Development of an Object-Oriented DBMS

David Maier
Jacob Stein
Allen Otis
Alan Purdy

Technical Report CS/E-86-005
15 April 1986

Revised June, 1988

Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

Presented at the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.

Development of an Object-Oriented DBMS

David Maier
Jacob Stein
Servio Logic Development Corp.
and Oregon Graduate Center

Allen Otis
Alan Purdy
Servio Logic Development Corp.
15025 S.W. Koll Parkway, 1a
Beaverton, Oregon 97006
(503) 644-4242

Abstract

We describe the results of developing the GemStone object-oriented database server, which supports a model of objects similar to that of Smalltalk-80. We begin with a summary of the goals and requirements for the system: an extensible data model that captures behavioral semantics, no artificial bounds on the number or size of database objects, database amenities (concurrency, transactions, recovery, associative access, authorization) and an interactive development environment. Object-oriented languages, Smalltalk in particular, answer some of these requirements. We discuss satisfying the remaining requirements in an object oriented context, and report briefly on the status of the development efforts. This paper is directed at an audience familiar with object-oriented languages and their implementation, but perhaps unacquainted with the difficulties and techniques of database system development. It updates the original report on the project [CM], and expands upon a more recent article [MDP].

1. Introduction

The GemStone database system is the result of a development project started three years ago at Servio. Our goal was to merge object-oriented language concepts with those of database systems. GemStone provides an object-oriented database language called OPAL, which is used for data definition, data manipulation and general computation.

Conventional record-oriented database systems, such as commercial relational systems, often reduce application development time and improve data sharing among applications. However, these DBMSs are subject to the limitations of a finite set of data types and the need to normalize data [Ea, Si]. In contrast, object-oriented languages offer flexible abstract data-typing facilities, and the ability to encapsulate data and operations via the *message metaphor*. Smalltalk-80 is an example of a completely implemented object-oriented system [GR, Kr].

Our premise is that a combination of object-oriented language capabilities with the storage management functions of a traditional data management system will result in a system that offers further reductions in application development efforts. The extensible data-typing facility of the system will facilitate storing information not suited to normalized relations. In addition, we believe that an object-oriented language can be complete enough to handle database design, database access, and applications. Object-like models have long been popular in CAD [CFHL, EM, Ka82, Ka83, LP, MNP, SMF], and seem well suited to support programming environments [PL], knowledge bases [DK], and office information systems [Ah+, Zd84]. Other groups are in the process of implementing object model database systems [DKL, Ni, ZW].

2. Goals and Requirements

2.1. An Extensible Data Model

The system must have a data model that supports the definition of new data types, rather than constraining programmers to use a fixed set of predefined types. New types should also be indistinguishable from system-supplied types for the purposes of application programming: operations that apply to new types should be syntactically similar to the built-in operations on predefined types. The distinction between data types and data structures is important in achieving this goal of extensibility.

Data structures are made up of atomic values (integers, strings, etc.), plus constructors (record, relation, set, tree). A data type is really a collection of operators, the *protocol*, for operating on a particular structure. The underlying structure need not be the same as the appearance provided by the protocol. In conventional database systems, the types correspond to the structures. There is usually a fixed set of operations on atomic values, such as arithmetic and comparison operations. Each constructor has a fixed set of operations; for example, a record constructor has "set field" and "get field", and a relation constructor has "add record", "delete record" and "select record". It is not possible to add new operations that appear syntactically similar to the built-in operations. Thus the set of data types that are directly supported is the same as the set of data structures since nested application of the constructors is not supported.

Our goal is to model the *behavior*, not just the structure, of entities in the real world [Mo]. Further, we must be able to package behavior with structure to create new data types. To get reasonable performance from such a system, the collection of constructors must be rich enough that most data types have fairly direct implementations. In particular, we should be able to capture many-to-many relationships, collections, and sequences

directly. For an easily usable system, we should be able to nest the structuring operations to arbitrary levels, and use previously defined data types as building blocks for other types. GemStone must have system management functions for monitoring system performance, performing backups, recovering from failures, adding and removing users, and altering user privileges.

2.2. Database Amenities

GemStone is first a database system, so it must provide shared access to persistent data in a multi-user environment. It should support stable storage of data objects on disk, while providing location transparency to the application programmer on the movement of objects between main memory and secondary storage. GemStone must provide for ownership of data objects, and requests by the owner to authorize sharing with other users. Each database session should appear to have complete control of a consistent version of the database, even while users are running concurrently, and should be provided with a transaction mechanism to commit or abort a set of changes atomically. Users should be able to request replication of critical data to guard against localized media failure.

GemStone should support auxiliary storage structures that provide alternative access paths to data, and should give users some control over physical grouping of objects, to improve efficiency of specific access patterns. Bounds on the number and size of data objects should be determined only by the amount of secondary storage, not main-memory limitations or artificial restrictions on data definition. Thus, fields in a record should be variable-length, with no fixed upper bound. Collections of objects, such as arrays and sets, should not have a bound on the number of elements. Similarly, the total number of objects in a database system should not be arbitrarily limited. Finally, the system should handle both small and large objects with reasonable efficiency [SSB].

2.3. Programming Environment

We feel that GemStone should provide at least the following tools and features for application development:

1. An interactive interface for defining new database objects, writing OPAL routines, and executing ad hoc queries in OPAL.
2. A windowing package upon which end-user interfaces can be built.
3. A procedural interface to conventional languages, such as C and Pascal.

