Oggetto: An Object Oriented Database Layered on a Triple Store

J. A. MARIANI
Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR

Interest in object oriented database (OODB) systems continues to grow. This paper describes the implementation of an OODB by layering over a triple store. The OODB thus implemented allows experimentation with an object-oriented query language. It is shown how this system, Oggetto, is capable of handling the four tasks outlined in ref. 2.

Received December 1989, revised May 1990

1. INTRODUCTION

Object-oriented databases (OODBs) are increasingly in demand in areas such as CAD/CAM, office information systems, software engineering environments and multi-media systems. They support the storage, retrieval and manipulation of flexible, fine-grained and complex structures. They also support schema evolution. The schema of a database consists of a description of the structure of the data held; data about the data, or metadata. The application areas above require metadata to be as flexible and changeable as data itself. Contrast this with conventional database management systems (DBMS), where schemas tend to be graven in stone; any restructuring of the schema can be a major task requiring the database to be unavailable for significant periods.

Several attempts at implementing an OODB have been layered on existing DBMS including MOKUM[25] (a relational DBMS and Prolog), IRIS[9] (based on common Lisp), GemStone[14] (based on Smalltalk), and Postgres[24] (based on Ingres). Our work is similar; the substratum being a binary-relational storage structure. The advantages are no different from any other layered architecture; we use the functionality of the storage structure to support the functionality of the OODB. Where the substratum provides functionality close to that required by the superstructure, however, this eases the implementation task. We hope to show that binary-relational storage structures are particularly apt for the support of an OODB superstructure. An important view of our work is that we may consider the OODB as a high-level interface to the underlying binary-relational storage structure.

Easing the implementation task allows the rapid construction of a prototype OODB and therefore expedites experimentation with its features. An object-oriented query language, which contains a transitive closure primitive, has been designed and implemented. This language and its use are described later.

We now introduce the two database technologies of concern: object-oriented databases and binary-relational storage structures.

1.1 Object databases

This section provides a brief overview of the power and capabilities of an Object Oriented Database (OODB) system. For further details see ref. 16. Using an OODB, we set up a schema by declaring new types, such as:

```
type person {
   string name;
   int age;
   char gender
   };
```

Example 1: Declaring a type person

Where `string`, `int` and `char` are basic types supported by the OODB. We can now introduce instances of the type, such as:

```
inst person DAVID (
   name: = 'David';
   age: = 32;
   gender: = 'm'
   );
```

Example 2: Declaring an instance of type person

Note that we have given the instance a unique name `DAVID` known as an id that can be referenced elsewhere in the data. An aspect of most object-oriented systems is inheritance. A type can inherit the attributes of another type; we can thus build hierarchies or networks if inheritance is allowed from a single or multiple parent types. For example:

```
type marriedPerson (
   person spouse;
   set of person hasChild
   ) inherits person;
```

Example 3: Declaring a type marriedPerson

A married person possesses all the attributes of person (name, age and gender), but has the additional attributes of a (single) spouse and (zero to many) children. Attributes are considered to be single valued by default, however, set valued attributes are flagged by the reserved words `set of`. Notice the use of a user-defined type person within this declaration. We can continue the process of inheritance further:

```
type wife (
    string maidenName
    ) inherits marriedPerson;
inst wife RUTH (
    name: = 'Ruth';
    age: = 30;
    gender: = 'f';
    spouse: = DAVID;
    hasChild: = (CLAIRE, PAUL, EMMA);
    maidenName: = 'Clarke'
    );
```

Example 4: Declaring a type wife and a subsequent instance

In this example, we can see the use of ids to reference other objects.

OODBs are flexible, it is easy to add a new attribute to a type (e.g. string religion to type person). Consider how this would be carried out in a relational system. The manipulations required to add a new attribute in our OODB are detailed in Appendix 1.

We can associate methods with types. A method is analogous to a procedure in that it consists of a body of 'code' which can be executed. Methods are tied to the declaring type, and can only be used in association with objects of that type. For example, it is unusual for a database to have a field 'age' which consists of a stored value. Instead, it should have a computed field, which references an algorithm that calculates the person's age based on their (stored) date of birth and the current date. In the type declaration for person, we might find:

```
(
date dateOfBirth;
method int age
    [
    todaysDate - dateOfBirth
    ];
);
```

where the type date has been defined, and the arithmetic operator minus also defined between two dates. A person's age attribute can be accessed as before; it makes no difference if it is stored or computed.

This adds a dynamic aspect to the database. We now require the ability to

* store methods within the database;
* execute methods whenever data is accessed.

1.2 Binary-relational storage structures

The binary-relational model (BRM) first came to prominence in 1974[1] and was further developed by Senko's work on DIAM (data independent access model) I and II.[19] The BRM consists of entities (which are any 'thing' that can be identified and is of interest) which have binary relationships between them. Such a relationship can be represented by a triple of the form

(subject, relation, object).

To represent the fact that John kicked the ball, we can say that John is the subject, the ball is the object, and the relationship is that of being kicked.

(John, kicked, the ball).

This falls out naturally as our original fact was a triple.

If we take a diagram (Fig. 1) from Abrial,[1] we can see a semantic network of information showing nodes (things) and connections between them. Abrial labels the connections with the access functions which can be applied. There are two such functions, each being the inverse of the other, i.e.

```
personofage(50) = {PETER, MARY}
age(JOHN) = {27}
```

The data can also be intuitively organised into what Abrial refers to as categories, i.e. JOHN, JANE, PETER and MARY are persons. Abrial goes on to show how arbitrary *n*-ary relations can be reduced to a group of binary relations. We can represent the schema of this data in a similar diagram, or choose to model it in the ER (entity relationship) model,[5] this time differentiating which connections are relationships and which are attributes (Fig. 2).

