INTEGRATING AN OBJECT-ORIENTED PROGRAMMING SYSTEM WITH A DATABASE SYSTEM

Won Kim, Nat Ballou,Hong-Tai Chou, Jorge F. Garza, Darrell Woelk

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

Jay Banerjee*

Unisys Corporation
P.O. Box 64942, M.S. WE-3C
St. Paul, MN 55164

ABSTRACT

There are two major issues to address to achieve integration of an object-oriented programming system with a database system. One is the language issue: an object-oriented programming language must be augmented with semantic data modeling concepts to provide a robust set of data modeling concepts to allow modeling of entities for important real-world applications. Another is the computational-model issue: application programmers should be able to access and manipulate objects as though the objects are in an infinite virtual memory; in other words, they should not have to be aware of the existence of a database system in their computations with the data structures the programming language allows. This paper discusses these issues and presents the solutions which we have incorporated into the ORION object-oriented database system at MCC.

1. INTRODUCTION

In the Advanced Computer Architecture Program at MCC, we have built a prototype object-oriented database system, called ORION. Presently, it is being used in supporting the data management needs of PROTEUS, an expert system shell also prototyped in the Advanced Computer Architecture Program at MCC. In ORION we have directly implemented the object-oriented paradigm [GOLD81, GOLD83, BOBR83, SYMB84, BOBR85], and added persistence and sharability to objects. We have two versions of ORION: a single-user, multi-task system called ORION-1; and a multi-user, multi-task system called ORION-1S, in which a single server provides persistent object management on behalf of several workstations. ORION is intended for applications from the AI [STEF86], multimedia documents [AHLS84, IEEE85, WOEL86], and computer-aided design domains [AFSA86], implemented in the object-oriented programming paradigm. Functions supported in ORION include versions and change notification [CHOU86, CHOU88], composite objects [KIM87], dynamic schema evolution[BANE87], transaction management [GARZ88], associative queries [BANE88], and multimedia data management [WOEL87].

ORION has been implemented in Common LISP[STEE84] on a Symbolics 3600 LISP machine [SYMB85], and has also been ported to the SUN workstation under the UNIX operating system. ORION extends Common LISP with object-oriented programming and database capabilities. (In contrast, the proposed Common LISP Object System (CLOS) [CLOS88] is only an object-oriented language extension to Common LISP.)

The objective of integrating a programming language system with a database system has provided much impetus to research in both the programming language and database communities. This has been motivated by the desire to enhance programming language systems with the benefits of database systems, such as persistent and sharable storage, database integrity control, and associative access to the database. One of the major objectives of ORION was the integration of a programming language (Common LISP) with a database system. To achieve this, we had to address two major issues. One is the language issue. Programming languages in general do not have the primitive semantic data modeling concepts which are necessary to model real-world entities. These include instantiation (an object is an instance of a class), aggregation (an object consists of a number of attributes), and generalization (a class can have a number of subclasses), composite objects (an object consists of a number of exclusive component objects), versions (an object may have a number of versions), and so on. Object-oriented programming languages embody some of these concepts, namely, instantiation, aggregation, and generalization; and as such, one may say that the gap

---

* Banerjee was a member of the MCC ORION group when this work was done.

between object-oriented programming languages and databases is substantially narrower than that between other programming languages (except the logic programming language, whose concepts subsume the relational model of data) and databases. The semantic data modeling concepts that programming languages do not have necessarily require language extensions. The traditional solution is to define a database language (such as SQL [IBM81]), and have the application programmers embed database-language statements in the application programs (typically written in FORTRAN, COBOL, PL/1, C, etc.). The trouble with this approach is that the programmers must learn to program in more than one language, and program the mapping between the data structures supported in the programming language and those supported in the data model [COPE84]. Clearly, database extensions to programming languages must not introduce new or conflicting language paradigms, and require unproductive mapping of data structures.

Another issue is the computational model that programming languages in general, and object-oriented programming languages in particular, imply. Programming language systems assume that all objects are in virtual memory, and computations are performed by chasing memory pointers and storing and referencing objects in data structures supported in the programming languages. In applications that require a large amount of data to be extensively accessed and manipulated, such as simulation of a complex assembly of mechanical parts or a design of an electronic device, computations cannot be offloaded to any traditional database system. This is because of the excessive overhead of crossing the boundary between the application and the database system. One of the important shortcomings of conventional database systems is that they are not designed to support navigational access over a large set of objects in virtual memory. The cost of fetching a single object is prohibitively high, especially in relational database systems (compared to a few memory lookups that it takes an application program manipulating objects in virtual memory). If the compute-intensive applications are to use the conventional database system simply as a repository of persistent objects, and copy objects out of the database into the application's address space for computations, it is entirely up to the application to manage consistency of the objects in its address space and to map the data structures to and from the database.

The objective of this paper is to present our approaches to addressing these two major issues in integrating a programming language system with a database system. In particular, we will outline our approach to integrating the object-oriented concepts with a programming language, further augmenting them with additional semantic data modeling concepts. Further, we will describe the architectural concepts and data structures we use in ORION to give the application programmers the illusion and the performance of an infinitely large virtual memory for their objects.

2. DATABASE EXTENSIONS TO A PROGRAMMING LANGUAGE

In this section, we will outline the ORION object-oriented extensions to Common LISP. First, we will describe the syntax for capturing the basic object-oriented programming concepts. Then we will augment the basic syntax with a number of database concepts, including associative access to objects and application-related data modeling concepts such as versions and composite objects. The objective of this section is to outline our approach to integrating these database concepts into the basic object-oriented programming paradigm, and demonstrate that the extensions are fairly simple. Therefore, we will not provide the full syntax and semantics of the ORION interface. The syntax, as presented in an abbreviated form in this paper, is similar in principle to that of a number of other object-oriented languages, such as Flavors [SYMB84] and LOOPS [BOBR83].