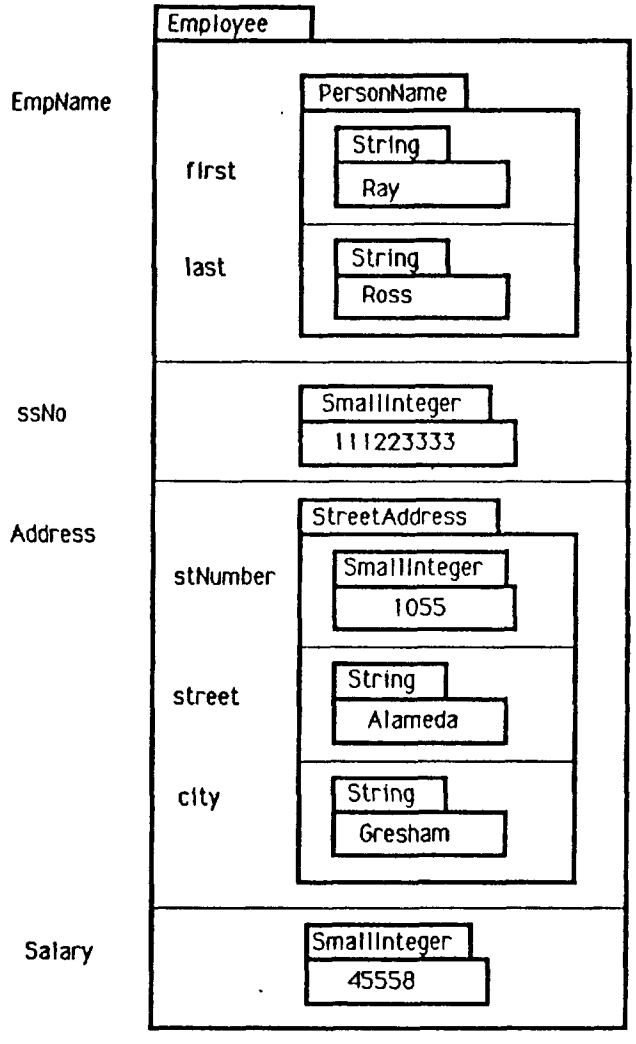
3. Advantages of an Object-Oriented Model

During the research stages of the GemStone project, we developed a mostly declarative query language that was deficient in procedural capabilities. Given the problems with providing procedural extensions and educating the marketplace to a completely new language, we decided to use an existing object-oriented language, Smalltalk-80 [GR], as the basis for product development. We have made extensions to Smalltalk in the areas of associative access support for queries, basic storage structures, typing and support for a multi-user environment. In the following subsections, we cover the advantages of an object-oriented approach as regards modeling and application development.

3.1. Modeling Power

GemStone supports modeling of complex objects and relationships directly and organizes classes of data items into an inheritance hierarchy. A single entity is modeled as a single object, not as multiple tuples spread amongst several relations [HL, JSW, LoP, PKLM]. Properties of entities need not be simple data values, but can be other entities of arbitrary complexity. The address component of an employee object need not be just a text string. In GemStone it can be a structured object, itself having components for street number, street and city, and its own defined behavior. (See Figure 1.) GemStone directly supports set-valued entities, without the encoding required in the relational model. Furthermore, sets can have arbitrary objects as elements, and need not be homogenous. We provide the physical data independence of relational databases without the limitations on modeling power.

FIGURE 1



3.1.1. Object Identity

GemStone supports *object identity* [Ma]. A data object retains its identity through arbitrary changes in its own state. Entities with information in common can be modeled as two objects with a shared subobject containing the common information. Such sharing reduces "update anomalies" that exist in the relational data model. In the relational model, the properties of an entity must be sufficient to distinguish it from all other entities. For one entity to refer to another, there must be some fields that uniquely and immutably identify the other entity. (Some extensions to the relational model incorporate forms of identity [Co, Za].) Uniqueness and immutability are ideals seldom present in the real world. We may choose to refer to departments by name,

but what happens if a department's name changes? (We note that all of our objects are assumed independent. Knowing that one object is owned by, or depends on, another could be useful for storage management [Gr, Ni, We].)

3.1.2. Modeling Behavior

GemStone supports simulation of the *behavior* of real-world entities. Data manipulation commands in conventional systems are oriented towards machine representations: "modify field," "insert tuple," "get next within parent." For an office management system, several applications might reserve a room. In a conventional database system, each application would contain DML (data manipulation language) statements to test for room availability, insert or change a record to indicate the reservation, and perhaps create another record with a reminder to the reserver. Changes to the structure of the database may require locating and modifying every application that makes use of the database. In GemStone, a `reserveRoom` message can be defined that takes a date and a time as parameters, and performs all the necessary checks and updates to the database to reserve a room.

The OPAL method that implements a message can execute any number of database queries and updates, with many advantages. Applications are more concise: sending one message takes the place of many database operations. The code is more reliable, as every application that reserves a room uses exactly the same procedure—the method associated with the `reserveRoom` message. The scope of changes required by alterations to the structure of a type is limited to the methods for the type. Further, messages can protect the integrity of the database, by consistency checks in their methods. If all applications that enter an item in the room reservation list are required to use the `reserveRoom` message, double-booking of rooms can be forestalled.

3.1.3. Classes

The class structure of GemStone speeds application development in several ways. GemStone comes with a large complement of classes implementing frequently used data types. Class definitions are the analogue to schemes in other database systems, but classes also package operations with the structure, to encapsulate behavior. Thus the message definition facilities along with class mechanism meet the requirement of an extensible data model. GemStone includes a hierarchy of classes. Whereas a class helps organize data, the class hierarchy helps organize the classes. The subclassing mechanism allows a database scheme to capture similarities among various classes of entities that are not totally identical in structure or behavior. Subclassing also provides a means to handle special cases without cluttering up the definition of the normal case [MBW].

3.1.4. Associating Types with Objects

Unlike most programming languages that support abstract data types, Smalltalk associates types with values, not the slots holding the values. Typing objects rather than names has liabilities for query processing, which we consider in the next section. We consider some advantages here.