We can store the metadata and the data directly in a BRM storage structure.

(PERSON has_spouse PERSON)
(PERSON has_child PERSON)
(PERSON has_age age)
(PERSON has_sex sex)

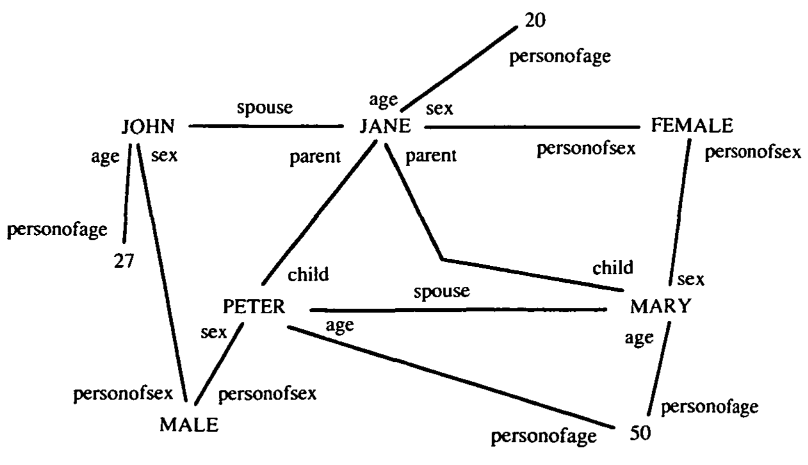

Figure 1. Abrial's diagram.

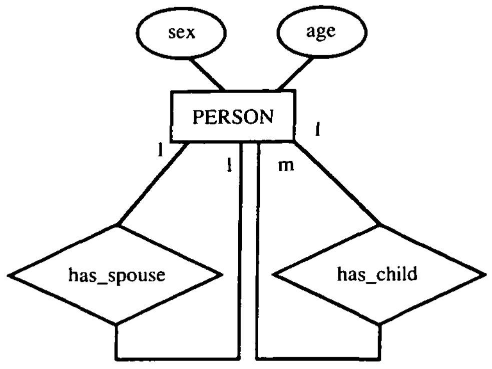

Figure 2. The ER diagram.

Finally, we can store the data within an OODB.

```
type PERSON (
    PERSON spouse;
    set of PERSON child;
    int age;
    char sex;
    );
```

Senko[19] shows how names are used to stand for things in the real world, and that 'world things' are divided up into groups. Within a group, each member is uniquely identified by a member name. In DIAM, it is possible for any member to have an infinite number of properties or associations with other member names; this facility within the BRM in general assists with the modelling of objects.

DIAM has a layered architecture, as shown in Fig. 3.

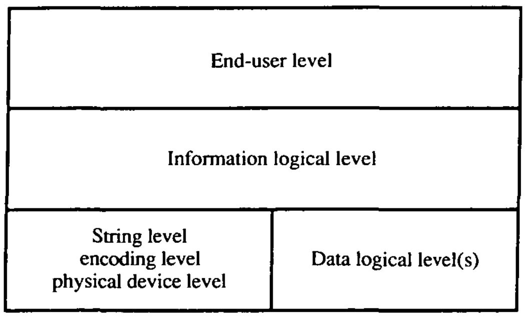

Figure 3. The DIAM layered architecture.

In Senko,[20] it is shown how the three levels correspond to ANSI SPARC schema levels, and how binary associations at the conceptual schema level can be mapped to the external schema level of hierarchic records, and to indexes, lists and hierarchic records at the internal schema. Senko claims that this flexible mapping means that binary representations can avoid the inefficiencies of stored binary file organisations.

In Sharman and Winterbottom,[21] the universal triple machine (UTM) is introduced, which supports six basic operations to manage a collection of triples. It is suggested that the UTM's operations are capable of supporting the data structures and operations found in a wide range of database and AI systems. The UTM's data repository consists of two stores: the name store and the triple store. The name store maintains a mapping between the internal unique identifiers and the external character representations of values appearing in the fields of a triple. The operations are described below:

```
Name store operations:
    INSERT_NAME (name) → identifier
    DELETE_NAME (identifier)
    FIND_NAME (identifier) → name
Triple store operations (all operands are identifiers):
    INSERT_TRIPLE (sub, rel, obj)
    DELETE_TRIPLE (sub, rel, obj)
    FIND_TRIPLE (sub,rel,obj, mask)
      → {(sub, rel, obj)}
```

To find a triple, we set up a query template by providing the known values for a field, and a three-bit mask to indicate which fields are known. For example, to find out what age everything is, we set up the query template

FIND_TRIPLE (, hasAge,, 010)

which then returns a set of triples that match the query template (i.e. all triples that have 'hasAge' in the relation field).

This work is relevant as it specifies and provides a dynamic, active aspect to triple stores as opposed to the normally static repositories. Some OODBs (for example, ref. 25) have been layered on Prolog, which possesses the ability to declare data and algorithms (rules), and to access stored and computed data in exactly the same way. These aspects lend Prolog to the support of an OODB but we argue that the lack of an appropriately efficient DB storage mechanism renders this approach invalid in the long term. Van de Riet,[25] addresses this by storing the static aspect of objects in a relational database. Sharman and Winterbottom,[21] show how Prolog algorithms can be expressed in the terms of their UTM primitives.

Thus we can implement a triple store capable of storing both facts and algorithms, and provide a substrate that can hold both objects and methods.

The rest of the paper is structured as follows. In section 2, we introduce the process of schema evolution and indicate how Oggetto meets the requirements. The Oggetto query language (OQL) is introduced in section 3, with the aid of examples. Section 4 uses the four tasks presented in ref. 2 to illustrate the power of Oggetto. We return to triple stores in section 5 and discuss the most appropriate underlying architecture for the support of object stores. Section 6 briefly considers some future work, and our conclusions are presented in section 7.

