ODE (Object Database and Environment): The Language and the Data Model

R. Agrawal
N. H. Gehani

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ODE is a database system and environment based on the object paradigm. It offers one integrated data model for both database and general purpose manipulation. The database is defined, queried and manipulated in the database programming language O++ which is based on C++. O++ borrows and extends the object definition facility of C++, called the class. Classes support data encapsulation and multiple inheritance. We provide facilities for creating persistent and versioned objects, defining sets, and iterating over sets and clusters of persistent objects. We also provide facilities to associate constraints and triggers with objects. This paper presents the linguistic facilities provided in O++ and the data model it supports.

1. INTRODUCTION

The object paradigm is a natural way of organizing data as it allows users to structure, retrieve and update data in terms of the application domain. ODE is a database system and environment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++ (it was called O in earlier versions of this paper). O++ is based on C++ [52]; it borrows and extends the object definition model of C++. This paper is an introduction to the linguistic facilities provided in O++ and the data model it supports.

Although conventional programming language objects and database objects are similar in that they encapsulate object properties, there are several differences. For example, database objects persist beyond the lifetime of the program creating them. Many database applications, such as computer-aided design and software management, require the capability to create and access multiple versions of an object [9, 24, 31, 53]. Object versions are also important for historical databases, such as those used in accounting, legal, and financial applications, that must access the past states of the database [20, 45]. Support for active databases, such as those used in computer integrated manufacturing, power distribution network management, and air-traffic control, requires the capability to attach with objects conditions and actions that are triggered when the conditions are satisfied [21, 51]. Finally, the ability to associate constraints with objects is necessary to ensure database integrity [37]. Programming languages typically do not support persistent objects (with the exception of files) or multiple object versions, nor they do provide facilities for associating constraints and triggers with objects. Consequently, if a conventional programming language is to be used as a database programming language, it must be augmented with facilities that support the needs of database systems.

O++ provides one integrated model for both database and general purpose manipulation. We use the C++ object model, called the *class*, as the basis for the object model of O++. The class facility supports data encapsulation and multiple inheritance. O++ extends C++ classes by providing facilities for creating and manipulating persistent objects and their versions, and associating constraints and triggers with objects. A major criticism of the current object-oriented databases and languages is that they lack the capability to pose arbitrary "join" queries, and that query processing "smells" of pointer chasing as in CODASYL database systems [36]. O++ alleviates these problems by providing iterators that allow sets of objects to be manipulated almost as declaratively as the database query languages based on relational calculus. The set iteration facility of O++ also allows the expression of recursive queries [2, 10], a major concern in deductive databases. The iterators can be qualified with clauses that specify iteration subsets and order, which can be used to advantage in query optimization. Many of the O++ facilities can be found in other languages and systems. Our major contribution lies in providing a clean fusion of the advances in both database and programming language research within an object-oriented framework.

Virtues of object-oriented database systems have been extolled elsewhere (see, for example, [12, 23, 25-27, 29, 30, 33-35]). O++ shares with these systems the goals of providing a rich type system to model complex and composite objects. It provides encapsulation to hide implementation details, supports multiple inheritance for organizing objects in taxonomies in which the more specialized objects inherit the data and functions of more generalized objects, separates type definition from type instantiation, allows explicit specification of relationships between objects, and supports object identities that allows persistent database objects to have an existence independent of their values. Some extensible database projects, such as [13, 18, 40, 45, 46, 49] also have similar goals. O++ is in the same spirit as the work done in designing database programming languages, such as [7, 11, 20, 38, 44, 47, 48, 50, 54]; it strives to be the single language for data definition, data manipulation and general computation to avoid the problems arising out of "impedance mismatch" [20]. O++ also shares the concerns of the persistent programming languages, such as [6, 17, 39, 41-43]; persistence is a property of object instances and not types, and persistent objects are accessed and manipulated in much the same way as volatile objects. O++ is related to the language E [42, 43] in that O++, like E, also uses the C++ object model

and adds persistence to it. Vbase [7] and O2 [33] also seek to blend an object-oriented data model with C.

In this paper, we concentrate on the data modeling and the query processing aspects of O++. We do not cover concurrency issues and some systems-oriented features such as schema evolution, authorization, security, etc. We also do not discuss the environment and the implementation strategies. These issues will be discussed in future papers. We have tried to be precise with the language constructs, but omitted details when they are obvious. Although we will occasionally refer to transactions, we do not discuss them in this paper. For the purpose of this paper, any O++ program that interacts with the database will be considered to be a single transaction. The rationale behind the design of O++ is discussed in [3].

The organization of the rest of the paper is as follows. Section 2 describes the data structuring facilities provided in O++. Section 2.1 gives an overview of the object definition facility, Section 2.2 discusses the inheritance model, Section 2.3 presents the persistence model, Section 2.4 discusses how persistent objects of the same type are grouped together in a cluster, and Section 2.5 describes the set data type. Section 2.6 discusses persistence in some related C-based languages/systems: E, Vbase, and O2. Section 3 presents the query processing facilities. Section 4 presents our versioning model, and Sections 5 and 6 discuss the facilities for associating constraints and triggers respectively with objects. Our conclusions are presented in Section 7.