We hope to enable database designers to model application domains they previously may have shied away from because of complexity or lack of regular structure. However, modeling an enterprise for the first time is a much different undertaking than building a database application for an area that has already been modeled, but has not yet been computerized. The basic modeling for financial record keeping was done thousands of years ago. The structure of the information involved is such that it readily fits into standard record-based data models. A development schedule based on scheme definition, application writing, database population and debugging is reasonable. Not so with a CAD task being modeled for the first time, or a database to support an expert system. The application area has not been modeled before, and there will be many iterations of the database scheme before the application is mature [ACO, AO, MP]. Being able to start writing database routines without completely specifying the structure and behavior of every class of entities can be of great advantage. Later, when the model has stabilized, typing can be associated with fields for integrity or efficiency.

By not associating types with variables, unanticipated cases (a company car might be assigned to a department as well as an employee) can be more easily handled. A routine (method) makes assumptions about the protocol of its arguments, not their internal structure. Such routines are robust in the face of new classes. If every object responds to the `printString` message to return a string representing itself, then we can write an OPAL method for `Set` that prints a string representation of all its elements, regardless of their classes.

3.2. A Unified Language

OPAL is much more powerful than standard data manipulation languages. It is computationally complete, with assignment and flow of control constructs. Almost all the computation required in an application can be written within OPAL. This ability helps avoid the problem of *impedance mismatch*, where information must pass between two languages that are semantically and structurally different, such as a declarative data sublanguage and an imperative general-purpose language. GemStone stresses uniformity of access to all system objects and functions, using the same mechanisms as for regular data objects.

4. Turning Smalltalk into a DBMS

Smalltalk is a single-user, memory-based, single-processor system. It does not meet the requirements of a database system. While Smalltalk provides a powerful user interface and many tools for application development, it is oriented to a single user workstation. To meet the requirements of a database system the following enhancements have been added.

4.1. Support of a Multi-User, Disk-Based Environment

Being disk-based does not mean simply paging main memory to disk as it overflows. The database must be intelligent about staging objects between disk and memory. It should try to group objects accessed together onto the same disk pages, try to anticipate which objects in main memory are likely to be used again soon, and organize its query processing to minimize disk traffic.

Since GemStone data is shared by multiple users, the system must provide concurrent access. Each user should see a consistent version of the database, even with other users running simultaneously. Since a user may make changes that are not committed permanently to the database, GemStone must support some notion of multiple workspaces, in which proposed changes to the database can later be discarded or committed. A related requirement is management of multiple name spaces. Smalltalk assumes a single user per image, and so provides a single global name space. Several partially related or unrelated applications can be under development on a single database at one time. It is unreasonable to expect either that users share a single global name space, or, at the other extreme, that user name spaces are disjoint.

Current Smalltalk implementations use a single processor for both display processing and object management. We expect

GemStone to support multiple, interactive applications. Hence, it does not seem wise to use the same processor for secondary storage management as for display processing at the end-user interface. We felt that the storage-management and user-interface functions in GemStone must be decoupled to run as separate processes on separate processors.

4.2. Data Integrity

Various kinds of failures (program, processor, media) and violations (consistency, access, typing) can compromise the validity and integrity of a database. A database system must be able to cope with failure by restoring the database to a consistent state while minimizing the amount of computation lost. It must also prevent violations from occurring.

By program failure we mean that an application program may fail to complete, say, because of a run-time error. If the program fails after some updates to the database have been made, the database can be left in an unexpected state. Database systems provide for multiple updates to be performed atomically (in an all-or-nothing manner) through the use of transactions. A transaction is used to mark a section of processing so that all its changes are made permanent (the transaction commits), or none are made permanent (the transaction aborts).

By processor failure, we mean that the processor handling GemStone storage management fails. For such failures, the database must be kept intact. Recovering from program and processor failure imply that master copies of objects on secondary storage must be updated carefully. Additionally, a good database system should be robust enough to tolerate additional failures during the recovery period.

By media failure we mean that damage or flaws in the secondary storage devices may cause committed data to be lost. No strategy can provide complete protection against media failure. We wanted GemStone to provide for both periodic backup and dynamic replication of sensitive information. By dynamic replication, we mean keeping multiple, on-line copies of a database, all of which are updated on every transaction.

Turning to violations, database consistency can be violated if transactions from multiple users interleave their updates. GemStone must support serializability of transactions: the net effect of concurrent transactions on the database must be equivalent to some serial execution of those transactions. The integrity of a database can also be violated if a user accesses data that he or she should not be permitted to see. In Smalltalk, all objects are available to the user. Gemstone must assign unique ownership to every object, and give the owner of an object to power to grant access to others.

Integrity constraints, such as keys and referential integrity, are assertions that a priori exclude certain states of the database. It is always a judgment call whether the database system should check constraints after each transaction, or whether the application programmer should be responsible for preserving consistency in each transaction. The former course is more reliable, but almost always more expensive. At a minimum, the database should support constraints that require subparts of an entity or collections to belong to a certain class. We note that referential integrity comes "for free" in GemStone. One object refers directly to another object, not to a name for that object. The reference cannot be created if the other object does not exist. Hence, there are no dangling references.

4.3. Large Object Space

Gemstone must store both large numbers of objects and objects that are large in size. The first Smalltalk-80 implementations had a limit of 2^{15} objects, 2^{15} instance variables in any object, and 2^{30} total words of object memory [GR]. More recent implementations raise these limits, but still use the same techniques to represent and manage objects [KK]. Large disk-based objects require new storage techniques. Some objects will be too large to fit in main memory, and must be paged in.

