” and “OberonTexts”. This classification is also a partition. Besides these two classifications on documents, we also have a collection to represent the “Authors” of these documents. The oval-shaped box labelled ”Writes” represents an association between “Authors” and “Documents” and will be discussed in detail later.

Now that we declared the collections, we can define the attributes that the objects will have in these collections. For example, all members of “Documents” have a title. This attribute is represented as a string of characters. They also have methods to print the document and to edit it. These methods are generic and may be overridden by those defined in “LatexDocs” and in “OberonTexts”. The attributes of the “Documents” collection are summarized in Table 1 and those of the “Drafts” collection in Table 2. As you can see, the only attribute defined for the “Drafts” collection is date giving the deadline for the submission of the document.

Documents	
Attribute	*Type*
title	String
print	Method
edit	Method

Table 1. Documents Attributes

Drafts	
Attribute	*Type*
deadline	Date

Table 2. Drafts Attributes

In an Oberon application, collections are created by calling procedures of the database management system. For example, to create the “Documents” collection, the application first calls the Collections.NewCollection procedure with a first parameter saying that this collection is not a subcollection, and a second parameter saying that this collection has three attributes. The application then defines the name of each attribute. Program 1 shows the creation of the “Documents” and “Drafts” collections.

The creation of an object is done in a similar way. For example, to create an author, we first call the function Objects.NewObject, then we add the object to the collections using the Add method of the collection, and finally, we define the value of the attributes of the object in each collection using the SetObjectAttribute method of the collection. Program 2 shows this process for an author and a draft document written in LaTeX.

As we have described, classification supports the structuring of a database but, until now, we have not discussed how to prevent the structure being misused. For example, it would be possible for a person to be at the same time in the collection “Men” and in the collection “Women”. Of course, one could say that this problem is under the responsibility of the application managing the database, but it is better to integrate a mechanism into the system to detect these integrity inconsistencies in the model itself. This is the only way to globally guarantee the logical consistency of the database.

Program 1 Creating the collections

```
VAR docs, drafts, subPapers,... , authors: Collections.Collection;

PROCEDURE CreateCollections*;
BEGIN
   docs := Collections.NewCollection(NIL, 3);
   docs.SetCollectionAttribute(0, "name");
   docs.SetCollectionAttribute(1, "print");
   docs.SetCollectionAttribute(2, "edit");
   drafts := Collections.NewCollection(docs, 1);
   drafts.SetCollectionAttribute(0, "deadline");
   ...
END CreateCollections;
```

In our system, the logical consistency is defined by a set of rules called *constraints*. A constraint is an invariant that must be valid at the beginning and at the end of each *transaction*. A transaction is an atomic sequence of operations: that is, a sequence of operations that are considered as a whole. We know that the constraints are valid before the beginning of a transaction, so the only place where we check constraints is at the end of each transaction. The transactions here are used to ensure *logical* consistency of the information, but we will see later that our system also supports transactions to ensure the *physical* consistency of the data.

In our model, the consistency constraints are specified by algebraic expressions. The formal definition of a constraint is given by the following syntax:

constraint = *expr* "=" *expr*.
expr = *term* {"$\cup$" *term*}.
term = *factor* {"$\cap$" *factor*}.
factor = "("*expr*")" | *collection* | $\emptyset$.

For example, to impose that the collection Men and Women must be disjoint, we write following constraint: Men $\cap$ Women = $\emptyset$. In our example of the document management system, we have a partition constraint over the three collection Drafts, SubPapers and PubPapers. To specify this, we declare a *partition constraint* over these three collections. Program 3 shows how this may be coded in Oberon.