2. DATA STRUCTURING CONSTRUCTS

A database is a collection of persistent objects, each identified by a *unique* identifier, called the object identifier (id) that is its *identity* [32]. We shall also refer to this object id as a *pointer to a persistent object*. We visualize memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. They are allocated on the program stack or on the heap and their lifetime is bounded by the life of the program. *Persistent* objects are allocated in persistent store and they continue to exist after the program that created them has terminated. Interaction with these objects is routed through an object manager, but this is hidden from the programmer.

2.1 Object Definitions: C++ Classes

Class declarations consist of two parts: a specification (type) and a body. The *specification* represents the class "user interface". It contains all the information necessary for the user of a class, and also for the compiler to allocate class objects. The *body* consists of the bodies of functions declared in the class specification but whose bodies were not given there.

Class specifications have the form

```
class name {
    private components
public:
    public components
};
```

The *private* components of a class are data items and functions that implement class objects. These represent internal details of the class and cannot be accessed by the user of a class. The *public* class components can be data items, constructors, destructors, member functions (operators), and friend functions (operators). The public components, which represent the class user interface, are the components that the user of a class can reference. Constructors are functions that are called automatically to construct a class value. Member and friend functions are used to manipulate class objects. Destructors are functions that are called automatically when a class object is explicitly deleted or when its scope is left.

We illustrate the use of the class facility by defining a class `item`. Here is its specification:

```
class item {
  Name nm;
  double wt; /* in kg */
public:
  item(Name xname, double xwt);
  Name name();
  double weight_lbs();
  double weight_kg();
};
```

The private part of the specification of class `item` consists of the declarations of two variables: `nm` and `wt`. The public part consists of a constructor function `item` (has the same name as the class name) and three member functions `name`, `weight_lbs` and `weight_kg`.

Here are the bodies of the member functions:

```
item::item(Name xname, double xwt)
{
  nm = xname;
  wt = xwt;
}
Name item::name()
{
 return nm;
}
double item::weight_kg()
{
  return wt;
}
double item::weight_lbs()
{
  return (wt * 2.205);
}
```

2.2 Inheritance

Inheritance allows objects to be organized in taxonomies in which the more specialized objects inherit properties, i.e., the data and functions, of more generalized objects. Similar objects with a few different properties can be modeled by specifying a common part, called the base (super) class, for the common properties and then deriving specialized classes from this base class. Derived classes can be used to construct heterogeneous data structures such as lists with different types of elements because a pointer to a class can point to any object whose type is derived from this class.

A derived class is specified by following its name with the name of the base class. A derived class inherits the data items as well as the member functions of the base class. As an example, consider the following class `stockitem` that is derived from class `item`:

```
class stockitem: public item {
 int consumption;/*qty consumed per year*/
 int leadtime;   /*lead time in days*/
public:
 int qty;
 double price;
 stockitem(Name iname,double iwt,int xqty,
           int xconsumption,
           double xprice,int xleadtime);
 int eoq();      /*economic order quantity*/
};
```

`stockitem` is the same as `item` except that it contains other information such as the quantity in stock, its consumption per year, its price and the lead time necessary to restock the item. Also, `stockitem` has its own constructor function and additional member functions.

2.2.1 Multiple Inheritance Multiple inheritance allows a new class to be derived from multiple classes. For example, class `stockitem` can be derived from the two classes `item` and `supplier`, as a result of which `stockitem` will have the properties of both `item` and `supplier`. First, here is the specification of class `supplier`:

```
class supplier {
  Name nm;
  Addr addr;
public:
  supplier(Name xname, Addr xaddr);
  Name name();
  Addr address();
};
```

Here is the new specification of class `stockitem`:

```
class stockitem: public item, public supplier {
  int consumption;
  int leadtime;
public:
  int qty;
  double price;
  stockitem(Name iname,double iwt,int xqty,
      int xconsumption, double xprice,
      int xleadtime,Name sname, Addr saddr);
  int eoq();  /*economic order quantity*/
  int reorder_level();
  Name itemname();
  Name suppliername();
  Name name();
};
```

Note that each of the base classes `item` and `supplier` has a member function named `name`. Ambiguities are resolved by using explicit qualification as shown below, for example, in the bodies of the member functions `itemname` and `suppliername`:

```
Name stockitem::itemname()
    { return item::name(); }
Name stockitem::suppliername()
    { return supplier::name(); }
```

If an application requires that the `name` member function for `stockitem` yield the `supplier`'s name, then `name` can be redefined as follows:

```
Name stockitem::name()
    {return supplier::name();}
```

2.3 Persistent Objects

When incorporating persistence in O++, we kept the following principles in perspective:

- Persistence should be orthogonal to type [8]. Persistence should be a property of object instances and not types. It should be possible to allocate objects of any type in either volatile or persistent store.
- There should be no run-time penalty for code that does not deal with persistent objects.
- Allocation and manipulation of persistent objects should be similar to the manipulation of volatile objects. For example, it should be possible to move objects from persistent store to volatile store and vice versa in much the same way as it is possible to move objects from the stack to the heap and vice versa.
- Inadvertent fabrication of object identities should be prevented.
- Language changes should be kept to a minimum.