2. SCHEME EVOLUTION IN OGGETTO

An important requirement for OODBs is the ability for the schema to evolve. Triple stores are uniquely placed to meet these requirements as the metadata is stored with the data itself,[10, 17] and we can thus apply triple store operations to the metadata. By describing how Oggetto supports schema evolution, we will also illustrate the mapping between object and triple stores.

The schema of an OODB is usually viewed as the structure built up by the inheritance mechanism, which can be considered as a directed acyclic graph. The nodes in this graph describe the types and their attributes. The edges of the graph represent the inheritance relationship.

Banerjee[3] describes the requirements for schema evolution as operations on the graph, and those supported by Oggetto are detailed in Appendix 1; the action of adding a new class is described here.

In response to a schema command triples are generated and/or deleted. In the text, we refer to schema and data triples, solely to aid in description; there are no such distinctions in the system.

syntax: `type` typeName (attributes) [`inherits` typeList];

semantics: There are several base types such as integers and strings, out of which more complex types can be built. These complex types can, in turn, be referenced by other new types. Consider example one, the introduction of a new type, person. This is converted into the following set of schema triples:

```
(person hasAttribute name) /*
  associates the name of an attribute
  with the type */
(person name string) /* associates the
  type of an attribute with the type,
  attribute pair */
(person hasAttribute age)
(person age int)
(person hasAttribute gender)
(person gender char)
```

To create an instance of a type, we provide an id and as many of the expected attributes as we can. The declaration of example two is converted into the following set of data triples:

```
(DAVID hasType person)
(DAVID name 'David')
(DAVID age 32)
(DAVID gender 'm')
```

Inheritance is supported with the use of supertypes, and is specified at type creation, as shown in example three. This type declaration is converted into the following set of schema triples:

```
(MarriedPerson hasAttribute marriedTo)
   /* name of attribute */
(MarriedPerson marriedTo MarriedPerson)
   /* type of attribute */
(MarriedPerson hasAttribute hasChild)
(MarriedPerson hasChild Person)
(MarriedPerson inherits Person)
(MarriedPerson→Person inOrder 1)
```

The last two triples refer to the inheritance information. The penultimate states that `MarriedPerson` inherits the attributes of `Person`. The last codifies that information within the subject field of the triple as `MarriedPerson→Person`, and states that `Person` is the first type that `MarriedPerson` inherits from. The order of inheritance is important in multiple inheritance schemes, and Oggetto supports that concept.

Instantiation of an attribute can be deferred. As an example, here are the declarations of two object instances, DAVID and RUTH.

```
Inst MarriedPerson DAVID (
   name: = 'David';  /* no references to
     marriedTo */
   );
inst wife RUTH (
   name: = 'Ruth';
   marriedTo: = DAVID;
   );
/* now we can fill in DAVID's missing
  attributes */
DAVID→marriedTo: = RUTH;
```

Attributes can be multivalued, thus there is a distinction between this model and relations. However, like the relational model, we insist that any complex item of data is given its own type; this means we can form queries that relate to these items.

3. THE OGGETTO QUERY LANGUAGE

3.1 Introduction

Research with Oggetto has led to the development of a query language that owes much of its power to the set-based functional model of relational query languages. Much of the literature features adaptations of SQL,[6] which has become a *de facto* standard. However, we have departed from this to arrive at a query language with an object flavour.

As in most modern QLs, the Oggetto query language (OQL) combines both the declaration of types and instances [the data declaration language (DDL)] with the data manipulation language. The DDL has been described in Section 2 and Appendix 1.

An important addition to OODBs is that of active components (methods). We begin by considering the architecture of the Oggetto system.

3.2 The Oggetto architecture

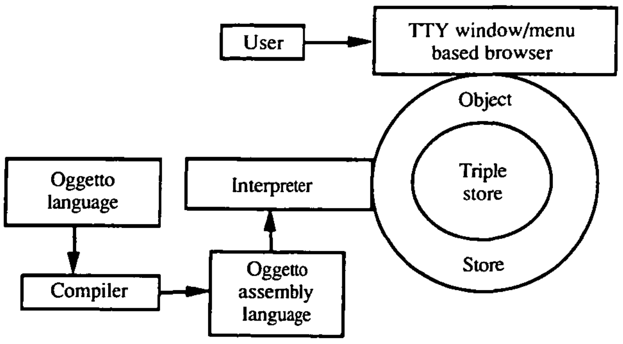

Figure 4. The Oggetto architecture.

A 'compiler' takes the Oggetto language and produces a sequence of low level 'assembly language' instructions. These are a mixture of instructions that could be considered as triple store 'machine code' instructions, such as 'insert triple CLAIRE has.type person', and some that could be considered as object store 'machine code' instructions, such as 'set attribute age of CLAIRE to 5'. These instructions are then interpreted.

3.2 The query language

The query language has a message passing syntax. To find out something about people, we direct a query message to that type name. For example:

```
[person query name = 'David'];
```

returns a list of ids of objects of type person that have the name attribute set to 'David'. Oggetto has several built-in messages that all objects respond to (this is conceptually similar to a base object type possessing several methods); query is one of them. Another is the print message, which is followed by a list of attributes.

```
[DAVID print: name, age, marriedTo];
```

which will result in the printing of 'David', 30, RUTH. Embedded messages are supported, so we can combine the above messages as:

```
[[person query name = 'David'] print:
    name, age, marriedTo];
```

This set-based approach – in that we allow messages to return multiple ids which can then in turn be sent messages – removes the need for the unwieldy 'for' loop constructs features in some QLs.