$Drafts \cap SubPapers = \emptyset$
$Drafts \cap PubPapers = \emptyset$
$SubPapers \cap PubPapers = \emptyset$
$Drafts \cup SubPapers \cup PubPapers = Documents$

With this type of constraint, it is now possible to check logical consistency at the collection level, but it would also be useful to check logical consistency at the object level when an object is added to a collection. For example, one could wish to restrain the age of the members of the "Employees" collection to be a number between sixteen and

Program 2 Creating the objects

```
VAR albert, josef, physicPaper, OberonPaper, CPaper: Objects.Object;

PROCEDURE CreateObjects*;
   VAR strAttr: Objects.StringAttr; dateAttr: Objects.DateAttr;
      prtAttr: PrintMthAttr; editAttr: EditMthAttr
BEGIN
   albert := Objects.NewObject();
   authors.Add(albert);
   strAttr.value := ”Albert”;
   authors.SetObjectAttribute(albert, ”name”, strAttr);
   ... (* similar for josef *)
   physicPaper := Objects.NewObject();
   documents.Add(physicPaper);
   strAttr.value := ”Physic Paper”;
   documents.SetObjectAttribute(physicPaper, ”docName”, strAttr);
   prtAttr.method := NIL (* no generic method *);
   documents.SetObjectAttribute(physicPaper, ”print”, prtAttr);
   ... (* similar for the edit method *)
   drafts.Add(physicPaper);
   dateAttr.value := Today();
   drafts.SetObjectAttribute(physicPaper, ”deadline”, dateAttr);
   latexDocs.Add(physicPaper);
   prtAttr.method := LatexPrint;
   latexDocs.SetObjectAttribute(physicPaper, ”print”, prtAttr);
   ... (* similar for the other documents *)
END CreateObjects;
```

seventy-five. It is also possible to define such constraints in our model. These are specified by a Boolean function that takes an object as argument and returns a value indicating whether or not the object conforms to the constraint. In Oberon, this function is defined by:

```
PROCEDURE (o: Objects.Object): BOOLEAN
```

In the preceding section, we specified how to define a constraint, but we did not explain what happens when a constraint is violated at the end of a transaction. Two solutions are possible: either the transaction is simply rejected and the database is reset to the state before the transaction, or the system first tries to restore the constraint by updating the database and, only in the case where the system is not able to restore it in an unequivocal way, is the transaction rejected. In our system, we chose to investigate the second approach dealing with constraint *propagation*.

When a constraint over collections is violated, the system tries to restore the database to a consistent state by propagating the changes made during the transaction. Suppose, for example, that the constraint “Men ∪ Women = Persons” is defined in a database and that, at the end of a transaction, the system detects that an object has been added to the “Women” subcollection, but not to the collection “Persons”. The constraint

Program 3 Definition of Constraint

```
PROCEDURE DefineConstraint;
  VAR c1, c2, c3, c4: Constraints.Constraint;
BEGIN
  c1 := Constraints.NewConstraint(Constraints.NewIntersection(
     Drafts, SubPapers), Collections.emptyCollection);
  c2 := Constraints.NewConstraint(Constraints.NewIntersection(
     Drafts, PubPapers), Collections.emptyCollection)
  c3 := Constraints.NewConstraint(Constraints.NewIntersection(
     SubPapers, PubPapers), Collections.emptyCollection)
  c4 := Constraints.NewConstraint(Constraints.NewUnion(
     Drafts, Constraints.NewUnion(SubPapers, PubPapers) , Documents)
END DefineConstraint;
```

is thus violated, but the system can restore it by propagating the insertion of the object to the collection “Persons”.

In the case of constraints on objects, the solution of how to deal with violation is left to the procedure checking the constraint. The function can either signal an error by simply returning FALSE or modify the object and return TRUE to signal that the object is valid.

Finally, we describe how relationships between objects are represented in our model. For example, we would want to represent the relation between the husband and his wife, or between an author and the documents he wrote. In our model, relationships are represented by *“associations”*. An association is a collection in which members are special objects, called *pairs*, representing the two related objects. The associations also have a direction. A link always goes from the *source* of the association to its *destination*. For example, the collection “Authors” is the source of the association “Writes” and “Documents” its destination. Our model supports only *binary* associations, but since it is always possible to represent associations with more than two members using many binary associations, this restriction does not limit the expressiveness of the model.

There is another type of constraint for associations, namely the *cardinality* constraint. This constraint verifies that neither too many nor too few objects are related through a given association. Such a constraint is given by two pairs of numbers: (m:n $\rightarrow$ o:p). The first pair (m:n) specifies that every member of the collection at the source of the association must participate in at least m and at most n instances of the relationship represented by the association. If n is replaced by a star ($*$), this means that there is no maximum limit. The second pair is similar but for the collection at the destination side of the association.