Persistent objects are referenced using pointers to persistent objects (that is, their identities); these pointers can be allocated in either the heap or the stack but persistent objects must themselves be allocated in persistent store. Persistent objects are allocated and deallocated in a manner similar to heap objects. Persistent storage operators `pnew` and `pdelete` are used instead of the heap operators `new` and `delete`. If successful, `pnew` returns a pointer to the persistent object created by it; otherwise, it returns the null pointer. Here is an example:[1]

```
persistent stockitem *psip;
...
psip = pnew stockitem(initial values);
```

Note that `psip` is pointer to a persistent `stockitem` object, and *not* a persistent pointer to a `stockitem` object. Thus, syntactically the keyword `persistent` is a type qualifier (like `const` and `volatile` in ANSI C[1]) and not a storage specifier, even though it refers to a type of storage. Also, `psip` is allocated on stack (not in the persistent store), but `pnew` allocates the `stockitem` object in persistent store and its id (returned by `pnew`) is saved in `psip`.

Persistent objects can be copied to volatile objects and vice versa using simple assignments:

```
*sip = *psip; ... *psip = *sip;
```

Components of persistent objects are referenced like the components of volatile objects, e.g.,

```
w = psip->weight_kg();
```

A persistent object can be deleted with the `pdelete` operator, e.g.,

```
pdelete psip;
```

deletes the object pointed to by `psip`.

2.4 Dual Pointers

Pointers to persistent objects always refer to persistent objects and ordinary pointers to volatile objects. Consequently, it is not possible to write a function that accepts as an argument a pointer to either volatile or persistent object. To allow writing of such functions, we introduce the notion of a dual pointer:

1. Creation of a persistent object is syntactically similar to the creation of a volatile object:
```
stockitem *sip;
...
sip = new stockitem(initial values);
```

one that can refer to a volatile or persistent object. Whether the object referenced is volatile or persistent is determined at run time. Pointers to volatile and persistent objects can be assigned to dual pointers. But dual pointers that refer to volatile objects can only be assigned to pointers to volatile objects and dual pointers that refer to persistent objects only to pointers to persistent objects. Here is an example illustrating the use of dual pointers:

```
class node {
    ...
    dual node *next;
public:
    ...
    dual node *add(dual node *n);
};

persistent node *p, *proot;
        /*proot: list in persistent store*/
node *v, *vroot;
        /*vroot: list in volatile store*/
...
proot = proot->add(p);
vroot = vroot->add(v);
```

2.5 Clusters of Persistent Objects

All persistent objects of the same type are grouped together into a *cluster*; the name of a cluster is the same as that of the corresponding type, that is, clusters are type extents [16]. Before creating a persistent object, the corresponding cluster must exist; it is *created* by invoking the `create` macro[2] — in this program or in another program:

```
int create(type-name);
```

type-name is the name of the cluster being created.[3]

A cluster can be destroyed by invoking the `destroy` macro:

```
int destroy(type-name);
```

type-name is the name of the cluster being removed. Note that all the objects in the cluster are destroyed. `destroy` returns 1 if successful; otherwise, it returns 0.

2.5.1 Subclusters Sometimes it may be desirable, for logical organization or efficiency reasons, to group objects in a cluster into subclusters so that they can be collectively referenced. For example, subclusters can be used to partition a `student` cluster into groups corresponding to the dorms in which the students live.[4] Subclusters are also created and destroyed with the `create` and `destroy` macros. A subcluster name is a string qualified by the corresponding cluster (object type) name, e.g.,

```
create(student::"Cascadilla Hall");
```

Objects are specified to belong to a specific subcluster when they are allocated in persistent store, e.g.,

```
sp = pnew student("Jack")::"Cascadilla Hall";
```

2.6 Sets

O++ supports multisets (sets whose elements do not have to be unique). Sets are a convenient mechanism for manipulating a collection of objects. Set declarations are similar to array declarations but, unlike arrays, set subscripting is not allowed. Here are some examples:

```
persistent part *partset<MAX>;
real s<100>;
```

When declaring set parameters, the maximum number of elements in a set need not be specified:

```
userid users<>;
```

The set operations supported are assignment, union, difference, insertion, and deletion. Set elements are accessed by using the set iteration facility (discussed later).

2.7 Related C-based Approaches

The database implementation language E [42,43] also started with C++ and added persistence to it. In E, persistent objects must be of special types called "db" types. Objects of such types can be volatile or persistent. Persistent objects can be allocated statically by using the storage class `persistent` in object definitions or they can be allocated dynamically by using the predefined db class `file`. Other features of E include generic classes and iterators.

Avalon/C++ [22], also provides facilities for persistence. Types whose objects are to be allocated in persistent store must be *derived* from the class `recoverable`. A persistent object is accessed by explicitly bringing it to volatile store from persistent store (called "pinning" the object), accessing the object, and then moving the object back to persistent store (called "unpinning" the object). The pinning and unpinning operations are provided by the class `recoverable`.

Vbase [7] combines an object model with C. It presents to the user two languages: the type definition language TDL for specifying classes and operations, and the C superset COP for writing methods to implement the operations. The O2 system [33] also integrates an object model with C. Type definitions are written in one language and methods are written in the C superset CO2. A class in O2 implicitly owns a persistent collection of objects of the class.