2.1 BASIC OBJECT-ORIENTED EXTENSIONS

The following message creates the definition of a new class.

(**make-class** Classname
:superclasses ListofSuperclasses
:attributes ListofAttributes
:methods ListofMethodSpecs)

Classname is the name of the new class. All keyword arguments are optional. The ListofSuperclasses associated with the **:superclasses** keyword is a list of the superclasses of the new class. The ListofAttributes associated with the **:attribute** keyword is a list of attribute specifications. An attribute specification is a list consisting of an attribute name and keywords with associated values, as follows:

(AttributeName [**:domain** DomainSpec]
[**:inherit-from** Superclass])

A DomainSpec is a LISP data type, a class, or a set of LISP data types or classes; and is used to specify the type(s) of an attribute. If the keyword **:inherit-from** is specified, the associated value is the name of the superclass from which the attribute will be inherited. Otherwise, the attribute is inherited from the first superclass in the ListofSuperclasses.

The ListofMethodSpecs associated with the **:methods** keyword is a list of pairs (MethodName SuperClass). The MethodName is the name of a method to be inherited from the SuperClass. The SuperClass is a class name. If the keyword **:methods** is not specified, methods are inherited from superclasses, and conflicts are resolved on the basis of superclass ordering.

An instance can be created by sending a **make** message to the class to which the instance will belong.

```
(make Classname  :Attribute1 value1
                 ...
                 :AttributeN valueN)
```

2.2 DATABASE EXTENSIONS

In this section we outline some important extensions to the basic object-oriented concepts. The extensions include associative access to objects, semantic data modeling concepts, and database control functions.

2.2.1 Associative Access

Once the database size exceeds the virtual memory size, it is obviously important to bring into virtual memory only those objects which the application will need. Programming languages in general have traditionally not been concerned with queries which will return from the database a small subset of the database that satisfies search conditions. To select all instances (or any one instance) of a class that satisfy a given query expression, we use a **select** (or **select-any**) message. A *set object* (possibly an empty set) containing these instances is returned. The messages for selection have the following format, where QueryExpression is a Boolean expression of predicates:

```
(select       Class  QueryExpression)
(select-any   Class  QueryExpression)
```

An example query is to select the instances of a class Vehicle whose weight is over 5000 lbs.

```
(select  'Vehicle  '(> Weight  5000))
```

To delete all instances of a class that satisfy a given query expression, a **delete** message is used.

```
(delete  Class  QueryExpression)
```

To delete a specific object, a delete-object message is used.

```
(delete-object  Object)
```

where Object is the object identifier.

Similarly, a change message is used to replace the value of an attribute of all instances of a class that satisfy a given Boolean expression.

```
(change Class  [QueryExpression]
               AttributeName NewValue)
```

2.2.2 Semantic Data Modeling Concepts

ORION supports two semantic data modeling concepts which are not part of the conventional object-oriented paradigm.

Composite Objects

The conventional object-oriented paradigm, although powerful, does not capture the IS-PART-OF relationship between objects; that an object *is a part of* another object. In [KIM87], we define a *composite object* as an object with a hierarchy of exclusive component objects. The classes to which the objects of a composite object belong are also organized in a hierarchy. This hierarchical collection of classes is called a *composite object hierarchy*. A non-root class on a composite object hierarchy is called a *component class*. Each non-leaf class on a composite object hierarchy has one or more attributes whose domains are the component classes. We call such attributes *composite attributes*. A constituent object of a composite object references an instance of its component class through a composite attribute.

To support composite objects, we extend the **make-class** message as follows.

```
(AttributeName  [:composite TrueOrNil])
```

The keyword **:composite** declares whether an attribute is a composite attribute.

An instance can be made a part of a composite object only at the time of creation of that instance. This is done by extending the **make** message with a **parent** argument:

```
(make Classname
      :parent (ParentObject ParentAttributeName)
      :Attribute1 value1
      ...
      :AttributeN valueN)
```

The keyword **:parent** is associated with a pair (ParentObject ParentAttributeName), where ParentObject with an attribute ParentAttributeName is to reference the instance being created. The **make** message, without the **:parent** keyword, is used to create root instances of composite objects.

Versions

The ORION model of versions and its implementation are presented in [CHOU88]. Here we will outline (not fully explain) some of the messages we support for versions.

An object is either *versioned* or *non-versioned*. A versioned object is an instance of a class which the application declares to be *versionable*. The **make-class** message was extended with an additional keyword argument, versionable, as follows.

```
(make-class Classname
            :versionable TrueOrNil)
```

The keyword **:versionable** can have a value true or nil, indicating whether versions can be created for instances of the class.

Our model distinguishes *transient versions* (temporary versions) from *working versions* (stable versions). A transient version may be created from scratch or *derived* from an existing version. Any number of transient versions may be derived at any time from an existing version, giving rise to a *version-derivation hierarchy* for each versioned object. We use the term *version instance* to refer to a specific version, and *generic instance* to refer to the abstract versioned object. A generic instance maintains the history of derivation of version instances for a versioned object.

When the user issues a **make** message to a versionable class, ORION creates a generic object, as well as the first version instance of the versionable object. The new version instance is a transient version, and becomes the root of the version-derivation hierarchy for the versionable object. The optional keyword arguments of a **make-class** message supply attribute names and values for the version instance.

To derive a new version from an existing version, a **derive-version** message is sent to a VersionedObject, as follows.

(**derive-version** VersionedObject)

The message causes a copy to be made of the VersionedObject. The copy becomes a new transient version, and is assigned a new version number and an object identifier.