While virtual-memory implementations page large objects, we felt we must get away from linear representations of long objects. Requiring objects that span disk pages to be laid out contiguously in secondary storage (or even virtual memory) will lead to unacceptable fragmentation or expensive compaction passes. In Smalltalk, to "grow" an object, such as an array, a new, larger object is created and the contents of the smaller object are copied into it. We want the time required to update or extend an object to be proportional to the size of the update or extension, *not* to the size of the object being updated. We also felt that Smalltalk's repertoire of basic storage representations was inadequate for supporting large unordered collections. Having to map such a collection into an ordered underlying representation imposes artificial restrictions. Thus, GemStone needs a basic storage type for unordered collections.

Finally, searching a long collection by a sequential scan will give unacceptable performance with a disk-based object. Searching for elements should be at most logarithmic in the size of the collection, rather than linear. Thus, GemStone should support associative access on elements of large collections: It should supply storage representations and auxiliary structures to support locating an element by its internal state. This requirement reinforces the need for typing on collections and instance variables. To index a collection `E` of employees on the value of the `salary` instance variable, the system need assurances that every element in `E` has a `salary` entry. Furthermore, if that index is to support range queries on `salary`, the systems needs a declaration that all `salary` values will be comparable according to some total order.

Along with storage-level support for associative access, OPAL must have language constructs that allow associative access.

4.4. Physical Storage Management

GemStone must provide features for managing the physical placement of objects on disk. Smalltalk is a memory-resident system, and so there is not much need to say where an object goes. The database administrator, or a savvy application programmer, should be able to hint to GemStone that certain objects are often used together, and so should be clustered on the disk. The administrator should be able to take objects off line, say for archiving, and bring them back on line later. Finally, as objects are never explicitly deleted, the system will be responsible for reclaiming the space used by unreferenced objects. (An alternative is to assume that a permanent object is never deleted, and that objects not referenced in the current state of the database should be shifted to archival storage.)

4.5. Access From Other Systems

While OPAL goes much further than conventional database languages in providing a single language for database application programming, we wanted to concentrate our initial efforts on storage management issues, rather than user interfaces. Thus, GemStone provides for access to its facilities from other programming languages. We want to support an application development environment for OPAL along the lines of the Smalltalk programming environment [GR], but we recognize that the application development environment may not be the same as the environment in which the finished application runs. However, we are committed to providing procedural interfaces to C and Pascal.

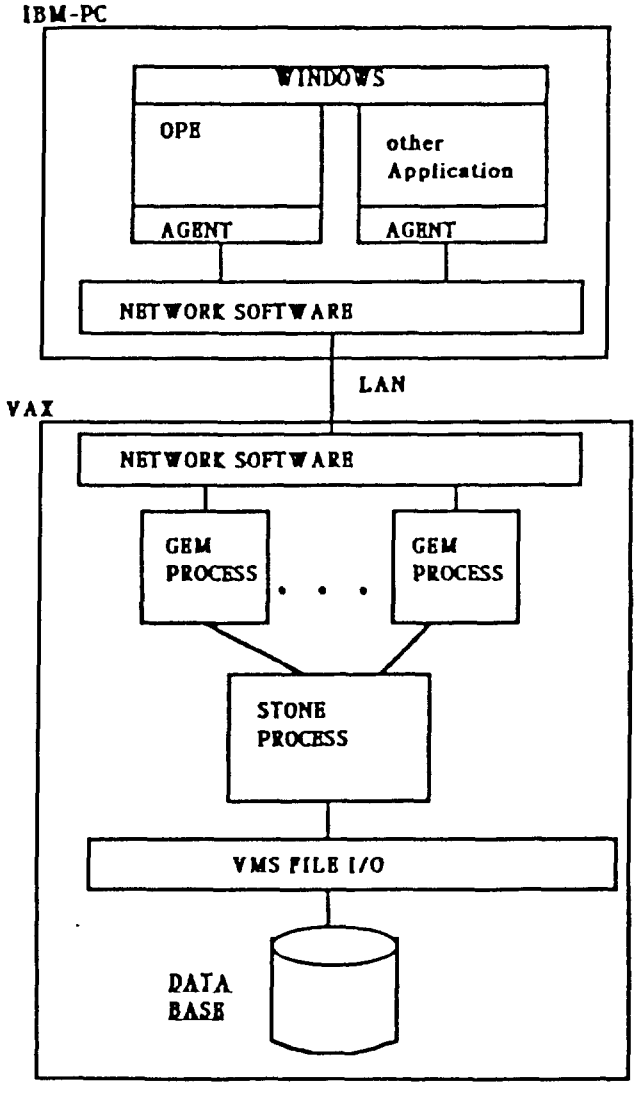
5. Our Approach

This section addresses how we provided the enhancements needed to Smalltalk to make it a database system. We start with an overview of the architecture of GemStone.

5.1. GemStone Architecture

Figure 2 shows the major pieces of the GemStone system. Stone and Gem correspond roughly to the object memory and the virtual machine of the standard Smalltalk implementation [GR]. Stone provides secondary storage management, concurrency control, authorization, transactions, recovery, and support for associative access. Stone also manages workspaces for active sessions. Stone uses unique surrogates, called *object-oriented pointers* (OOPs) to refer to objects. Stone uses an *object table* to map an OOP to a physical location. This level of indirection means that objects can easily be moved in memory. While the object table can potentially have 2^{31} entries, we expect that the portion for objects currently in use by various sessions is small enough to fit in main memory. Stone is built upon the underlying VMS file system. The data model that Stone provides is simpler than the full GemStone model, and provides only operators for structural update and access. An object may be stored separately from its subobjects, but the oops for the values of an object's instance variables are grouped together. Others have considered decomposed representations of objects [Ch+, CK].