3. QUERY PROCESSING CONSTRUCTS

One misgiving researchers have with the current object-oriented databases is that they may take us back to the days of CODASYL databases in which data is accessed by "using pointers to navigate through the database" [36]. In object-oriented databases, the pointers are the object ids. By introducing clusters, sets, and high-level iteration facilities for accessing objects in clusters and sets, O++ provides an alternative to using object ids to navigate through the database. Another criticism of object-oriented databases is that they lack the capability to express arbitrary "join" queries [36]. In O++, arbitrary joins can be expressed by iterating over the clusters to be joined and the join conditions can be specified as the iteration conditions.

3.1 Iterating Over Sets and Clusters

Values of the elements of a set, a cluster or a subcluster can be accessed with a `for` loop of the form[5]

2. We intend to extend the `create` macro to allow the programmer to give additional information to assist the object manager in implementing efficient accesses to objects.
3. The ODE environment can be queried about the clusters that exist in persistent store. In case of classes, it can also be asked to display the specification of the objects in the cluster.
4. The type inheritance mechanism can also be used to simulate the effect of subclusters but this may require the creation of a large number of types, e.g., one type for each dorm.

for *i* in *set-or-cluster-or-subcluster*
[*suchthat-clause*] [*by-clause*] *statement*

The loop body is executed once for each element of the specified set, the cluster or the subcluster; the loop variable *i* is assigned the element values in turn. The type of *i* must be the same as the element type.

The *suchthat-clause* has the form

suchthat (e_{st})

and the *by-clause* has the form

by (e_{by} [, *cmp*])

The suchthat and by expressions e_{st} and e_{by} must contain the loop variable *i*. If the suchthat and by expressions are omitted, then the for loop iterates over all the elements of the specified grouping in some implementation-dependent order. The suchthat clause ensures that iteration is performed only for objects satisfying expression e_{st}. If the by clause is given, then the iteration is performed in the order of non-decreasing values of the expression e_{by}. If the by clause has only one parameter, then e_{by} must be an arithmetic expression. If the by clause has two parameters, then the second parameter *cmp* must be a pointer to a function that compares two elements of type *t*, where *t* is the type of the by expression e_{by}. *cmp* compares its arguments and returns an integer greater than, equal to, or less than 0, depending upon whether its first argument is greater than, equal to, or less than its second argument.

As an example, here is a program segment that illustrates iteration over clusters and sets. It prints the name of people along with the names of their children whose profession is computer science (CS):

```
class person {
public:
  Name name;
  char sex;
  Name Profession;
  persistent person *children<>;
};
...
for p in person
  for c in p->children
       suchthat(c->Profession == Name("CS"))
    printf("%s %s\n", p->name, c->name);
```

Although the suchthat and by clauses can be simulated by using an *if* statement within the loop body and by sorting the set of values in volatile memory, these clauses facilitate optimization, similar to those performed in relational database systems, as follows: we expect to pass these clauses to the object manager to select only the desired object ids and deliver them in the right order for the for loop.

3.1.1 Iterating Over Cluster Hierarchies Clusters mirror the hierarchy relationship of the corresponding types. If type *x* is derived from type *y*, then the corresponding clusters also have the same relationship. It is sometimes necessary to collectively access objects in a cluster and those in related "derived" clusters.[6] This can be done with the forall loop which has the form

forall *oid* in *cluster* [*suchthat-clause*] [*by-clause*] *statement*

Except for the inclusion of objects in derived clusters, the semantics of the forall loop are the same as those of the for loop for iterating over a cluster. Thus, given the class item and the derived class stockitem as defined in Section 2, the statement

```
for ip in item
  tot_wt += ip->weight_kg;
```

computes the weight of only objects of type item, but the statement

```
forall ip in item
  tot_wt += ip->weight_kg;
```

computes the weight of all items including stockitems.

3.1.2 Arbitrary Joins The for and forall loops can have multiple loop variables:[7]

for i_1 in *set-or-cluster-or-subcluster*$_1$, ...,
i_n in *set-or-cluster-or-subcluster*$_n$
[suchthat (e_{st})] [by (e_{by})] *statement*

forall oid_1 in *cluster*$_1$, ..., oid_n in *cluster*$_n$
[suchthat (e_{st})] [by (e_{by})] *statement*

The loop body will be executed for every combination of values for the loop variables that satisfy the suchthat expression and in the order specified by the by clause.

These loops allow the expression of operations with functionality of the arbitrary relational join operation. For example, we can write

```
for e in employee, d in dept
    suchthat(e->dno == d->dno)
  printf("%s %s\n", e->name, d->name);
```

to print the name of the employee from the employee object and the name of the department from the dept object.

3.1.3 Type Test The object type can be determined with the is operator which has the form

e is *type*

This expression evaluates to true if expression *e* is of the specified type, and to false otherwise. Note that if *e* is of a dual pointer type, then the is operator will return true if *type* is the dual pointer type, or either the corresponding ordinary pointer type or the corresponding persistent pointer type depending upon whether *e* refers to a volatile object or a persistent object.

Suppose that we want to compute and print the average income of university employees and, separately, that for faculty and students in a university database (employees can be other than faculty and students, e.g., staff). Class person has the member function income, and classes student and faculty have been derived from it. Here is the code to perform the above computation:

5. The square brackets [and] indicate an optional item. The suchthat and by clauses are based on similar clauses in SQL [19] and Concurrent C [28]. Similar for loops have been provided, among others, in Pascal/R [48], Rigel [44], Plain [54] and Trellis/Owl [39] and Vbase [7].