In ORION, both the generic instance and a version instance of the generic instance have object identifiers. An object, either a version instance or a non-versioned object, may reference one or more other objects. If an object references a version instance, the reference may be the object identifier of a generic instance or that of a version instance. If the reference is to a generic instance, the system dynamically binds the object to a *default version instance*.

The **delete-object** message is used to delete a version instance or a generic object. If the message is sent to a generic object, the entire version-derivation hierarchy is deleted. In other words, all version instances of the versionable object, as well as the generic object, are deleted. If a **delete-object** message is sent to a version instance, the version instance is deleted. If the version instance is a transient version, or a working version from which no other versions have been derived, the history (or version descriptor) of the version instance is deleted as well. (The history of a version instance is maintained within the generic object of the version instance) If the version instance is the only version instance of the versionable object, the generic object is also deleted. If the **delete-object** message is sent to a working version that has other derived versions, however, the history of the version instance is not deleted.

To fetch, update, or delete version instances of a versionable class based on a QueryExpression, the **select**, **change**, and **delete** messages shown earlier can be used, without any changes in their syntax or semantics. These messages cause all version instances of the specified class to be examined.

2.2.3 Database Control Functions

ORION provides an extensive set of messages for the user to control the integrity and resources of the database, including physical clustering of objects, schema evolution (changes to the definition of a database), secondary index management, and so on. Because of space limitations, we will indicate only some of the messages for transaction management and schema evolution here.

Transactions [GRAY78] are an important capability in database systems. A *transaction* is an atomic sequence of database operations that takes the database from one consistent state to another consistent state, and is a unit of concurrency control and recovery. If a transaction aborts, all database changes made by the transaction are backed out. A transaction is shielded from the effects of other concurrently executing transactions. If a transaction commits, all updates are safely recorded in stable storage. The messages to commit and abort transactions are as follows:

(commit)
(abort)

The schema of an ORION database is a class hierarchy (actually a directed acyclic graph); and as such two types of changes to the schema are meaningful: changes to the definitions of a class (contents of a node) in the class hierarchy, and changes to the structure (edges and nodes) of the class hierarchy. Changes to the class definitions include adding and deleting attributes and methods. Changes to the class hierarchy structure include creation and deletion of a class, and alteration of the IS-A relationship between classes (adding and deleting the superclass-subclass relationship between a pair of classes). The complete taxonomy of schema changes we allow in ORION is given in [BANE87].

To append a class to the superclass list of an existing class, or to remove a superclass from the superclass list of an existing class, one can use the messages:

(add-superclass Class Superclass)
(remove-superclass Class Superclass)

where the arguments Class and Superclass are the names of classes.

The **change-attribute** message given below can be used to add a new attribute to a class, to change the inheritance of an attribute, or to change the properties of an attribute. All keyword arguments in the message are optional, and they indicate the types of change to be made to the attribute.

(change-attribute Class AttributeName
[:recursivep TrueOrNil]
[:domain DomainSpec]
[:inherit-from Superclass])

The keyword **:recursivep** has a default value T. If nil, it indicates that the change to the attribute definition is limited to the specified class, and must not be propagated to its subclasses. If non-nil, it indicates that the change must be propagated.

To add a new attribute as a locally defined attribute, the **:inherit-from** keyword is used with a nil value. If the

attribute name was a previously defined attribute of the class, it is simply re-defined. To change the inheritance of an attribute, the **:inherit-from** keyword is used, and its associated value is the name of the superclass from which the attribute is to be inherited.

3. IN-MEMORY OBJECT MANAGEMENT

Conventional database systems allocate a buffer pool of page frames in an attempt to pin in virtual memory data likely to be accessed again soon [TRAI82]. The pages in the buffer pool are accessed using a fix/unfix protocol [TRAI82, EFFE84]. That is, the caller (various components of a database system) must request the page buffer manager to pin down a page in memory before accessing it. Further, when the caller is done with the page, it informs the page buffer manager that the buffer page can be re-used. A page is thus guaranteed to stay in the same memory location during a fix/unfix period; that is, there is no danger that it is swapped out while the caller (access manager or storage manager) is still working on it. The page buffer manager typically uses an LRU replacement algorithm or its variants [CHOU85]. A buffer with a positive fix count, that is, a buffer which is still being worked on, is exempt from replacement decisions. The page buffer manager keeps track of all the pages in the buffer pool through a page table.

The buffering scheme used in conventional database systems is not adequate for supporting a programming language environment. One problem with this approach is that it tends to force the application programmers to map the data structures between the application and the database system. In a programming language environment, for storage and retrieval efficiency, the objects need to be stored on disk in one format (the disk format); however, the applications must be able to manipulate the objects in their in-memory format, the format supported by the programming language. A somewhat related problem is that, as we discussed in Section 1, database techniques for maintaining database consistency do not extend to the objects in virtual memory which the applications directly access and manipulate. Another problem with the conventional buffering scheme is memory utilization. As many important applications need to cache a large number of objects in virtual memory to perform extensive computations on them, it is often undesirable to keep page frames in the database buffer pool which contain many unneeded objects.

To solve the above problems, we have adopted a dual-buffer management scheme, in which the available database buffer space is partitioned into a page buffer pool and an object buffer pool. The *workspace* discussed in the context of the GemStone database system [MAIE86] is similar to the object buffer pool in ORION. To access an object, the page that contains the object is brought into a page buffer, and then the object is located, retrieved, and placed in an object buffer. ORION supports data structures for efficiently managing objects in the object buffer pool, and addresses issues that arise from the fact that the object buffer pool and the database may contain different copies of the same object during a transaction (a sequence of read and write requests against the database; this sequence is treated by the database system as an atomic action for purposes of recovery). Applications can directly access the objects in the object buffer pool, and the transaction management feature of ORION ensures database consistency (concurrency control and crash recovery) for these in-memory objects. In this section, we describe the data structures ORION has implemented to manage in-memory objects, that is, objects in the object buffer pool. The impacts of dual buffering on the architecture of a database system, and the solutions we have implemented in ORION, will be discussed in Section 4.