FIGURE 2



Gem sits atop Stone, and elaborates Stone's storage model into the full GemStone model. Gem also adds the capabilities of compiling OPAL methods into bytecodes and executing that code, user authentication, and session control. (OPAL bytecodes are similar, but not identical, to the bytecodes used in Smalltalk.) Part of the Gem layer is the *virtual image*: the collection of OPAL classes, methods and objects that are supplied with every GemStone system.

Figure 3 shows the class hierarchy in the current GemStone virtual image. Comparing it to the Smalltalk hierarchy, we have removed classes for file access, communication, screen manipulation and the programming environment. The file classes are unnecessary, as we have persistent storage for all GemStone objects. Computation for screen manipulation needs to happen near the end user, and needs fast bytecode execution. GemStone is optimized toward maintaining large numbers of persistent objects, rather than fast bytecode execution. The programming environment classes are replaced by a browser application that runs on top of GemStone, which we describe in a later subsection. We have added classes and methods to make the data management functions of transaction control, accounting, ownership, authorization, replication, user profiles and index creation controllable from within OPAL.

FIGURE 3

- Object
  - Association
    - SymbolAssociation
  - Behavior
    - Class
    - Metaclass
  - Boolean
  - Collection
    - SequenceableCollection
      - Array
        - InvariantArray
        - Repository
      - String
        - InvariantString
          - Symbol
    - Bag
      - Set
        - Dictionary
          - SymbolDictionary
            - LanguageDictionary
        - SymbolSet
        - UserProfileSet
  - CompiledMethod
  - Magnitude
    - Character
    - DateTime
    - Number
      - Float
      - Integer
        - SmallInteger
  - MethodContext
    - Block
      - SelectionBlock
  - Segment
  - Stream
    - PositionableStream
      - ReadStream
      - WriteStream
  - System
  - UndefinedObject
  - UserProfile

The Agent interface is a set of routines to facilitate communication from other programs in other languages running on processors (possibly) remote from Gem. The Agent interface currently supports calls from C and Pascal programs running on an IBM-PC for session and transaction control, sending messages to GemStone objects, executing a sequence of OPAL statements, compiling OPAL methods, and error explanation. In addition, the Agent provides "structural access" calls, which perform the following functions:

1. determining an object's size, class, and implementation;
2. inspecting a class-defining object;
3. fetching bytes or pointers from an object;
4. storing bytes or pointers in an object;
5. creating objects.

Information passes between the Agent and Gem in the form of bytes and GemStone object pointers. Certain objects have predefined object pointers, such as instances of `Boolean`, `Character` and `SmallInteger`. Instances of `Float` and `String` are passed as byte sequences. Instances of other classes must be decomposed into instances of the classes mentioned, in order to pass their internal structure between the Agent and Gem. However, the identity of any object can be passed between the Agent and Gem, regardless of its complexity. The idea is to do the computation and manipulation of objects in Gem, and only pass data used for display through the Agent to the interface routines.

Gem, Stone and the Agent interface are structured as separate processes. Our current mapping of processes to processors has Gem and Stone running on on a VAX under VMS. The Agent interface supports communication with Gem from an IBM PC. While a GemStone system has a single Stone process, it maintains a separate Gem process for each active user, and the Agent interface handles communication on a per-application basis.

5.2. Multiple Users

Stone supports multiple concurrent users by providing each user session with a workspace that contains a *shadow copy* of the object table derived from the most recent committed object table, called the *shared table*. Whenever a session modifies an object, a new copy of that object is created, and placed on a page that is inaccessible to other sessions. The shadow copy of the object table is updated to have the object's OOP map into the new page.

Conceptually, the shadow object table for a workspace is a complete copy of the version of the shared table when the session starts. Actually, we do not make a copy all at once. Object tables are represented as B-trees, indexed on OOPs. For a shadow object table, we need only copy the top node of the committed object table. As the objects are changed by a session, the shadow object table adds new nodes that are copies of its shared object table with the proper changes. Figure 4 shows the state of a shadow object table after the alteration of a single object. Multiple paths have been copied since several objects may have been on the same page as the altered object.

We chose an optimistic concurrency control scheme: one in which access conflicts are checked at commit time, rather than prevented from occuring through locking. For each transaction, Stone keeps track of which objects the transaction has read or written. At commit time, Stone checks for read-write and write-write conflicts with transactions that have committed since the time the transaction began. If there are no conflicts, the transaction is allowed to commit. For a commit, the shadow object table of the session is treated as if it were *transparent* on the entries that have not been modified, and is overlaid on the most recent version of the shared table. Thus, only entries in the shared table for objects that have been copied by the session are changed. In this way, the changes made by the committing session are merged with those of other transactions that committed after the committing session began. If the current transaction conflicts with a previously committed transaction (or is aborted by the application), the changes in its shadow table are discarded, after using the table to reclaim pages used for new copies of objects.

This optimistic scheme ensures that read-only transactions never conflict with other transactions. Such a transaction gets a consistent copy of the database state, does its reading, and has no changes to make to the shared table on commit. Only transactions that write can conflict with each other. This scheme never deadlocks, as a session experiences no contention with other sessions before a commit point. However, it is possible that an application that writes a large portion of the database may fail to commit any transactions for an arbitrarily long time. While shadowing has had some bad press, it seems a natural approach to us, given that we have an object table. It avoids some extra reads, makes commit and abort simple, and is a excellent candidate for write-once memory, since active pages are never changed in place.