6. POSTQUEL [45] allows a * to be specified after the relation name to retrieve tuples from the named relation and all relations that inherit attributes from it. Orion [12] also provides similar functionality.

7. Rigel [44] also allows multiple loop variables in its for loop.

```
np = ns = nf = incomep = incomes = incomef = 0;
forall p in person {
  incomep += p->income(); np++;
  if (p is persistent student *)
    { incomes += p->income(); ns++;}
  else if (p is persistent faculty *)
    { incomef += p->income(); nf++;}
}
```

3.2 Fixpoint Queries

Aho and Ullman [5] have shown that the least fixpoint operator is an essential addition to the relational query languages, and considerable research has been devoted to developing notations for expressing least fixpoint queries and designing algorithms for evaluating them (see, for example, [2, 10]). When iterating over a set or a cluster, we allow iteration to also be performed over the elements that are added during the iteration, which allows the expression of recursive queries [5]. Thus, given the class `person` as above, the following statements finds all the descendents of `abraham`:

```
for p in person
    suchthat(p->name == Name("abraham"))
  descendants=p->children;/*recursion basis*/
for d in descendants
  descendants += d->children;    /*recursion*/
```

Modifying a set or a cluster while iterating over it makes the by clause inoperative because such modification is likely to destroy the ordering. Deleting an element from the set or cluster means that the loop will not iterate over this element provided such an iteration has not already taken place or is not currently taking place.

4. VERSIONING

Many database applications, such as computer-aided design and software management, require the capability to create and access *multiple versions* of an object [9, 24, 31, 53]. Object versions are also important for historical databases, such as those used in accounting, legal, and financial applications that require accesses to the past states of the database [20, 45].

In ODE, all persistent objects can have versions and there is no pre-defined limit on the number of versions that an object can have.[8] The current version of an object is updatable, but old versions may be read-only depending upon the implementation. As in the case of persistence, versioning is an object property and not a class property. Objects belonging to the same class can have different numbers of versions.

Depending upon the application, one may want to access the current version or a specific version. For example, an address-book object that keeps track of current addresses would like to reference the latest versions of person objects (to access their latest addresses). On the other hand, a software configuration object may want to reference specific versions of objects representing component modules.[9]

Our version model supports both of these modeling requirements. An object and all its versions are treated as one logical object with one id. A pointer to a persistent object can refer to a logical object in which case it is called a *logical* id or to a specific version of an object in which case it is called a *version* pointer. Accessing an object using a logical object id results in access to the current version of the object.

Thus, the address-book object can refer to the current versions of `person` objects using logical ids. When the address of a person is changed and a new version of the `person` object created, no change is required to the address-book object. On the other hand, version pointers are used to refer to specific versions, e.g., in the software configuration object. O++ provides macros for coercing version pointers to logical object ids and vice versa.

Updating a persistent object does not automatically create a new version. A new version is created explicitly by calling the macro `newversion`. The effect of the call

```
newversion(p)
```

where `p` is a logical object id is to create a new version of the specified object.[10] Henceforth, `p` refers to the new version. Logically, the contents of the new version will be identical to those of the previous version.

Macro `previous` can be used for accessing different versions of an object. It takes a pointer to a persistent object and a non-negative integer n and returns a pointer to the n^{th} previous version; if there is no such version, then `previous` returns the null pointer. Thus, the total salary over the last five years of an employee can be computed as:

```
persistent Employee *e, *ve;
...
for(i=0; i < 5; i++)
  if ((ve = previous(e, i)) == NULL)
    error("%s has worked < 5 yrs\n",e->name);
  total += ve->salary;
}
```

Given a logical object id, operator `pdelete` deletes the object and all its versions. Given a version pointer, `pdelete` deletes the specified version.[11]

5. CONSTRAINTS

Constraints are used to maintain a notion of consistency beyond what is typically expressible using the type system [37]. Updates that violate the specified constraints should not be permitted. Interpretations of consistency are usually application specific and may be arbitrarily complex. Constraints, which are Boolean conditions, can be associated with classes. Objects must satisfy all the constraints associated with the corresponding class. Constraints can also be used in query optimization.

Constraints are specified in the constraint section of a class definition as follows:

8. We are assuming that we have a large, if not infinite, persistent store. Persistent store may be hierarchically organized and old versions may be flushed out to a slower medium. An implementation may impose a limit on the number of versions that can be created.

9. Orion [12] and Iris [14] also provide facilities for specific reference to a particular version and generic reference to a versioned object.

10. In this paper, we will only describe the linear versioning mechanism. O++ allows the version graph of an object to be a tree; see [4] for details.

11. Implementations may choose not to allow the deletion of specific versions.

```
constraint:
    constraint1;
    constraint2;
    ...
    constraintn;
```

constraint$_i$ is a Boolean expression that refers to components of the specified class. Constraints are checked at the end of constructor and member (friend) function calls. Therefore, although we do not prohibit accessing components of an object directly (if the components are specified to be public), it is the programmer's responsibility to ensure that such accesses do not violate any constraints (because no checking will be done automatically).

Here is an example of a constraint:

```
constraint:
  supplier_state == Name("NY") ||
          supplier_state == Name("");
```

If a constraint is not satisfied, then the access is aborted[12] and the object restored to its original state.