3.1 OBJECT BUFFERING

Figure 1a shows a high-level block diagram of the ORION architecture. The message handler receives all

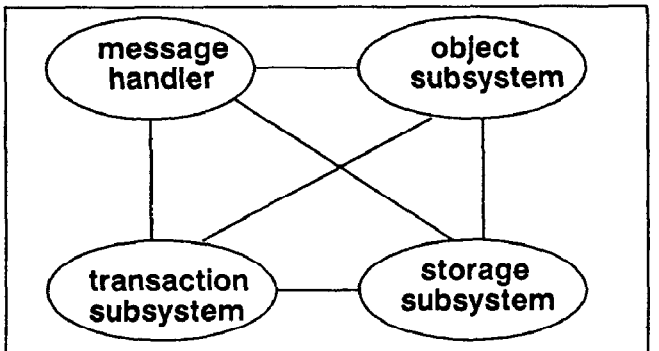

Figure 1a. ORION Architecture

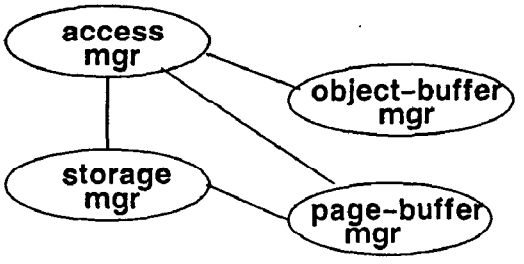

Figure 1b. ORION Storage Subsystem

messages sent to ORION objects. The object subsystem provides high-level functions, such as schema evolution, version control, query optimization, and multimedia information management. The storage subsystem provides access to objects on disk. It manages the allocation and deallocation of pages on disk, finds and places objects on the pages, and moves pages to and from the disk. The transaction subsystem provides a concurrency control and recovery mechanism to protect database integrity while allowing concurrent execution of multiple transactions. As in conventional database systems [GRAY78], concurrency control uses a locking

protocol, and a logging mechanism is used for recovery from system crashes and aborts.

The storage subsystem consists of the access manager and the storage manager, as shown in Figure 1b. The storage manager manipulates objects in their disk format and performs the transformation between the disk format and the in-memory format. The access manager controls the transfer of objects between the object buffer pool and the page buffer pool.

The ORION page buffer manager is similar to the buffer manager in conventional database systems [TRAI82]. It manages a pool of page frames and implements a page replacement algorithm. The page buffer pool serves as a staging area for regular (small) objects as well as the buffer area for caching portions of long multimedia objects.

The object buffer manager performs two major functions: it manages the object buffer pool; and maintains a virtual-memory address table (*resident object table or ROT*) for objects in the object buffer pool. There is a single physical object buffer pool, and multiple applications may concurrently access objects in the buffer pool. An application can accumulate objects in the object buffer pool by creating new objects or sending object requests to ORION.

A request to access an object through its system-wide unique identifier (UID) is directed to the access manager. It calls the object manager to first search the ROT. If the object is not registered in the table (i.e., an object fault occurs), the access manager calls the storage manager to retrieve the object from the database, and have the object buffer manager register it in the table and place the object in an object buffer.

The most frequent operation to the ROT is looking up the location of an object. Since the ROT can grow to a substantial size, a hash table is used to speed up associative searches based on UIDs. The key of the hash table is the UID, and the value is a pointer to the descriptor for the object associated with the UID (to be discussed shortly). Insertions and deletions of the ROT entries are two other frequent operations that are necessary for supporting object swapping. Sometimes a collection of objects in the buffer pool must be accessed: for example, when the modified objects need to be flushed to the database to commit a transaction, or when the contents of the object buffer pool are invalidated because of changes to the database schema (we will discuss these in more detail in Section 4).

Buffer management for objects in the object buffer pool is inherently more complex than that for pages because of the variability of object sizes. Placement of a newly retrieved object is a nontrivial task, since a free block of memory with at least the size of the object must be found. Fragmentation of the buffer pool becomes more severe as objects of different sizes are swapped in and out of memory. Expensive compaction of the object buffer pool may be required from time to time. The difficulty of object buffering is further compounded by the fact that objects in the buffer pool are directly accessible to the application. It is difficult, if not impossible, to keep track of all the outstanding object references (memory pointers) in the application program. Adding the fix/unfix protocol to the application interface would make the interface too cumbersome. Therefore, we need to rely on a garbage collection technique to reclaim space occupied by inactive objects.

3.2 Resident Object Descriptors

When the application requests an object, ORION returns a pointer to a descriptor of the object in the object buffer pool, rather than a pointer to the object. This is also the approach taken in LOOM [KAEH81]; however, our ROD structure consists of several fields in addition to those used in LOOM, because of our consideration for the performance and integrity of the database in a multiple concurrent-user environment. (The rest of this paper will make this clear.) The descriptor, called the resident object descriptor (ROD), is illustrated in Figure 2a. The ROD is an intermediate data structure between