For example, in the Iris QL, where we have a 'Personnel' type (much like a relation) as follows:

```
Create Personnel (name, birthplace,
  rank) instances
    Kirk ('J.T. Kirk', Earth,
      'Commander'),
    Spock ('Spock', Vulcan, 'Lt.
      Commander'),
    McCoy ('L.H. McCoy', Earth, 'Lt.
      Commander');
```

to find the names of personnel born on Vulcan requires the following query:

```
Select name(p)
    for each Person p
    where Vulcan = birthplace(p);
```

The same query in SQL with the appropriate relation would be:

```
select name
from personnel
where birthplace = Vulcan
```

The SQL query is set-based as opposed to the Iris 'for' construct. In OQL, the query would be:

```
[[personnel query birthplace = Vulcan]
  print: name];
```

As well as the usual dyadic relational operators (=, >, etc.), Oggetto supports a monadic operator, exist. This is used to tell if an instance possesses a named attribute. For example, we can use the exist operator to obtain the ids of married people who are also parents as follows:

```
[ MarriedPerson exist hasChild ];
```

Programmers can explicitly build a list of ids using the list directive. For example,

```
list (DAVID, RUTH)
```

builds a list of the ids DAVID and RUTH. Wherever an id may appear in a list directive, so too may a message i.e.

```
list ( [ DAVID age], [RUTH age] )
```

builds a list of David and Ruth's ages.

OQL supports arithmetic operations such as sum and mult as conventional messages. Arithmetic messages appear as follows:

```
[[[person query gender = female ] age ]
  sum]
```

would sum up the ages of all female persons.

3.4 Methods

In terms of methods, the most important assembly instruction is the message instruction. For example, the OQL message

```
[ person query name = 'David' ];
```

is compiled to

```
message (person, query, operand, name,
  operand, 'David', operator, =)
```

The compiler converts the query expression to reverse polish and flags the type of each item in the assembly instruction (to avoid the repetition of identifying the type in the interpreter).

Embedded messages are dealt with by using an object store register, id_list. This contains the results of the last message. The message

```
[ [ person query name = 'David' ] print:
  age, gender ];
```

becomes

```
message (person, query, operand, name,
  operand, 'David', operator, =)
message (id_list, print, age, gender)
```

A method is any allowable message. To define an hasGrandChild method for MarriedPerson, we add the following to the declaration of the type:

```
method hasGrandChild
    [ [ self hasChild ] hasChild ]
endMethod
```

An instance of the type (or subtype) can access the method in the same way it would an attribute:

```
[ [ THOMAS hasGrandChild ] print:
  name ];
```

This will print the names of THOMAS's grand children.

The Oggetto compiler compiles the message that is a method's body exactly as it would messages in the DML part of an Oggetto program, but instead of placing the assembly instructions in the output file, records them as schema triples in the following format:

```
(MarriedPerson, hasMethod,
  hasGrandChild)
(MarriedPerson→hasGrandChild, 1,
  'message (self, hasChild)')
(MarriedPerson→hasGrandChild, 2,
  'message (id_list, hasChild)')
```

When the interpreter has to execute a method, it retrieves the assembly instructions which make up the body of a method, sorts them in order, and then interprets them as it would instructions from the assembly file.

4. THE FOUR TASKS OF ATKINSON AND BUNEMAN

4.1 Introduction

Atkinson and Buneman[2] presents a survey of types and persistence in database programming languages. To illustrate their comparisons, a database containing the inventory of a manufacturing company and four tasks are described. In particular, it contains the representation of a 'parts explosion'; composite parts are built up from other parts, which may be base parts or further composite parts. The structure of a composite part forms a directed acyclic graph. The four tasks are as follows:

Task 1: Describe the database.
Task 2: Print the names, cost and mass of all imported parts that cost more than $100.
Task 3: Print the total mass and total cost of a composite part.
Task 4: Record a new manufacturing step in the database, that is, how a new composite part is manufactured from sub-parts.

These four tasks were chosen to be characteristic of database programming and yet some are difficult to implement. For example, task 3 is impossible in most relational query languages. We use these tasks here to illustrate the power of the Oggetto system.

The Oggetto query language schema for this data (performing task 1) is shown below:

```
schema alias dollars, int;
schema alias grams, int;
type Part {
   string name;
   };
type basePart {
   dollars cost;
   grams mass;
   supplier suppliedBy;
   } inherits Part;
type compositePart {
   dollars assemblyCost;
   grams massIncrement;
   useType components;
   } inherits Part;
type useType {
   Part subPart;
   int quantity
   };
```

The schema is based on terminology found in ref. 12. The OOPs+ approach they describe stems from the differentiation between a type declaration, i.e. basePartType – an intentional type, and a collection of objects of that type, basePart – an extensional type. Oggetto supports the concept of an extensional type (i.e. the list of ids returned as the result of a message), but we see no need to use them in this task.

Below we give the test data used in this work.

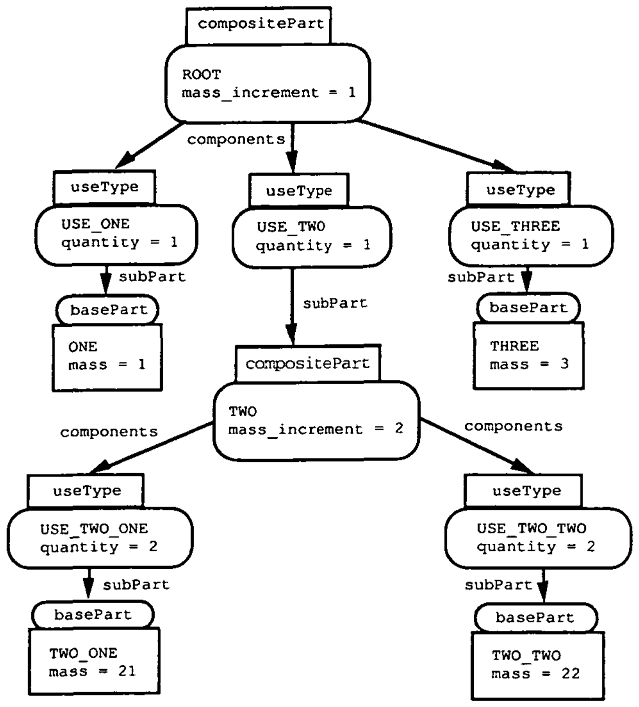

Figure 5. Our test data.