5.1 Constraint Inheritance

A derived class inherits the constraints of its parent class and new constraints can be added. Consequently, constraints can be used to specialize classes, as in

```
class female: public person {
public:
    ...
constraint:
    sex == 'f' || sex == 'F';
};
```

Such constraint-based specializations are useful in many applications, e.g., in frame-based knowledge representation systems [15].

6. Triggers

Triggers (also called alerters or monitors), like integrity constraints, monitor the database for some conditions, except that these conditions do not represent consistency violations [37]. When trigger conditions become true, the associated trigger action is executed. Triggers are necessary for supporting active databases, such as those used in computer integrated manufacturing, power distribution network management, air-traffic control, etc. and are found in many database systems such as POSTGRES [51], HiPAC [21], and Vbase [7].

Triggers are associated with objects. There are two types of triggers: *once-only* (default) and *perpetual* (specified using the keyword `perpetual`).[13] A once-only trigger is automatically deactivated after the trigger has "fired", and it must then be reactivated explicitly if desired. On the other hand, a perpetual trigger is automatically reactivated after being fired.

Triggers are specified within class definitions:

```
trigger:
    [perpetual] T1 (parameter-decl1) : trigger-body1
    [perpetual] T2 (parameter-decl2) : trigger-body2
    ...
    [perpetual] Tn (parameter-decln) : trigger-bodyn
```

T_i are the trigger names. Trigger parameters can be used in the trigger bodies which have the form

trigger-condition ==> *trigger-action*

within *expression* ? *trigger-condition* ==> *trigger-action* [: *timeout-action*]

The second form is used to specify a timed trigger. Once activated, the timed trigger must fire within the specified period (floating-point value specifying the time in seconds); otherwise, the timeout action, if any, is fired.

Triggers are set by explicitly activating them after the object has been created. A trigger T_i associated with an object whose id is *object-id* is activated by the call

object-id->T_i (*arguments*)

The trigger activation returns a trigger id (value of the predefined class `triggerid`). The object id can be omitted when activating a trigger in the body of a member function. Note that there can be more than one activation of a trigger in effect.

An active trigger fires when its condition becomes true. Firing means that the action associated with the trigger is "scheduled" for action. Unlike [51], a trigger action is not considered part of the transaction that causes the trigger to be fired (triggering transaction). Each trigger firing results in the creation of an independent transaction with the trigger action being the transaction body. Conceptually, trigger conditions are evaluated at the end of each transaction. Transactions representing trigger actions are executed after (but not necessarily immediately after) the triggering transaction, i.e., there is "weak coupling" [21] between the triggering transaction and the trigger action. If the triggering transaction is aborted, the trigger actions generated by it are aborted.

Triggers may be deactivated explicitly before they have fired as follows:

~trigger-id

~object-id->T_i (*arguments*)

The first form deactivates a trigger whose trigger id is specified as the argument. The second form deactivates a trigger that was activated with the specified arguments. If a trigger is not active, then deactivating it has no effect. The object id can be omitted when a trigger is referenced within the body of a member function.

We provide one special macro, named `changed`, for use in trigger conditions. This macro returns true if the value of its argument (which must be a data component of the object containing the trigger) has been changed in the current transaction and false otherwise.

Note that the perpetual triggers cannot be simulated with once-only triggers by re-activating the trigger in the trigger action. The problem with doing this is that the trigger action is executed as a transaction at some later time. As a result, the trigger will be inactive for the period between the firing of the trigger and until the activation instruction in the trigger action is executed.

12. Violation of a constraint will cause the transaction of which this access is a part of to be aborted and rolled back at a later time, we may provide additional exception handling mechanisms which will allow the class designer or the user to respond to the violation of a constraint.

13. POSTGRES [51] has similar triggers.

As an example of a once-only trigger, consider the following class `inventory`:

```
class inventitem: public stockitem {
public:
  inventitem(Name iname,double iwt,int xqty,
     int xconsumption, double xprice,
     int xleadtime, Name sname, Addr saddr);
  void deposit(int n);
  int withdraw(int n);
  ...
trigger:
  order(): qty < reorderlevel() ==>
                    place_order(this, eoq());
      /*"this" refers to the object itself*/
};
```

Trigger `order` is activated in the constructor function `inventitem` and in the member function `deposit`:

```
inventitem::inventitem(): (Name iname,
        double iwt,int xqty,int xconsumption,
        double xprice, int xleadtime,
        Name sname, Addr saddr)
{
    ...
    order();  /*trigger activation*/
}
void inventitem::deposit(int n)
{
    qty += n;
    order(); /*trigger activation*/
}
```

The action associated with the trigger `order` will be executed after its condition becomes true (as result of executing the `withdraw` operation whose body is not shown).

As an example illustrating a perpetual trigger, consider a securities broker's database. For each customer, a portfolio is maintained. When the customer wants to buy or sell shares, a buy or sell order is issued by sending an appropriate message to the "market maker". Amongst other objects, the market maker's database has an object for each stock which stores such information as the stock price, and buy and sell orders for that stock from various brokers. The stock price is continuously updated based on the buy and sell orders. Here is the specification of class `stock`:

```
class stock {
  list-of-outstanding-orders;
  void fillorders();
public:
  Name stockname;
  double price;
  stock(Name xname, double xprice);
  void update(double xprice);
  void sell(Name broker,int orderid,
            int amount, double lowlimit);
  void buy(Name broker, int orderid,
            int amount, double highlimit);
  ...
trigger:
  perpetual fill(): changed(price) ==>
                              fillorders();
};
```

Member function `sell` checks the current stock price, and if it is above the lower limit specified for selling the stock, the stock is sold immediately; otherwise, the sell order is added to the list of outstanding orders. Similarly, member function buy causes stock to be bought immediately if the current stock price is below the upper limit specified for buying the stock; otherwise, the buy order is added to the list of outstanding orders.