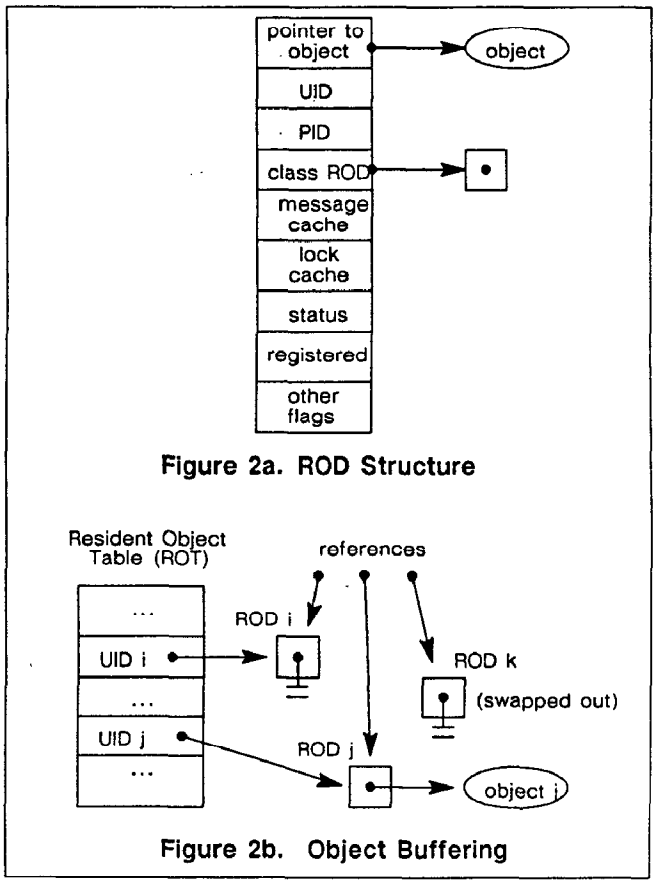

Figure 2a. ROD Structure

Figure 2b. Object Buffering

the ROT and the actual object. The pointer-to-object field in the ROD contains a pointer through which the contents of the object can be accessed. The UID field contains the UID of the object; the PID field contains the physical address of the object on disk; and the

class-ROD is the pointer to the ROD of the class object of which the object is an instance. The status field is used to indicate if the object is changed. The registered field is used to indicate whether the ROT contains a pointer to the ROD. The other fields of the ROD, message-cache and lock-cache, are used to speed up message passing and concurrency control; their use will be discussed later.

We have introduced the ROD as a compromise between two somewhat conflicting goals that we need to satisfy for locating in-memory objects. On one hand, we would like to pass back the actual object (actually a pointer to the object) when a user requests it through a UID. On the other hand, we need to retain the ability to swap out any object in memory when such a need arises, for example, when the main memory is flooded with too many old objects. However, without direct hardware support, there is no easy way to catch a direct reference to a swapped-out object and take appropriate actions, as in a paged virtual memory system.

An object may be swapped out when it is not referenced in any active transaction, or when the object buffer pool becomes full. Then the pointer (to the object) in the ROD is changed to nil. The memory pointer to the ROD in the ROT is also removed, so that the ROD itself can be garbage collected when there are no more outstanding pointers to it. However, the ROD stays in memory so that the access manager can bring the object back in case the object is re-accessed through the ROD. There are situations where a ROD may be created before the object is brought into memory. For example, the result of a query is a set of RODs. There is no need to bring all the objects into memory since some of them may not be accessed at all. Under this situation, the access manager will create the RODs at query time, but fetch the objects only on demand. As shown in Figure 2b, some objects may have a ROT entry and a ROD that points to an in-memory copy of the object. Queried objects which have not been brought in have a ROT entry and a ROD containing a nil pointer. Swapped-out objects may have a ROD but no ROT entry. Finally, there are objects that reside only on disk and have no in-memory data structures associated with them.

The ROT is initially empty. The first time an object is accessed by a user, the object buffer manager detects that the object is not in the table and the access manager brings it in from the database. The access manager creates a ROD for the object and has the object buffer manager register it in the ROT with the UID as the key. The access manager passes a pointer to the ROD to the user, who can then directly access the contents of the object through the ROD. When another request comes in for the same object, the object buffer manager will locate the ROD (through the ROT) and pass back a pointer to the ROD. As shown in Figure 3, object y is referenced by both objects x and z through the same ROD.

The object buffering and ROD manipulation discussed above are all transparent to ORION users. An

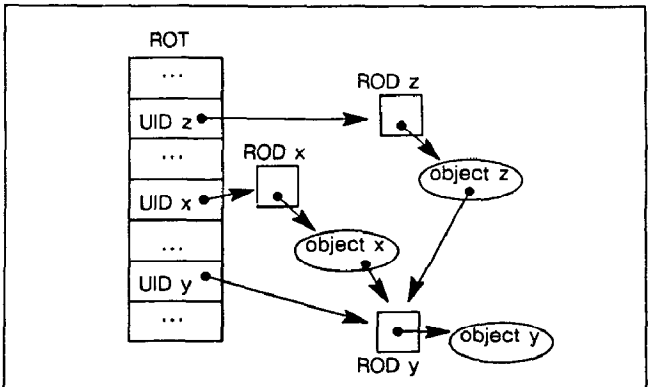

Figure 3. Concurrent References to an Object

ORION user simply sends a message to an object and expects a return message as in any object-oriented system. The objects that a user sees are actually pointers to RODs. To process a message sent to an object, the ORION message handler first examines the ROD and have the object read in from the database, if necessary.

Under our scheme for managing objects in the object buffer pool, it is possible for an extraneous ROD to be created for the same object while a ROD for the object already exists. Figure 4a illustrates this situation. Object i is swapped out, and its ROD, i1, is "de-registered" from the ROT; however, the ROD i1 is still being pointed to by a memory pointer, and is thus not garbage collected. Now another request causes object i to be swapped in (again), creating a second ROD, i2, as well as an entry in the ROT. To minimize the number of obsolete RODs, the next time object i is accessed through the old ROD i1, ORION converts the ROD into an indirection (forwarding) pointer, called an *invisible pointer* [SYMB85], to the new ROD i2, as shown in Figure 4b. The Symbolics machine garbage collects invisible pointers (Figure 4c), making the memory pointer to the old ROD a direct pointer to the new ROD.