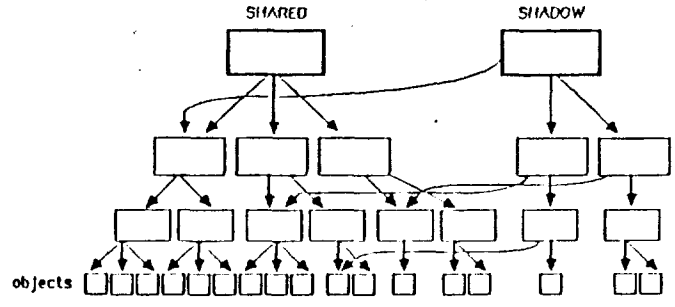


FIGURE 4

5.3. Efficiency Considerations

One problem with recording all the objects a session reads or writes is that the list can grow quite long, and GemStone will spend a lot of time adding entries to such lists. One optimization is that certain classes of objects, such as `SmallInteger`, `Character`, and `Boolean` are known to contain only instances that cannot be udpated. Thus, such objects need not be recorded for concurrency control. Even excluding these objects, single objects are just too fine a granularity for concurrency control. Thus, we introduced the notion of *segments*, which are logical groupings of objects that are the unit of concurrency control in GemStone, much like the segments of the ADAPLEX LDM [CFLR]. A segment may contain any number of objects. GemStone keeps a list of just the segments read or written by a session, rather than all objects. Also, at a physical level, pages respect segment boundaries. Thus the practice of copying all objects in a page when one is changed causes no additional conflicts. Segments are visible from within OPAL through the class `Segment`. Users can control placement of objects in segments, to group objects to try to avoid conflict. If an application has a group of private objects, all those objects and no others can be placed together in one segment. At the system level, system objects that are shared by many users, but are almost never updated (such as the class describing object for a system class)

can be placed on a single segment, so that accesses to them never causes a conflict.

5.4. Name Spaces

Multiple name spaces are managed by Gem. The virtual image has a class `UserProfile` that is used to represent properties of each user that are of interest to the system, such as user Id, password, native language and local time zone. A `UserProfile` object also contains a list of dictionaries that are used to resolve symbols when compiling OPAL code for that user. When an identifier is encountered in OPAL code, and that identifier is neither an instance variable nor a class variable, the dictionaries are searched in order to find an object corresponding to that identifier. There may be any number of dictionaries for a user, to accommodate various degrees of sharing. For example, a programmer's first dictionary may contain objects and classes for his or her portion of a project, the second may be for objects shared with other programmers working on the same project, and the third could contain system objects. Note that symbol resolution can be performed at runtime, thus providing for dynamic symbol resolution during method execution.

5.5. Transactions and Recovery

Most of our approach to transactions has been covered in a previous section. To reiterate, every session gets a shadow copy of the shared object table when it begins, and installs its shadow copy as the shared copy when it successfully commits a transaction. Further, a session always writes changed and new objects into pages that are not accessible to any other transaction before commit time. Thus, aborting a transaction means throwing away its shadow object table, and committing means replacing the shared table with a shadow copy. The only issue that needs more elaboration is atomicity—that the changes of a transaction are made, seemingly, all at once. As object tables are trees, atomicity is not hard to provide. When a shadow table is to replace the shared table, and the shadow table differs from the table it is about to replace, the new table can replace the old by simply overwriting the root of the shared object table with the root of the new object table. (Actually, there is a "root of the database" above the root of the object table that gets overwritten. The database root references some other information besides the object table, such as a list of active transactions.) Rewriting the root is the only place where any part of the shared copy of the database is overwritten.

Recovery from processor failure does not require a great amount of additional mechanism over what we have for concurrency control. Our unit of recovery is a transaction. Changes made by committed transactions are kept, changes not yet committed are lost. Since the shared version of the database is never overwritten, we need almost nothing in the way of logs to bring the database to a consistent state, since it never leaves one. The only tricky part is a processor crash while writing a new database root. To handle that eventuality, we keep two copies of the root, which reside at a known place. To restore a consistent state of the database after the crash, we simply check those two pages. If they are different, we copy one that can be determined to be uncorrupted over the other. The real work on recovery is garbage collection: removing detritus of the transactions that had not committed before the crash.

To guard against media failure, we have introduced a structure called a *repository*. A repository is the unit of replication, and also the unit of storage that can be taken off line. Most of what we said before about the database actually pertains to repositories. Segments partition repositories, and all the objects in a segment are stored in the segment's repository. A repository may be taken off line, which means all its objects become inaccessible. `Repository` is an OPAL class providing internal representatives of repositories. A `Repository` instance can respond to a message `replicate`, which means two copies of the repository will be maintained (at increased cost in time and space), usually on separate external devices. The copies know about each other, and if the medium for one fails, the other is still available.

5.6. Authorization

Segments are also the unit of ownership and authorization. Every user has at least one segment, and when he or she creates new objects, they go in an owned segment. A user may grant read or write permission (write implies read) on a segment to other users or groups of users. Such grants must always come from the original owner. Read or write permission on a segment implies the same permission on all objects assigned to the segment. User identification is handled by Gem, using `userId` and `password` from a `UserProfile`.

There are some subtleties of read and write permission in an object model. First, having the identity of an object (its OOP) is not the same as reading the object. Second, having permission on an object does not imply have permission on all its subobjects. So, for example, an `Employee` object, along with the objects that are values for instance variables `empName`, `ssNo` and `address` could reside in one segment. By putting a `SalaryHistory` object in another segment, authorization can be granted to just a portion of an employee's personnel information. Third, name spaces are the first line of defense against unauthorized access. If a user cannot find an object, he or she cannot read the object.

5.7. Large Object Space

In designing GemStone, we have tried to always set limits on object numbers and sizes so that physical storage limits will be encountered first. A GemStone system can support 2^{31} objects (2^{31} counting instances of `SmallInteger`) and an object can have up to 2^{31} instance variables. Segments have no upper bound on the number of objects they can contain, other than the number of objects in the system.