The stock price is updated by member function `update`. The trigger condition `changed(price)` becomes true whenever `price` changes, and the trigger action `fillorder` is executed:

```
void stock::fillorders()
{
  for each outstanding order a
    if (price >= a.lowlimit) {
      sellstock(this, a.amount, price);
      message(a.broker, a.orderid, a.amount,
                                price, "sold");
      delete outstanding order a;
    }
    if (price <= a.highlimit) {
      buystock(this, a.amount, price);
      message(a.broker, a.orderid, a.amount,
                              price, "bought");
      delete outstanding order a;
    }
}
```

Timed triggers can be used to specify time limits within which the order is to be executed:

```
class stock {
public:
  Name stockname;
  double price;

  stock(Name xname, double xprice);
  void update(double xprice);
  ...
trigger:
  sellorder(Name broker,int orderid,
    int amount,double lowlimit,double time):
      within time ? price > lowlimit ==> {
          sellstock(this, amount, price);
          message(broker, orderid, amount,
                                price, "sold");
        }
  buyorder(Name broker,int orderid,
    int amount,double highlimit,double time):
       within time ?  price < highlimit ==> {
          buystock(this, amount, price);
          message(broker, orderid, amount,
                              price, "bought");
        }
};
```

When the triggers `sellorder` and `buyorder` are activated, the time limit (parameter `time`) within which the trigger must fire is specified.

7. CONCLUSIONS

An important goal of research in programming languages design is to provide a better fit between problems and programming notation. We have tried to do this for databases by using one unified data model for data definition and data manipulation. We started with the object-oriented facilities of C++ and extended them (minimally) with features to support the needs of databases, putting to good use all the lessons learned in implementing today's database systems. The resulting database programming language O++ makes available to database objects the full range of type-structures and multiple inheritance found in C++ and, at the same time, provides features such as persistence, iterators, fixpoint querying capabilities, versions, constraints, and triggers.

This paper presented the linguistic facilities for specifying and accessing data provided in O++. As mentioned, the rationale for our design decisions is discussed in [3]. We have begun a prototype implementation of O++, and we hope to report on the implementation strategies and experiences in the near future.

8. ACKNOWLEDGMENTS

The persistence model was influenced by discussions we had with Steve Buroff and Mike Carey. David DeWitt's suggestions lead us to modify the paper that made our presentation more complete. We are also grateful to J. Annevelink, S. Buroff, H. T. Chou, A. R. Feuer, D. H. Fishman, R. Greer, L. M. Haas, H. V. Jagadish, J. D. Jordan, K. Kelleman, B. W. Kernighan, D. E. Perry, A. Singhal, D. Shasha, W. P. Weber, and A. L. Wolf for their comments and suggestions.

REFERENCES

[1] ANSI C, Draft Proposed American National Standard for Information Systems—Programming Language C, , 1988.

[2] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[3] R. Agrawal and N. H. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", *2nd Int'l Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[4] R. Agrawal and N. H. Gehani, "Version Managment in ODE", AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, 1989..

[5] A. V. Aho and J. D. Ullman, "Universality of Data Retrieval Languages", *Proc. 6th ACM Symp. Principles of Programming Languages*, San-Antoniao, Texas, Jan. 1979, 110-120.

[6] A. Albano, L. Cardelli and R. Orsini, "Galileo: A Strongly Typed Interactive Conceptual Language", *ACM Trans. Database Syst. 10*, 2 (June 1985), 230-260.

[7] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented development Environment", *Proc. OOPSLA '87*, Orlando, Florida, Oct. 1987, 430-440.

[8] M. P. Atkinson and O. P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys 19*, 2 (June 1987), 105-190.

[9] T. M. Atwood, "An Object-Oriented DBMS for Design Support Applications", *Proc. IEEE 1st Int'l Conf. Computer-Aided Technologies*, Montreal, Canada, Sept. 1985, 299-307.

[10] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 16-52.

[11] F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, "FAD, a Powerful and Simple Database Language", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 97-105.

[12] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk and N. Ballou, "Data Model Issues for Object-Oriented Applications", *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 3-26.

[13] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twitchell and T. Wise, "GENESIS: An Extensible Database Management System", *IEEE Trans. Software Eng.*, Nov. 1988.

[14] D. Beech and B. Mahbod, "Generalized Version Control in an Object-Oriented Database", *Proc. IEEE 4th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 14-22.

[15] R. J. Brachman and H. J. Levesque, (ed.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[16] P. Buneman and M. Atkinson, "Inheritance and Persistence in Database Programming Languages", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 4-15.

[17] L. Cardelli, "Amber", Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1984.

[18] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. H. Schuh, E. J. Shekita and S. L. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", Computer Sciences Tech. Rep. #808, Univ. Wisconsin, Madison, Nov. 1988.