In our document management example, the cardinality constraint of the “Writes” association is (1:$*$ $\rightarrow$ 1:$*$). This means that an author has to write at least one document, and that a document needs to have at least one author.

Now consider the following scenario in our document management system: Albert is writing a paper on Physics using LaTeX and Josef is writing two papers, one on Oberon and the other on C++, using the Oberon Text Editor. Then Albert submits his paper, which has to "evolve" from the Drafts collection to the SubPapers one. Later on, the paper of Albert is accepted and thus will evolve again from the SubPapers collection to the PubPapers one. Josef does the same, but his paper on C++ is rejected. He decides then to rewrite part of it, convert it to LaTeX and send it to another conference. So his document moves from the OberonTexts to the LatexDocs one, moves back from the SubPapers collection to the Drafts one and then from the Drafts one to the SubPapers one again. Now the paper is submitted for the other conference, and this time the paper is accepted and it can then move from the SubPapers collection to the PubPapers one.

The object evolution mechanism in our system can easily support such a scenario. For example, if a paper has to evolve from the "Drafts" to the "SubPapers" collection, we first remove it from "Drafts", then add it into "SubPapers". We can then define the new attributes that the object has in its new collection. Program 4 shows this evolution process.

Program 4 Submit a Paper

```
PROCEDURE SubmitPaper*(paper: Objects.Object);
   VAR dateAttr: Objects.DateAttr;
BEGIN
   drafts.Remove(paper); subPapers.Add(paper);
   dateAttr.value := Today();
   subPapers.SetObjectAttribute(paper, "submissionDate", dateAttr);
END SubmitPaper;
```

3 The Algebra

In the past, one of the main criticisms made against object-oriented database systems was their lack of an associated algebra and query language. In this section, we present an algebra, based on [6], [7], in which the operations on collections of our model are defined. Algebra operations always generate collections of objects. For other operations such as aggregation, the user has to write specific Oberon code. For example, if we are interested in the number of papers written by a given author, we can use the algebra to retrieve the corresponding papers, then use an Oberon procedure to compute the cardinality, that is the number of the members, of the result.

The three basic operations on collections are those from set theory. They are the *intersection*, the *union* and the *difference*. These operations are valid on all collections. Two other operations are defined for all collections: the *select* operation used to extract objects that satisfy a given predicate from a collection and *flatten* that takes a collection of collections and flattens them to a single collection.

The *select* operation comes in different flavours depending on how the predicate is defined. For simple queries, the selection can be done by searching for a given value, or a range of values, in an index, but the system allows also for more complex selections, or for selections for which no index exists. In this case, the predicate is specified by an Oberon function, similar to the one we used for constraints, that takes an object as parameter and returns a boolean indicating if the object satisfies the select condition or not. This form of selection is very powerful because the only limiting factor for the predicate expressiveness is the expressiveness of the Oberon language itself.

Some operations of the algebra are defined only for associations and using them with unary collections is forbidden by the system. The *Domain* function, which extracts the source of an association and *Range*, which extracts its destination, are typical examples. Here is the formal definition for these operations:

– Domain: $\mathsf{dom}\ S = \{x \mid \exists\, y : (x, y) \in S\}$
– Range: $\mathsf{rng}\ S = \{y \mid \exists\, x : (x, y) \in S\}$

Range restriction is also such an operation. It takes an association A and a collection C and forms an association comprising all those pairs of A with second value in C. Formally, we can write:

– Range Restriction: $A\ \mathsf{rr}\ C = \{(x, y) \mid (x, y) \in A \wedge y \in C\}$

In addition to these, the algebra also has operations to compute the *inverse* of an association, to *compose* associations and to *nest* or *unnest* associations.

Each operation is made available to the application programmer as an Oberon procedure. Program 5 shows how, in our document management system, we can find all published documents written in LaTeX. In this program, the result of the Algebra.Intersection function is a temporary collection C. The Collections.Enumerate procedure is an evaluator that applies the function DisplayDoc to every member of the temporary collection C.