When an object is larger than a page, the object is broken into pieces and organized as a tree spanning several pages. To handle large unordered objects (instances of `Bag` and its subclasses), we have added a new basic storage structure called a *non-sequenceable collection* (NSC). This structure supports adding, removing and testing for membership, along with iteration over all the elements. However, NSCs have *anonymous instance variables*, which means their component objects may not be referenced by name or index. Large NSCs are also stored as trees, but ordered by OOP. In the next section, we show how content-based retrieval is supported for an NSC object.

A large object can be accessed and updated without bringing all the pages of an object into a workspace. The tree structure for large objects makes it possible to update pieces of them without rewriting the whole object, much as for the object table. Since pages of a large object need not be contiguous in secondary storage, such objects can grow and shrink with no need to recopy the entire object.

5.8. Associative Access and Typing

We briefly cover some of the language and typing issues relating to associative access, along with index structures and their maintenance. The fundamental language issue is being able to detect opportunities for using auxiliary storage structures

In a computationally complete language such as OPAL, it is neither necessary or desireable to consider using auxiliary structures for every database manipulation. Conceivably, we could analyze all OPAL methods to detect places where alternative access paths might be used. We felt that approach was too complex, and instead decided that the programmer must flag opportunities to use auxiliary structures.

OPAL supports the use of indices to speed the evaluation of expressions of the form

```
aBag select: aBlock
```

The block has one variable and returns a `Boolean`. The result of the expression is the subset of elements of `aBag` for which `aBlock` returns `true`, and resembles the relational selection operator in the Cypress data model [Ca]. This statement is evaluable in OPAL without indices, but at the cost of examining every element in `aBag`. Since a block can contain arbitrary OPAL expressions, indices are not useful in evaluating every expression in a blocck. Hence, for use with indices, we added *path syntax* to the OPAL language. For any variable, we can append to it a path composed of a sequence of pieces called *links*, which specify some subpart of an object. For example, **anEmp.empName.last** might access the last name of an `Employee` object. A question arises why sequences of unary messages do not suffice to the same thing, such as **anEmp name last** The reason is that we want to support associative access at runtime without performing message sends, and so the support can come from the Stone level.

A selection block for associative access can contain an conjunction of *path comparisons*, where a path comparison an expression of the form <path expression> <comparator> <literal> or <path expression> <comparator> <path expression>:

```
anEmp.empName.last = 'Sanders'
anEmp.salary > anEmp.dept.manager.salary
```

Index use is requested by using set braces in place of brackets around the block in a `select:` message

```
empSet select:
  {:anEmp | anEmp.empName.lastName = 'Sanders'}
```

rather than

```
empSet select:
  [:anEmp | anEmp.empName.lastName = 'Sanders']
```

The two expressions give the same result, but the first one requests OPAL to use an index if available, while the second will *always* be evaluated by iterating through `empSet`. If no appropriate index exists, then the first expression might still be evaluated without the use of message sends if, as discussed below, the path is appropriately typed. Otherwise, the first expression evaluates using the same method as the second. We found it a great help in testing associative access processing to have a brute-force way to evaluate selection queries, as a kind of a "semantic benchmark" for checking index-based evaluation.

Another central decision in designing associative access was what to index, classes or collections? Many applications may use instances of the same class, and store them in different collections (like having several relations on the same scheme [Ha]). Indexing on the class means that applications that do not use the index still bear the overhead for instances they use being in the index. Further, a classwide index presents authorization problems. No one user may have read access to set all the objects in the class, so no one is able to request the index be created. Also, indexing a collection allows the possibility that instances of subclasses be included in a collection that is indexed. Indexing on a class basis makes it easier to trace changes to the state of an object that could cause the object to be positioned differently within an index. We decided that minimizing cost for programs not using an index was the top priority, so we index on collections, but other systems have chosen class indexing [ZW]. Additionally, if indexing by class, intersecting the result of a lookup with a collection may be time consuming with respect to the size of the collection. Note that a class can be implemented to keep a collection of all instances if desired, and that collection can be indexed.

Indices are created and abandoned by sending messages to a `Bag` or `Set` object, giving the path to index. For example, if `empSet` is a set of `Employee` objects, we can request an index on `empName.last`. There are two kinds of indices: *identity* and *equality*. An identity index supports searching a collection on the identity of some subobject of one of the elements, without reference to the subobject's state. An equality index supports lookup on the basis of the value or internal state of objects, and range searches on values.

The path syntax for associative selections and the kind of index desired dictate what typing information is required to support indexing. Referring back to the discussion in Section 4.3, to have an identity index on a collection using a particular path, we must know that the path expression is defined (leads somewhere) for every object in the collection. For an equality index, we must additionally know that the values of the paths for every element of the collection are comparable with respect to equality and the other comparisons supported by equality indices. OPAL provides typing for names and anonymous instance variables. For any named instance variable, the value of that variable can be constrained to be a *kind* of a given class. A value is a kind of a class if it is an instance of that class or of some subclass thereof. For example, we can declare that the `empName` instance variable of class `Employee` must have kind `PersonName`, which means the value can be a `PersonName` instance, or an instance of some subclass, say `TitledPersonName`. Subclasses of `Bag` and `Set` can be restricted in the kind of elements their instances may contain.

Both named and anonymous instance variable typing are inherited through the class hierarchy. Additionally, typing can be further restricted in a class's subclasses. If in `Employee` class instance variable `empName` is constrained to `PersonName`, then in a subclass of `Employee` `empName` can be constrained to `TitledPersonName`.

In order to create an identity index into `empSet` on `empname.last`, variable `empName` must be constrained to a class in which a variable `last` is defined. However, it would not be necessary for `last` to be be constrained within that class, for the comparisons supported by identity indices need not know the structure of employee's last names. In an equality index, employee's last names would need to be constrained to a class whose instances are totally ordered, in order to provide the comparisons supported by equality indices. The constraint on the last link of a path upon which an equality index is built is restricted to `Boolean`, `Character`, `DateTime`, `Float`, `String`, `Smallinteger` or subclasses thereof. For `Boolean`, `Character` and `SmallInteger`, there is no dis-

tinction between equality and identity indices.