When the access manager receives a request to fetch an object based on its UID or ROD, it calls first the object buffer manager to see if the object is already in the object buffer pool. If it is, the access manager returns a pointer to the object's ROD. Otherwise, it directs the storage manager to determine the PID of the object by hashing into UID-PID table for all objects in the database (this is different from the ROT), fetch the page containing the object, isolate the object within the page, and transform the object from its disk format to the in-memory format. Finally, the access manager calls the object buffer manager to place the transformed object in the object buffer pool, and returns a pointer to the object's ROD. The PID of the object is recorded in the PID field of the ROD, shown in Figure 2a. This is to avoid the UID-to-PID translation overhead, when the object has to be flushed (written) to disk, or fetched again after it has been swapped out. To insert new objects, the storage manager determines the PIDs of the objects so as to

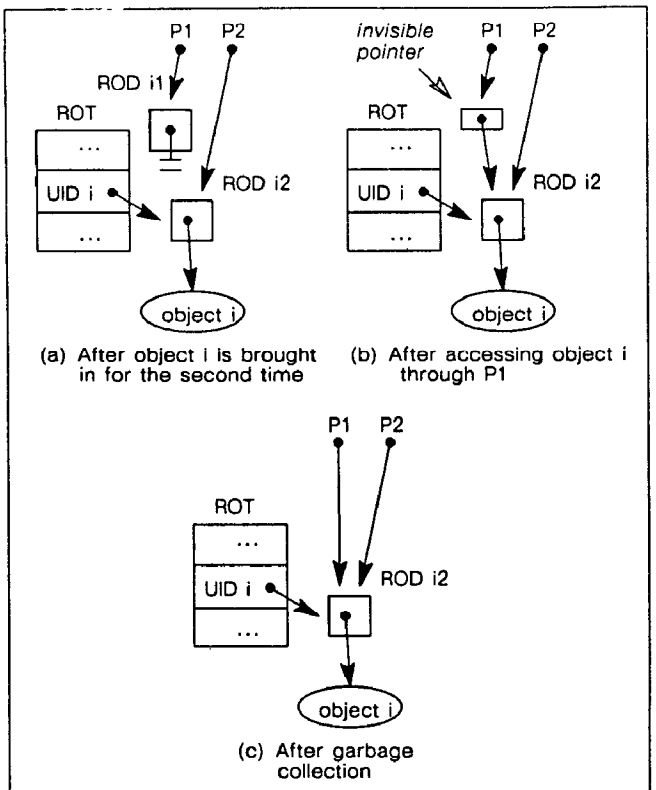

Figure 4. Handling Duplicate RODs

cluster instances of the same class in the same physical segment, registers the objects in the UID-PID hash table for all database objects.

3.3 MESSAGE CACHE

ORION supports four types of messages: instance attribute messages, class attribute messages, instance method messages, and class method messages. We will use the term attribute messages to stand for both instance and class attribute messages, and the term method messages for both instance and class method messages. The function that is placed in the function cell of a message symbol in the LISP system is called a message function (this is called a discriminating function in CommonLoops, and a generic function in Flavors). A message function simply dispatches a message: it contains no knowledge of how the message is implemented.

The main data structure of the message handler is the message cache. The *message cache* consists of two arrays. The first array is the instance message cache: the instance message cache holds instance attribute messages and instance method messages. The second array of the message cache is the class message cache: the class message cache holds class attribute messages and class method messages. Each entry in the message cache contains a vector that holds either class messages or instance messages for a particular class. The entries in the message cache are called *class caches*. There are exactly two class caches for any class: one for class messages, and another for instance messages.

When a message is sent to an ORION object, the message-cache field in the ROD (Figure 2a) is first checked to see if the class cache has already been checked for the object. If the class cache is present, the message is dispatched on the cache. If it is not, the message handler looks up the object's class cache in the appropriate message cache. That is, if the object is a class, the class message cache is used; if it is an instance, the instance message cache is used). If the cache entry is found, it is recorded in the ROD for the object, and the message is dispatched. If it is not found, a cache entry is created, recorded in the appropriate message cache, and placed in the ROD for the object.

3.4 LOCK CACHE

Most commercial database systems use a locking protocol to control access to a shared database by more than one concurrent transactions (applications) [GRAY78]. A transaction must set a lock in an appropriate mode on an object before it can access the object. If a transaction has already set a lock on an object, another transaction attempting to access the same object in a conflicting mode is forced to wait until the first transaction releases the lock. (A read or write request by a transaction conflicts with a write lock set by another transaction.) ORION uses a sophisticated locking protocol [KIM87] based on that used in IBM's SQL/DS [IBM81].

Unlike LOOM, ORION supports multiple concurrent transaction. This means that before the access manager can return the ROD pointer of an object to a requesting transaction, it must check whether another transaction is accessing the object in a conflicting mode. This check is relatively expensive, since the access manager must call the lock manager in the transaction subsystem, and the lock manager must search the lock table.

To avoid this overhead whenever possible, when the access manager first creates or retrieves an object from the database, it encodes in the lock-cache field of the object's ROD the mode of the lock which is set on the object. In this way, the access manager needs to call the lock manager only the first time the object must be locked, and when a read lock on the object must be upgraded to a write lock (i.e., the object was first retrieved from the database with a read lock, and now object must be updated). In all other situations, calls to the lock manager may be avoided.

4. CONSEQUENCES OF OBJECT BUFFERING ON THE DATABASE SYSTEM ARCHITECTURE

Dual buffering has significant consequences on the architecture of a database system. These have to do with the fact that an object may have two different copies during a transaction: one in the object buffer pool and

another in the database. One consequence is obviously the need for a translator to transform an object between its disk format and the in-memory format. A second consequence is the need to invalidate the in-memory objects, when certain types of changes are made to the database schema. A third consequence is the need to screen the database copy of an object from the result of a query, if the object has an in-memory copy.