Task 2 is as follows:

```
[
   [ basePart query (cost > 100) and
   (exist suppliedBy) ] print: name, cost,
]; mass
```

Compare this with the OOPS+ query:

```
expensiveParts =
   {
   let parts part;
   foreach part in (parts where cost >
100 and suppliedBy! = {} ) do
   print (part.name, part.cost,
part.mass)
   } ();
```

Note that the OOPS+ query addresses the set of parts, which is the union of the set of base parts and composite parts. However, in the schemas of OOPS+ and Oggetto, composite parts do not possess the attribute cost (as used in the OOPS+ query expression) or mass (as used in the print expression). Therefore, we have made the destination of the query message basePart rather than Part although the latter would still function as expected.

Among the major problems reported in Atkinson and Buneman[2] is the lack of a transitive closure operation, which has to be hand-coded using recursion. This is highlighted by task 3. The OOPS+ solution involves a recursive call. Our solution also uses recursion, but through methods; this offers a more elegant solution. Methods are added to the schema as follows:

```
type basePart (
    method weightOf
      [ self mass ];
    endMethod;
    ) inherits part;
type compositePart (
    method weightOf
      [ list ( [self massIncrement],
        [[[self components] weightOf]
          sum] )
    sum];
    endMethod;
    ) inherits part;
type useType (
    method weightOf
      [ list ( [[self subPart ]
        weightOf],
          [self quantity] ) mult];
    endMethod;
    );
```

Each type involved has a method weightOf. In basePart, it merely returns its own mass. In useType, it finds the total weight of all its subparts and multiplies that by the number of subparts it uses. In compositePart, it finds the weight of all its components and adds its incremental weight. Because we have used the same name throughout, we can apply weightOf to any part of the data structure and obtain an appropriate result. These methods will recurse if need be, i.e. finding the weight of a composite part that contains composite parts.

4.2 The expand operator

Oggetto offers a transitive closure operator, expand. As an example, consider the classic 'hasChild' relation. If we have the following relationships:

```
THOMAS hasChild BRIAN hasChild ANTHONY
```

The expand operator gathers all ids involved in the relationship.

```
expand [ THOMAS hasChild ]
```

results in the group of ids, BRIAN and ANTHONY. Similarly, we can gather all the ids of parts involved in a composite part –

```
expand [[ ROOT components ] subPart ]
```

– where ROOT is the root of the tree making up the part structure. However, owing to the conditions of task 3, we cannot directly obtain the sum of the weights of the parts because of the separation of parts (basePart and compositePart) and the number of parts used (useType). If we change the conditions of the test and eliminate the useType level of the hierarchy by storing n parts where n is the number of parts used in the structure, then the above expand directive returns the ids of parts involved in a composite part, and the number of times they are used will be reflected in their repetition. To illustrate, we give the revised diagram of our test data, with the useType's removed.

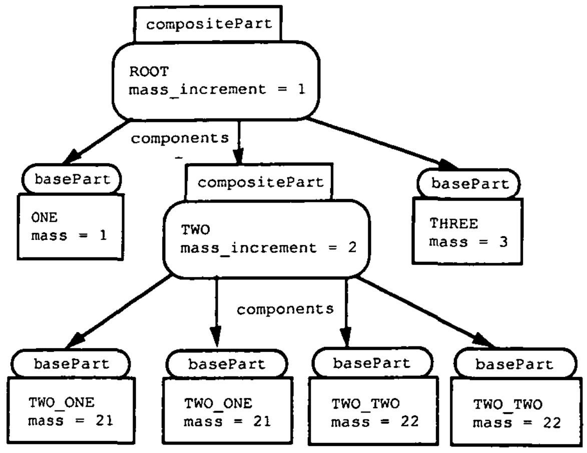

Figure 6. Test data with useType eliminated.

Now we can proceed and use OQL's arithmetic operators to calculate the answers to task 3.

```
ids: = expand [cPartInst components]
  subPart ] ;
totalCost: = [[ids costOf] sum ];
totalMass: = [[ids massOf] sum];
```

While expand will do the transitive closure required, we are not addressing a further aspect of task 3; that the totals should be calculated 'on the run'. This could be partially addressed by an extension to the expand operator:

```
weights: = expand [cPartInst components]
  subPart] with weightOf;
```

This gathers up the weights (integers) as the answer by applying the weightOf operator to each set of new_ids as they are formed. It is a minor alteration to allow multiple withs:

```
costs, masses: = expand [cPartInst
  components] subPart] with weightOf,
  costOf;
totalCost: = [costs sum];
totalMass: = [masses sum];
```

expand currently does not assist with task 3; we must fall back on recursion.

Task 4 deals with the introduction of a new composite part to the database. Below we show the OQL instance declaration required to compose the composite part, TWO.

```
inst compositePartType TWO (
    name: = two;
    massIncrement: = 2;
    components: = (
        inst useType USE_TWO_ONE (
            quantity: = 2;
            subPart: = inst basePartType
                        TWO_ONE (
                name: = two_one;
                mass: = 21
                )
            ),
        inst useType USE_TWO_TWO (
            quantity: = 2;
```

```
          subPart: = inst basePartType
                          TWO_TWO (
              name: = two_two;
              mass: = 22
                  )
              )
          )
    );
```

The system could generate ids; we could replace line 7 of the above example with

```
subPart: = inst basePartType any (
```

The reserved word any indicates the system should provide an 'appropriate' id for the part; this alleviates the need for the user to explicitly name every object they generate.