Program 5 find Published Documents Written in LaTeX

```
PROCEDURE FindPubDocInLatex*;
   VAR C: Collections.TempCollection;
BEGIN
   C := Algebra.Intersection(pubDocuments, latexDocs);
   Collections.Enumerate(C, DisplayDoc);
END FindPubDocInLatex;
```

4 Query Processing

Having outlined the algebra, we can now describe how queries are evaluated. This process can be divided into three phases: A *front-end* reads and analyses the query given by the user and generates an intermediate structure representing the query in question. An *optimizer* transforms this structure to make its evaluation more efficient. Finally, a *back-end* evaluates the query represented by the intermediate structure and returns to the user a collection of objects matching the query. The functioning of the front-end is straightforward and will not be described further here. However, in this section, we describe briefly the intermediate structure, the functioning of the optimizer and of the evaluator.

To handle a query easily and efficiently, the system needs to represent it in an appropriate form. We use a *"syntax-tree"* to represent the queries. In such a tree, each operation previously defined in the algebra, and each reference to a collection, is represented by a node.

For example, consider the following query: "Find all authors who have published a document written in LaTeX". An algebraic representation of this query would be:

$$\mathsf{dom}(Writes \ \mathsf{rr} \ (PubPapers \wedge LatexDocs))$$

Then we have to build the syntax tree representing the query. Figure 2 shows a graphical representation of that tree. There are two ways of introducing queries into an application. We can either write a string representing the query in a query language and give the string to the front-end, or we can bypass the front-end and build the tree directly using procedures from the system. Program 6 shows a procedure that directly builds the syntax-tree. The advantage of this second method is that the validity of the query is checked at compile-time and not only at run-time.

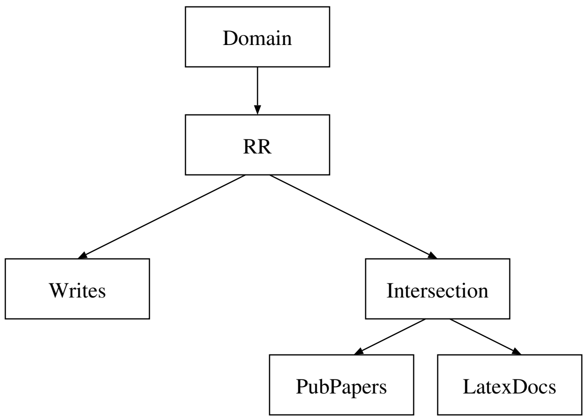

Fig. 2. Example of the internal representation of a query

The optimization phase transforms the tree to make its evaluation by the back-end more efficient. The tree being modified will possibly contain special "optimized" nodes that could not have been produced by the front-end. For example, a closure operation followed by a selection will be replaced by a special "ClosureSelect" node. This special node will prevent the back-end from producing huge intermediate collections.

The last phase of the query processing is the evaluation of the syntax-tree representing it. This last phase is similar to the evaluation of an arithmetic expression, but with collections instead of numbers.

Program 6 Procedure to Build a Syntax-Tree

```
PROCEDURE BuildQuery*;
   VAR n: Queries.Node;
BEGIN
   n := Query.Intersection(PubPapers, LatexDocs);
   n := Query.RR(writes, n);
   n := Query.Domain(n);
END BuildQuery;
```

5 Overview of the Implementation

The persistent store is implemented using the *"log principle"* [3]. Each time an object is written to the store, a new space is allocated and the object is written in this new space. In other words, an object is never changed "in place" but always completely rewritten. With this mechanism, the system is very robust because every operation can be undone allowing the system to always recover by restoring a previous status. This is used to implement transactions and to recover in the case of physical failures such as power failure or a disk crash.

The disadvantage of this mechanism is that a lot of "dead objects" stay in the store, using a lot of space. Therefore, the mechanism also requires a good garbage collection process to eliminate the garbage from the memory. However, if the garbage collector process tries to re-organize the whole store, the length of time of database unavailability would be too great. We solved this problem by implementing an *incremental* garbage collector. Using this method, only a small part of the storage is reorganized and the system remains available for other transactions. Details of the transaction mechanism is beyond the scope of this paper.