4.1 OBJECT FORMAT TRANSLATION

To support efficient storage and retrieval, an object has to be packaged into a form that is suitable for disk storage. An object transformer is a part of the storage subsystem in ORION. The storage format for disk-resident objects is as shown in Figure 5.

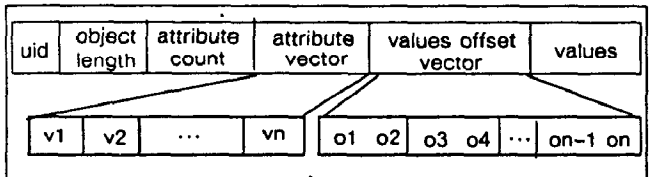

Figure 5. Storage Format for Disk Objects

The uid consists of two parts: the unique identifier of the class to which the object belongs, and the unique identifier of the object within the class. The object-length and attribute-count record the total length of the object and the number of attributes stored in the disk format. The attribute vector consists of the identifiers vi of all attributes for which the object has explicitly specified values. The values-offset vector consists of the offsets oi, in the values part of the object storage format, of the values of the attributes vi. A value can be a primitive value (such as an integer, string, etc.), or a reference to another instance, namely, the uid of the referenced object.

4.2 OBJECT BUFFER FLUSHING

Applications accumulate objects in the object buffer pool by creating new objects, fetching and updating objects from the database. The new objects and updated copies of objects need to be written to the database when the transaction which has created or updated the objects commits (successfully finishes). Of course, the objects are transformed to their disk format before being written to the database. New objects are registered in the UID-PID hash table for database objects, and updated objects replace their old copies in the database.

Further, when changes are made to the database schema (i.e., class definitions and the structure of the class hierarchy) which add or drop an attribute from a class, instances of the affected class which reside in the object buffer pool become invalid and must be purged from the object buffer pool. Of the 20 or so schema change operations ORION allows, the following invalidate objects in the object buffer pool.

1. Add a new attribute to a class
2. Drop an existing attribute from a class
3. Change the inheritance of an attribute (if any previously inherited attributes are lost)
4. Make a class S a superclass of a class C
5. Remove a class S from the superclass list of a class C
6. Drop an existing class

4.3 QUERY PROCESSING

The access manager applies search predicates specified in a query to instances of a class. Our dual-buffering scheme complicates the implementation of a predicate-based access of objects. The two copies of the same object have the same identifier, but may differ in contents. Under an architecture which supports dual buffering, there are two fundamental approaches for processing a predicate-based access. One, which we will call a *dual-buffer evaluation scheme*, is to evaluate the predicates on a class twice: once against the objects of the class in the object buffer pool, and then against those objects of the class in the database whose copies are not in the object buffer pool. Another, which we will call a *single-buffer evaluation scheme*, is to flush (move) the new and updated objects in the object buffer pool to the database, transforming them into the disk format, and then to evaluate the predicates against the database.

Let us discuss the two options in more detail. Consider the situation shown in Figure 6. Objects X, Y,

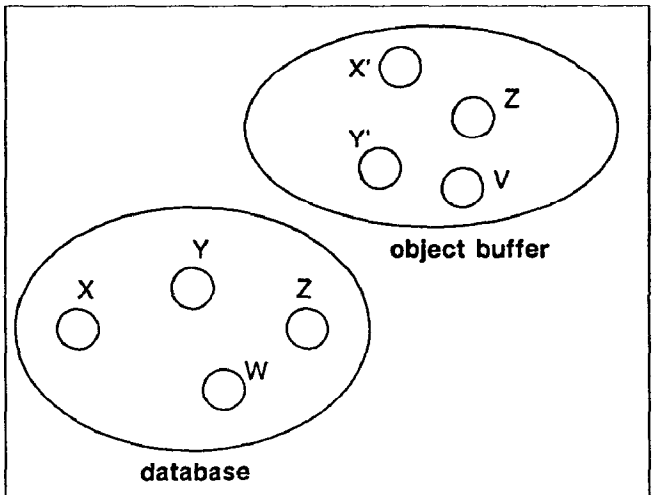

Figure 6. Dual-Buffer Query Evaluation

and Z, all of which are instances of the same class, have been placed in the object buffer pool, and X and Y have subsequently been updated and a new instance V created. We can see that predicates should not be evaluated against X and Y in the database, since updated copies of the objects, X' and Y', exist in the object buffer pool. Also if Z satisfies the predicate, it should not be brought into the object buffer pool, since a copy already exists in the object buffer pool. Further, in the case of a deletion, if X' or Y' satisfies the predicate, its older copy, X or Y, must also be deleted.

The dual-buffer evaluation scheme may proceed in a number of different ways. One reasonable algorithm is as follows, assuming that the access request is confined to a single class C.

(1) Evaluate the predicates against the instances of class C in the object buffer pool, generating a set O of object identifiers that satisfied the predicates.
(2) Evaluate the predicates against the instances of class C in the database, generating a set D of object identifiers that satisfied the predicates. On this step, instances with copies in the object buffer pool are not evaluated again.
(3) The result of the access request is the union of steps 1 and 2.

We note that on step 2 above the predicates need be evaluated against only those objects in the object buffer pool that have been marked as new or updated, since objects that have not been updated after their retrieval from the database have already been evaluated on step 1. When objects get flushed to the database, copies of the updated objects are sent to the access manager and applied to the database. The update flags for the new or updated objects in the object buffer pool are then reset (cleared).

The single-buffer evaluation scheme proceeds as follows, again assuming that objects that satisfy predicates on the attributes of a single class C are to be determined and retrieved.