5. THE UNDERLYING TRIPLE STORE

We have access to three local triple stores implemented using different techniques:

(*a*) A simple heap of triples. This has been used to allow student projects to build interfaces to triple stores without the need for the full DBMS software.

(*b*) Trible: this system uses inverted files to speed up access to triples. The description of a relational version of Trible, Beta, can be found in ref. 15.

(*c*) OSROS:[23] this system uses dynamic hashing on one dimension (subject, relation and object) and two dimensions (subject & relation, relation & object, object & subject) where required to achieve fast access to triples. This system is similar to Asdas[10] in that it uses a dynamic hashing scheme and hashing on combinations of triple fields. Asdas, however, used all possible combinations (right up to subject & relation & object); OSROS tries to be more economical in space by only using the second dimension when required.

The interface is static (Appendix 2) and we can interchange all three versions of the triple store supporting Oggetto. Conversely, if any triple store possesses this interface we can port Oggetto to that store.

5.1 Comments on storage

Rather than store objects by layering within other database models, there are more basic possibilities for the storage of objects. A simple option might be to think of an object as a page on the backing store; the page is laid out as might be appropriate for the type of object it is an instance of; here, we are approaching the idea of a 'frame' stored on backing store. However, this presents the early binding problem. It is difficult to add a new attribute to an object. Oggetto 'inherits' the late binding advantages of the binary relational model; all we do is add a new triple.

Moreover, OSROS uses multi-dimensional extendible hashing, supporting set-at-a-time retrieval. The new triple is guaranteed to be stored in the same bucket as the other attributes of an object; retrieving all the attributes of an object will retrieve the new extended set of attributes with no extra cost (unless bucket overflow has occurred).

5.2 Internal storage

For convenience and efficiency, in-store object information is held in a data structure which stores the name and type of every attribute in the type. The same structure is also used to hold instances of a type by using an additional field, values. Instances of a type are built up internally before being converted into a set of triples for external storage. The internal structure is built up dynamically when required.

6. FUTURE WORK

6.1 Introduction

Oggetto, as outlined in this paper, is a prototype; it is a vehicle for investigation into the suitability of triple stores to support object stores, and for the design of a novel object-oriented query language. In this section, we outline some of the areas we feel worthy of further attention.

6.2 Objects with more than one type

Clearly, an entity may be associated with more than one type. For example, DAVID can be a person and also a parent, etc. Oggetto only allows an entity to be of a single type. The system needs to be extended to allow entities to be declared independent of type, and then freely associated and dissociated with several types (as described in ref. 4).

6.3 Version control

Version control is important in all data repositories. It becomes more intriguing when combined with the schema evolution supported by OODBs. Not only do we have to maintain versions of data, but also of metadata. We have to consider how previous versions of the data can be viewed by the current metadata, and how current data is viewed by previous versions of the metadata.[22]

The particular interest in our layered approach is to add versions to the triple store (none of our current stores support versions) with appropriate interfaces, and then investigate how this affects the object store. We hope to show that this emphasises the correctness of our approach.

7. CONCLUSIONS

In this paper we have described an object store layered on a triple store, and showed the suitability of triple stores for this function. Moreover, one particular triple store used (OSROS) has underlying features that make it particularly suitable for the support of an object base. It is possible to view the Oggetto system as a high-level interface to the triple store.

In Senko[19] he recommends that data description languages should consider forms of binary association as their primitive elements to achieve a more representation independent and more stable information level. If we view the Oggetto model and language as matching the DIAM end-user level then our binary relations match the information level, with the various binary relational storage structures available to Oggetto as matching the data logical levels.

This work, like others, has tackled the implementation of an OODB through layering over 'conventional' programming languages and/or databases. We hope we have shown how our particular layered approach is powerful in that

* the underlying binary-relational model is capable of modelling any set of relationships, no matter how complex;
* the suitability of the model in supporting the kinds of operations expected in an OODB, i.e. schema evolution. Schema evolution as described in this paper is static, in that we cannot dynamically form new sets through some kind of data analysis, i.e. clustering. These issues are addressed in refs 8, 13 and partially applied to OODBs in ref. 18. The BRM is well suited for this form of manipulation, as reported in ref. 7. This is an area which will be followed up in Oggetto;
* the addition of an active element in the form of UTM (or equivalent) gives us an expressive power equivalent to Prolog;
* the use of a suitable (and well known) backing store access technique (dynamic hashing) lends efficiency to the retrieval of objects and their attributes.

Acknowledgements

Thanks are due to Dr David Hutchison, Steve Elliott, Peter Hurley and the referees for their helpful comments on earlier drafts.