[19] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner and B. W. Wade, "SEQUEL 2: A Unified Approach To Data Definition, Manipulation, and Control", RJ 1798, IBM, June 1976.

[20] G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Massachusetts, June 1984, 316-325.

[21] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny and R. Jauhari, "The HiPAC Project: Combining Active Databases and Timing Constraints", *ACM-SIGMOD Record 17*, 1 (March 1988), 51-70.

[22] D. D. Detlefs, M. P. Herlihy and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *IEEE Computer 21*, 12 (Dec. 1988), 57-69.

[23] J. Diederich and J. Milton, "ODDESSY: An Object-Oriented Database Design System", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 235-244.

[24] K. Dittrich and R. Lorie, "Version Support for Engineering Database Systems", Rep. RJ4769, IBM Research Lab., San Jose, California, July 1985.

[25] K. R. Dittrich, W. Gotthard and P. C. Lockemann, "DAMOKLES – A Database System for Software Engineering Environments", LNCS 244, 1987.

[26] A. Ege and C. A. Ellis, "Design and Implementation of GORDION, an Object Base Management System", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 226-234.

[27] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan and M. C. Shan, "Iris: An Object-Oriented Database System", *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 48-69.

[28] N. H. Gehani and W. D. Roome, "Concurrent C", *Software—Practice & Experience 16*, 9 (1986), 821-844..

[29] M. F. Hornick and S. B. Zdonik, "A Shared Segemented Memory System for an Object-Oriented Database", *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 70-95.

[30] S. E. Hudson and R. King, "Object-Oriented Database Support for Software Environments", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 491-503.

[31] R. Katz, E. Chang and E. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986.

[32] S. N. Khoshafian, G. P. Copeland and 406-416, "Object Identity", *Proc. OOPSLA '86*, Portland, Oregon, Sept. 1986.

[33] C. Lecluse, P. Richard and F. Velez, "O_2, an Object-Oriented Data Model", *Proc. ACM-SIGMOD 1988 Int'l Conf. on Management of Data*, Chicago, Illinois, June 1988, 424-433.

[34] D. Maier, J. Stein, A. Otis and A. Purdy, "Development of an Object-Oriented DBMS", *Proc. OOPSLA '86*, Portland, Oregon, Sept. 1986, 472-486.

[35] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model", *Proc. Int'l Workshop Object-Oriented Database System*, Asilomar, California, Sept. 1986.

[36] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research", Tech. Rep.-88-001, Int'l Computer Science Inst., Berkeley, California, May 1988.

[37] R. S. Nikhil, "Functional Databases, Functional Languages", in *Data Types and Persistence*, M.P. Atkinson, P. Buneman and R. Morrison (ed.), Springer Verlag, 1988, 51-67.

[38] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulis and M. Stanley, "Implementation of a Compiler for a Semantic Data Model", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 118-131.

[39] P. O'Brien, P. Bullis and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proc. Int'l Workshop Object-Oriented Database System*, Asilomar, California, Sept. 1986, 113-123.

[40] H. B. Paul, H. J. Schek, M. H. Scholl, G. Weikum and U. Deppisch, "Architecture and Implementation of the Darmstadt Database Kernel System", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 196-207.

[41] Persistent Programming Research Group, "The PS-Algol Reference Manual, 2d ed.", Tech. Rep. PPR-12-85, Computing Science Dept., Univ. Glasgow, Glasgow, Scotland, 1985.

[42] J. E. Richardson and M. J. Carey, "Persistence in the E Language: Issues and Implementation", Computer Sciences Tech. Rep. #791, Univ. Wisconsin, Madison, Sept. 1988.

[43] J. E. Richardson, M. J. Carey and D. H. Schuh, "The Design of the E Programming Language", Computer Sciences Tech. Rep. #824, Univ. Wisconsin, Madison, Feb. 1989.

[44] L. Rowe and K. Shoens, "Data Abstraction, Views and Updates in RIGEL", *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, Boston, Massachusetts, May-June 1979, 77-81.

[45] L. A. Rowe and M. R. Stonebraker, "The POSTGRES Data Model", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 83-96.

[46] H. J. Schek and M. Scholl, "The Relation Model with Relation-Valued Attributes", *Information Sys. 11*, 2 (1986), .

[47] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl and R. Meyer, "OOPS – An Object Oriented Programming System with Integrated Data Management Facility", *Proc. IEEE 4th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 118-125.

[48] J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247-261.

[49] P. M. Schwarz, W. Chang, J. C. Freytag, G. M. Lohman, J. McPherson, C. Mohan and H. Pirahesh, "Extensibility in the Starburst Database System", *Proc. Int'l Workshop Object-Oriented Database System*, Asilomar, California, Sept. 1986.

[50] J. M. Smith, S. Fox and T. Landers, *ADAPLEX: Rationale and Reference Manual, 2nd ed.*, Computer Corp. America, Cambridge, Mass., 1983.

[51] M. Stonebraker, E. N. Hanson and S. Potamianos, "The POSTGRES Rule Manager", *IEEE Trans. Software Eng. 14*, 7 (July 1988), 897-907.

[52] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley., 1986.

[53] W. Tichy, "RCS: A System for Version Control", *Software Practice and Experience 15*, 7 (July 1986), 637-654.

[54] A. Wasserman, "The Data Management Facilities of PLAIN", *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, Boston, Massachusetts, May-June 1979.