In the persistent store, the address of an object may change over time. In fact, it will change each time the object is written and each time the garbage collector relocates the object. So, to find the object without having to scan the whole database, we implemented an *id table*. This table is indexed by the identifier of the object and gives its physical location. When an object is moved, the corresponding entry in the table is modified. The table itself must persist, so we divide the persistent store into two domains: the id-table, which starts at the beginning of the store and grows toward the end, and the space for the objects which starts at the end of the store and grows towards the beginning. Figure 3 shows this structure. When a program needs to load an object from the store, it can either access it through its identifier or using a root-table, similar to the one used in PS-algol[10]. This table is nothing more than a collection of pairs <name, id> which allows an object to be found using a name instead of its id.

Having described the structure of the persistent store, we now show how the objects themselves are represented. Figure 4 shows the implementation of an object representing a draft document. The `belongsTo` structure is a dynamic array that indicates which collections contain the object, and for each collection, it defines the attributes that the object has in the collection. In this figure, we see that the object (ID=35) belongs to

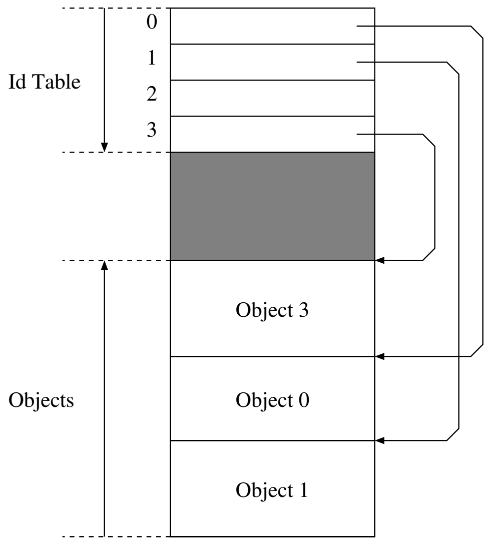

Fig. 3. Structure of the Persistent Store

Documents (ID=7), Drafts (ID=12) and OberonDocs (ID=47). In Documents, it has the attributes Relativity (Name), PrintMth (Print) and EditMth (Edit). In Drafts, it has 1.9.97 (Deadline). In OberonDocs, it has OPrintMth (Print) and OEditMth (Edit).

Figure 5 shows the implementation of a collection. As we can see, a collection has not only the ID as a predefined attribute, but also the ID of the parent collection in the case of a subcollection. In addition, it has a list of attribute names defined for this collection and an extension containing the references of the members of the collection. In this example, the collections Drafts (ID=12) is a subcollection of Documents (ID=7), it has only one attribute ("deadline") and contains the objects 35, ...

We implemented this system using the original Oberon-2 Compiler with no changes to the Oberon run-time system. The system now runs under HP-Oberon [4] V4.4, but since we developed it with portability in mind, it should be easy to port it to other Oberon systems, such as Oberon System 3 or Oberon/F.

6 Conclusion

We have presented an environment to provide database programming support for Oberon programmers. This support is realised through a combination of a module library for the management of collections of database objects and an extended run-time system for the implementation of a persistent store.

The underlying data model is an adaptation of an existing generic object-oriented data model. The key features of this model are its support for conceptual modelling in terms of classification and association constructs and for query processing in terms of its algebra over collections of objects. As a result, unlike most other object-oriented