REFERENCES

1. J. R. Abrial, Data semantics. In *Data Base Management – Proceedings IFIP Working Conference on Data Base Management*, edited J. W. Klimbie and K. L. Koffeman, pp. 1–59. North-Holland Publishing Company (1974).
2. M. P. Atkinson and O. P. Buneman, Types and persistence in database programming languages. *ACM Computing Surveys* **19** (2), 105–190 (1987).
3. J. Banerjee *et al.*, Data model issues for object-oriented applications. *ACM Trans. on Office Information Systems* **5** (1), 3–26 (1987).
4. D. Beech, Groundwork for an object database model. In *Research Directions in Object-Oriented Programming*, edited B. Shriver and P. Wegner, pp. 317–354. MIT Press (1987).
5. P. Chen, The entity-relationship model: towards a unified view of data. *ACM Trans. on Database Systems* **1** (1), 9–36 (1976).
6. C. J. Date, *A Guide to the SQL Standard.* Addison-Wesley (1987).
7. Beshir E. M. A. Elgalal, Minimally-redundant data structures and reasonable hypothesis: some general heuristic methods of knowledge processing. Ph.D. thesis, University of Strathclyde (1985).
8. D. H. Fisher, Knowledge acquisition via incremental conceptual clustering. *Machine Learning* **2**, 139–172 (1987).
9. D. H. Fishman *et al.*, Iris: an object-oriented database management system. *ACM Trans. on Office Information System* **5** (1), 48–69 (1987).
10. R. A. Frost, *Asdas – a simple database system aimed at the naive user, Proceedings of 6th ACM European Regional Conference on Systems Architecture*, pp. 234–240. IPC Business Press Ltd, London (1981).
11. R. A. Frost, Binary-relational storage structures. *Computer Journal* **25** (3), 358–367 (1982).
12. E. Laenens and D. Vermeir, *An Overview of OOPS+, an Object-Oriented Database Programming Language*, pp. 350–373, ECOOP (1988).
13. M. Lebowitz, Experiments with incremental concept formation: UNIMEM. *Machine Learning* **2**, 103–138 (1987).
14. D. Maier *et al.*, *Development of an Object-Oriented DBMS*, pp. 472–482. OOPSLA '86 (1986).
15. J. A. Mariani, Implementation of a general purpose database package. In *Implementation of Small Computer Systems*, edited D. Whiddett, pp. 39–58. Ellis Horwood (1989).
16. J. A. Mariani, Object oriented database systems. In *Object-Oriented Languages, Systems and Applications*, ch. 7, edited G. S. Blair *et al.* Pitman (1990).
17. D. R. McGregor and J. R. Malone, The fact database system. In *Research and Development in Information Retrieval*, edited C. J. Von Rijsbergen and P. W. Williams, pp. 203–217. Butterworth (1981).
18. G. T. Nguyen and D. Rieu, Schema evolution in object-oriented database systems. *Data & Knowledge Engineering* (4), 43–67 (1989).
19. M. E. Senko, The DDL in the context of a multilevel structured description: DIAM II with FORAL. In *Data Base Description*, edited B. C. M. Douque and G. M. Nijssen, pp. 239–258. North-Holland Publishing Company (1975).
20. M. E. Senko, Data structures and data accessing in data base systems past, present, future. *IBM System Journal* **3**, 208–257 (1977).
21. G. C. H. Sharman and N. Winterbottom, *The Universal Triple Machine: a Reduced Instruction Set Repository Manager, Proceedings of BNCOD* **6**, 189–214 (1988).
22. A. H. Skarra and S. B. Zdonik, *The Management of Changing Types in an Object-oriented Database, ACM OOPSLA 86 Proceedings*, pp. 483–495 (1986).
23. C. Snape, OSROS: a binary relational database system. B.Sc. Project Internal Report, Computing Dept., Lancaster University (1986).
24. M. Stonebraker and L. A. Rowe, *The Design of Postgres, Proceedings of ACM SIGMOD International Conference on the Management of Data*, pp. 340–355 (1986).
25. R. van de Riet, MOKUM: an object-oriented active knowledge base system. *Data & Knowledge Engineering* **4**, 21–42 (1989).

APPENDIX 1: MORE SCHEMA MANIPULATIONS IN OGGETTO

A 1.1 Change the name of a class

syntax:`schema change type name oldTypeName to newTypeName`

semantics: We build up an internal structure, `type`, which holds all the information about the type. This includes the name of the type, attributes, attribute types, and inheritance. We use the internal structure (passing it to a procedure `destroy_type`) to generate the appropriate sequence of 'delete triple' commands; we delete all the schema triples associated with the `oldTypeName`.

We change the name of the type held within the internal structure to `newTypeName`, and pass the structure to a procedure `store_type`. This procedure now generates the appropriate schema triples.

We must also consider the inheritance triples. We retrieve all triples of the form

```
(inheritingType, 'inherits', oldTypeName)
(inheritingType→oldTypeName,
'inOrder', order)
```

and replace them by

```
(inheritingType, 'inherits', newTypeName)
(inheritingType→newTypeName, 'inOrder',
order)
```

The last thing we must do is to change the type name associated with all instances of the `oldTypeName`. We retrieve all triples of the form

```
(ANY_STRING, 'hasType', oldTypeName)
```

and replace them with triples of the form

```
(id, 'hasType', newTypeName).
```

A 1.2 Add a new attribute to a class

```
Syntax: schema add attribute attributeName
  with type attributeType to type
```

semantics: attribute and attribute type information is stored in a set of schema triples as follows:

```
(type, hasAttribute, attributeName)
(type, attributeName, attributeType)
```

The ordering of attributes within a single type is unimportant. To add a new attribute, we add two schema triples of the above format. For example, the command

```
add attribute religion with type string
  to person
```

adds the two new schema triples:

```
(person hasAttribute religion)
(person religion string)
```

Because of the BRM's capacity for handling 'null' values elegantly, we can leave the addition of new data triples to some later time (possibly never). When viewing an instance, Oggetto will only display those recorded values.

A 1.3 Drop an existing attribute from a class

```
syntax:schemadeleteattributeattributeName
  from type
```

semantics: we remove the two schema triples of the format given in A 1.2. Currently, instances of the effected type remain unaltered, although we could delete all instances of the attribute within the type and its subtypes.

A 1.4 Change the name of an attribute from a class

```
syntax: schema change attribute name
    oldAttributeName in type to
    newAttributeName
```

semantics: The schema triple

```
(type, oldAttributeName, attributeType)
```

is replaced with

```
(type, newAttributeName, attributeType)
```

All instances of the type and its subtypes are identified; attributes of an instance are stored as instance triples in the form

```
(instance, attributeName, value).
```

Every such attribute instance is retrieved and replaced by triples in the form

```
(instance, newAttributeName, value).
```

A 1.5 Change the domain of an attribute of a class

```
syntax: schema change attribute domain
  attributeName in type to newDomain
```

semantics: the schema triple

```
(type, attributeName, oldDomain)
```

is replaced by the new schema triple

```
(type, attributeName, newDomain).
```

All instances of the type and its subtypes are identified; every attribute instance affected are deleted.