(1) Select those objects of class C in the object buffer pool that have been marked as new or updated since their retrieval from the database, and force copies of them to the database. This will make the two copies of each new or updated object identical.
(2) Evaluate the predicates against objects of class C in the database, generating a set D of object identifiers that satisfied the predicates. D is the set of all objects to be retrieved.
(3) Eliminate from D those objects of class C that are in the object buffer pool. Retrieve into the object buffer pool only those objects in the resulting set D'.

One major problem with the dual-buffer evaluation scheme is that the objects in the object buffer pool are in a different storage format from that used for objects in the database. As such, we need two different implementations of object search and predicate evaluation algorithms. We also need to support efficient access paths for the objects in the object buffer pool, so that we may avoid sequential searches of all objects. The shortcoming of the single-buffer evaluation scheme is of course that updates must be flushed to the database, and that the objects must be transformed from their in-memory format to the disk format for predicate evaluation. The overhead incurred in object transformation in a LISP environment led us to adopt the dual-buffer evaluation scheme for ORION: under a different environment, the single-buffer evaluation scheme may be superior.

SUMMARY

In this paper, we discussed two major issues in integrating a programming language system with a database system, and presented the solution we have implemented in integrating an object-oriented extension to Common LISP with the ORION object-oriented database system. One is the language issue. A programming language does not provide the primitive semantic data modeling concepts which are necessary to model real-world entities and the relationships among them. Database extensions to a programming language must not introduce a new or conflicting paradigm, or force the programmers to map between data structures in the programming language and those understood by the database system. Another issue is the computational model. A database system must support the illusion (along with adequate performance) that an application program has at its disposal an infinite virtual memory, in which it may access and manipulate the objects in the data structures supported by the programming language.

REFERENCES

[AFSA86] Afsarmanesh, H., Knapp, D., McLeod, D., and Parker, A. "An Object-Oriented Approach to VLSI/CAD," in *Proc. Intl Conf. on Very Large Data Bases*, August 1985, Stockholm, Sweden.

[AHLS84] Ahlsen M., A. Bjornerstedt, S. Britts, C. Hulten, and L. Soderlund. "An Architecture for Object Management in OIS," *ACM Trans. on Office Information Systems*, vol. 2, no. 3, July 1984, pp. 173-196.

[BANE87] Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, San Francisco, Calif., May 1987.

[BANE88] Banerjee, J., W. Kim, and K.C. Kim. "Queries in Object-Oriented Databases," in *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, Calif. Feb. 1988.

[BOBR83] Bobrow, D.G., and M. Stefik. *The LOOPS Manual*, Xerox PARC, Palo Alto, CA., 1983.

[BOBR85] Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. *CommonLoops: Merging Common Lisp and Object-Oriented Programming*, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, Palo Alto, CA., 1985.

[CHOU85] Hong-Tai Chou and D. DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems," in *Proc. 11th Intl Conf. on Very Large Data Bases*, August 1985.

[CHOU86] Chou, H.T., and W. Kim. "A Unifying Framework for Versions in a CAD

Environment," in *Proc. Intl Conf. on Very Large Data Bases*, August 1986, Kyoto, Japan.

[CHOU88] Chou, H.T., and W. Kim. "Versions and Change Notification in an Object-Oriented Database System," to appear in *Proc. 25th Design Automation Conference*, June 1988.

[CLOS88] X3J13 Standards Committee Documents 88-002 and 88-003, 1988.

[COPE84] Copeland, G. and D. Maier. "Making Smalltalk a Database System," *ACM SIGMOD*, June 1984, pp. 316-325.

[EFFE84] W. Effelsberg and T. Haerder. "Principles of Database Buffer Management," *ACM Trans. on Database Systems, vol. 9, no. 4*, pp. 560-595, December 1984.

[GARZ88] Garza, J.F., and W. Kim. "Transaction Management in an Object-Oriented Database System," in *Proc. ACM SIGMOD*, June 1988.

[GOLD81] Goldberg, A. "Introducing the Smalltalk-80 System," *Byte*, vol. 6, no. 8, August 1981, pp. 14-26.

[GOLD83] Goldberg, A., and D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983

[GRAY78] Gray, J.N. *Notes on Data Base Operating Systems*, IBM Research Report: RJ2188, IBM Research, San Jose, Calif. 1978.

[IBM81] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, Jan. 1981.

[IEEE85] *Database Engineering*, IEEE Computer Society, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky).

[KAEH81] T. Kaehler. "Virtual Memory for an Object-Oriented Language," *BYTE*, pp. 378-387, August 1981.

[KIM87] Kim, W., et al. "Composite Object Support in an Object-Oriented Database System," in *Proc. 2nd Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, Florida, Oct. 1987.

[MAIE86] Maier, D., J. Stein, A. Otis, and A. Purdy. "Development of an Object-Oriented DBMS," in *Proc. OOPSLA-86 Conference*, 1986, pp. 472-482.

[STEE84] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb, "Common Lisp", *Digital Press*, 1984.

[STEF86] Stefik, M., and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40-62.

[SYMB84] *FLAV Objects, Message Passing, and Flavors*, Symbolics, Inc., Cambridge, MA, 1984.

[SYMB85] Symbolics Inc., "User's Guide to Symbolics Computers," *Symbolics Manual #* 996015, March 1985.

[TRAI82] I. Traiger. "Virtual Memory Management for Database Systems," *ACM Operating Systems Reviews, vol. 16, no. 4*, pp. 26-48, October 1982.

[WOEL86] Woelk, D., W. Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, Washington D.C., May 1986.

[WOEL87] Woelk, D., and W. Kim. "Multimedia Information Management in an Object-Oriented Database System," in *Proc. Very Large Data Bases*, Brighton, England, Sept. 1987.