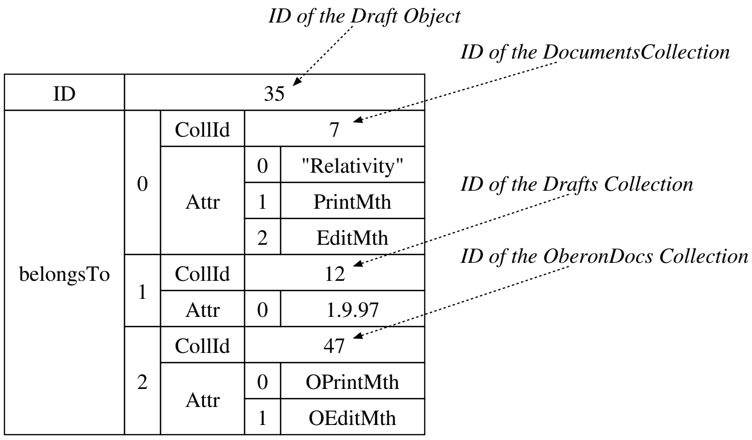

Fig. 4. Structure of a Draft Object

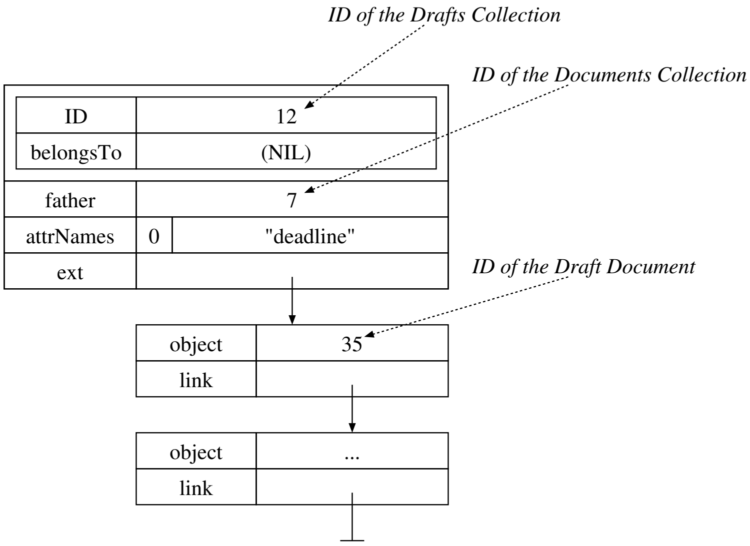

Fig. 5. Structure of the Documents Collection

programming systems, we provide not only data persistence, but also constraint maintenance, programming by querying and support for object evolution.

The current status of the system is that we have implemented a first version of the persistent store together with the basic modules for the management of database collections and their associated constraints. Investigations on the performance of the persistent store and refinements for performance are to be undertaken.

While all aspects of both algebraic and physical query optimisation are outside the scope of the current project, a general framework that enables query optimisation has been established and some simple optimisation strategies investigated. It is hoped that the topic of query optimisation will be investigated thoroughly in future work. Other issues for future research include distribution, concurrency control, advanced transaction management and additional support for schema and database evolution.

References

1. A. Albano, R. Bergamini, G. Ghelli and R. Orsini. An object data model with roles. In *Proceeding of the 19th VLDB Conference*, pages 39–51, Dublin, Ireland, 1993. Morgan Kaufmann.
2. G. Gottlob, M. Schrefl and B. Röcki. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), July 1996.
3. J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, January 1989. Also appears as University of California, Berkeley, Technical Report UCB/CSD 88/467.
4. J. Supcik. HP-Oberon™: The Oberon implementation for Hewlett-Packard Apollo 9000 Series 700. Technical Report 212, Institute for Computer Systems, ETH Zürich, Switzerland, 1994.
5. J. Templ. *Metaprogramming in Oberon*. PhD thesis, ETH Zürich, Switzerland, 1994.
6. M. C. Norrie. *A Collection Model for Data Management in Object-Oriented Systems*. PhD thesis, University of Glasgow, Scotland, 1992.
7. M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *12th Intl. Conf. on Entity-Relationship Approach*, pages 390–401, Dallas, Texas, December 1993. Springer-Verlag, LNCS 823.
8. M. Knasmüller. Adding persistence to the Oberon-System. Technical Report 6, Institut für Informatik, Johannes Kepler Universität Linz, Austria, 1996.
9. M. Norrie, A. Steiner, A. Würgler and M. Wunderli. A model for classification structures with evolution control. In *15th International Conference on Conceptual Modelling. ER 96*, Cottbus, Germany, 1996.
10. M. P. Atkinson, K. J. Chisholm and W. P. Cockshott. PS-algol: an Algol with a persistent heap. *ACM SIGPLAN Notice*, 17(7), July 1981.
11. M. Reiser. *The Oberon System. User Guide and Programmer's Manual*. Addison-Wesley, 1991.
12. N. Wirth and M. Reiser. *Programming in Oberon. Steps beyond Pascal and Modula*. Addison Wesley, 1992.
13. O. Deux. The O_2 system. *Communication of the ACM*, 34(10):34–48, October 1991.
14. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, second edition, 1994.