A 1.6 Make a class S a superclass of a class C

```
syntax: schema add superType to type
```

semantics: the relation of type to `supertype` is stored as two triples.

```
(type inherits superType)
```

indicates the basic relation and

```
(type→superType inOrder numericOrder)
```

this secondary triple codifies the `inherits` relation within a single value using the → symbol to represent the relation. This allows us to store information about that relation; here, the order in which the inheritance occurs.

When we add a new supertype it is automatically placed at the end of the ordering. However, we can change this ordering later (see A 1.8).

A 1.7 Remove a class S from the superclass list of a class C

```
syntax: schema remove superType from type
```

semantics: we have to alter the two triples as stored for A 1.6 but must also update the ordering information. We retrieve the appropriate

```
(type→superType inOrder numericOrder)
```

triple, retrieve the subsequent triples in the sequence and delete them, restore the subsequent triples with a modified numeric order and lastly delete the base relation triple.

A 1.8 Change the order of superclasses of a class C

```
syntax: schema change superType in type
  order numeric
```

semantics: This is not currently handled in an optimum fashion. The ordering information is brought into a main store list, deleted from the triple store, the main store list is manipulated as appropriate, and lastly the altered main store information is returned to the triple store as a set of order information triples.

APPENDIX 2: THE SUBROUTINE INTERFACE TO THE TRIPLE STORE

```
void insert_triple (char s[], char r[],
  char o[])
```

Takes three strings as parameters; treats them as the subject, relation and object of a triple, and adds them to the triple store.

```
int query_triple (char s[], char r[],
  char o[])
```

Takes three strings as parameters; treats them as the subject, relation and object of a triple template. Templates have three fields; they can contain any string value or a special wildcard value. `query_triple` compares the template with all stored triples and remembers the location of any triples that matched the template. It returns a 'Boolean' which is TRUE if any triples matched and FALSE otherwise.

```
int query_triple_num (char s[], char r[],
  char o[])
```

Similar to the above, but it returns the number of triples matched.

```
int retrieve_triple (char s[], char r[],
  char o[])
```

After a `query_triple` has been issued, `retrieve_triple` is used to sequentially return the matched triples. It returns them in the three string parameters. It also returns the 'Boolean' result TRUE if there are still triples left to be returned, FALSE if the list of hit triples is exhausted. `Query_triple` and `retrieve_triple` are normally used in tandem as follows:

```
query_triple(a,b,c)
while (retrieve_triple(d,e,f) == TRUE)
    {
    /* process triple*/
    };
```

```
int open_db (char dbname[])
```

This takes as parameter the name of a triple store and attempts to open it. It returns TRUE if successful, FALSE otherwise.

```
void create_db (char dbname[])
```

This takes as parameter the name of a triple store and attempts to create it.

```
void close_db ()
```

This closes the current triple store.

```
void delete_triples (char s[], char r[],
  char o[])
```

Like `query_triple`, this takes a triple template as parameter, and locates all triples that match the template. Then, however, it goes on to delete all the matching triples.

Book Review

K. Devlin
Logic and Information
Cambridge University Press. £17.95
ISBN 0 521 41030 4.

This book starts well, with a clear statement that there is a need for a science of information. Moreover, the author in his acknowledgements makes the valid point that efforts to satisfy this need must initially be driven by scientific curiosity without necessarily an explicit utilitarian objective. These are propositions that need to be published widely to stimulate academic workers in the information engineering field to recognise that the conceptual foundations of systems engineering are conspicuous by their absence, essentially because there is no science of information but there ought to be and there needs to be such a science.

However, the author then assumes that mathematics is the master science so that he devotes the rest of his book to a determined attempt to formulate concepts such as 'Infons', 'Situations' and 'Constraints', and symbols to represent them, as a first step in the foundation of a mathematics of information. Perhaps in a future book the author will use his symbols to derive propositions whose validity can be checked by experiment, but if there are such propositions in the present book they are well hidden. Curiously there is a clear statement that the concepts proposed are recursively defined, but no mention of observed hyperbolic statistical distribution of symbols in meaningful text that could be interpreted as offering experimental support for the recursive definitions. Indeed, this reviewer could find no reference at all to repeatable observations, which surely should appear in the foundations of a proposed new branch of science.

To force information into a mathematical mould the author had endeavoured to extend the scope of mathematics and logic to include intuition and judgement. Certainly these subjective techniques play an important role in the use of information by all people including mathematicians, but it is generally understood that they are consciously excluded from published mathematical work, indeed it is the objectivity of mathematical techniques derived from the conscious exclusion of intuitive judgements that accounts for the utility of mathematics in so many branches of established science. It is therefore far from obvious that the author's declared interpretation of a 'science of information' as a 'mathematics of information' is valid with the generally accepted interpretation of the words.

Many readers for whom mathematics is a useful tool but not a way of life will find this book difficult to follow, so that it will probably have less impact than it deserves. This is unfortunate, since the book breaks new ground in a field that is likely to become of increasing importance. The book can be recommended to anyone who has recognised the need for a better understanding of the nature of information and who is prepared to put effort into understanding Devlin's presentation. Perhaps there is gold in this book but it is not clear to this reviewer. Devlin promises more volumes – certainly he should be encouraged to do more work on the problem that he has recognised so clearly, but his objective should be to formulate some novel and checkable conclusions and to distil the essence of his work into a smaller volume.

G. Scarrott
Welwyn