In addition to supporting indexed associative access, typing of names can be used as an integrity constraint on named variables. In particular, it can be used to assure that the value of a named instance variable will understand a given protocol in all objects that are a kind of a given class.

Indices are implemented as a sequence of index components, one for each link in the path upon which the index is built. Each component is implemented as a B-tree. An index on `empName.last` into `empSet` would have two components, one from names to employees in `empSet` and one from last names to names of employees. Common prefixes of indexed paths share the index components that correspond to the common prefix. For example, an index on `empName.first` would share the index component from names to employees with the index on `empName.last`. Additionally, identity indices are implicitly created on the path prefixes of an indexed path. Creating an index on `empName.last` implicitly creates an identity index on `empName`.

Indexing is discussed in further detail elsewhere [MS]. We here mention that the maintenance of indices is a problem that is related to that of maintaining *referential integrity constraints* [Da] in relational databases, and that Stone manages concurrent access to indexing structures since indices are maintained in object space.

5.9. Garbage Collection

Gem uses reachability information to remove temporary objects before a transaction commits. Objects that have been created during the current transaction and can not be accessed transitively from the current state of some object present in the shared object table are temporary. Permanent objects, those that have previously been committed, are garbage collected offline using a mark-sweep algorithm. We believe this preferable to reference counting in that reference counting would require accessing an object every time a reference to it is added or removed.

5.10. Access From Other Systems

GemStone does not provide direct access to a human interface through OPAL. Applications manage the human interface through C modules running on a PC. These modules have complete access to OPAL through the *Gemstone C Interface* (GCI), which is the interface to the agent. The role of the GCI in application development is depicted in Figure 5.

The *OPAL Programming Environment* (OPE) is a collection of Microsoft Windows-compatible applications for the creation of OPAL classes and methods. In addition, the OPE provides applications for the bulk loading and dumping of GemStone databases. The development of a GemStone application consists of two parts: first creating Gemstone classes and methods through the OPE and then developing a human interface on a PC that accesses OPAL through the GCI.

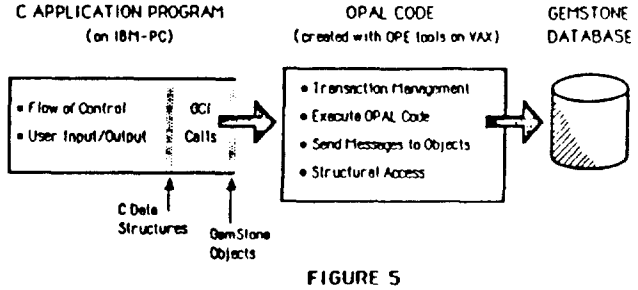


FIGURE 5

6. Future Development Plans

As with all computer-based systems, performance efficiency is a perennial concern. We have two approaches to addressing GemStone's performance efficiency: improving the execution model, and improving database functions. Research of Smalltalk virtual machines has demonstrated several techniques for improving their efficiency [Kr]. Database systems spend most of their time in searching and sorting tasks. These functions can be improved by better algorithms and better buffer management.

We also plan to add several features to GemStone. We plan to add multiple repositories to allow users to dismount a portion of the database and either transport it to another site or preserve it offline. Support for distributed databases will allow multiple sites to share a collection of geographically distributed data. As our user population increases, we expect users to need increased performance efficiency in certain new classes. To support this need, we expect to add selected new classes to the set of predefined classes. GemStone currently supports only the IBM-PC workstation. We plan to offer interfaces to additional workstations such as Lisp and Smalltalk systems.

7. Acknowledgements

The authors would like to thank the following people for their contributions to the GemStone project: Ken Almond, quality and change control; Robert Bretl, Stone object manager; John Bruno, browser; Maureen Drury, virtual image; Jack Falk, documentation; Lynn Gallinat, virtual image; Larry Male, OPE editor; Daniel Moss, PC/VAX communications; Bruce Schuchardt, OPE implementation, bulk loader and dumper; Harold Willams, Gem implementation; Monty Williams, quality and support implementation; Mike Nastos, documentation and bug hunting; Rick Nelson, VAX system manager; D. Jason Penney, Stone implementation, process scheduler; Mun Tuck Yap, Gem object manager, PC/VAX communications.

8. Bibliography and Trademarks

[Ah+] Ahlsen, M., A. Bjornerstedt, S. Britts, C. Hutten, and L. Suderland, An architecture for object management in OIS, *ACM TOOIS* 2:3, July 1983.

[ACO] Albano, A., L. Cardelli, and R. Orsini, Galileo: a strongly-typed interactive conceptual language, *ACM TODS* 10:2, June 1985.

[AO] Albano, A., and R. Orsini, A prototyping approach to database applications development, *Database Engineering* 7:4, December 1984.

[Ca] Catell, R.G., Design and implementation of a relationship-entity-datum model, Xerox CSL 83-4, May 1983.

[Ch+] Chan, A., A. Danberg, S. Fox, W.-T. K. Lin, A. Nori, and D. Ries, Storage and access structures to support a semantic data model, *Proc. Conference on Very Large Databases*, September 1982.

[CFLR] Chan, A., S.A. Fox, W.-T. K. Lin, and D. Ries, Design of an ADA compatible local database manager (LDM), TR CCA 81-09, Computer Corporation of America, November 1981.

[CFHL] Chu, K.C., J.P. Fishburn, P. Honeyman, and Y.E. Liem, Vdd - a VLSI design database, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Co] Codd, E.F., Extending the database relational model to capture more meaning, *ACM TODS* 4:4, December 1979,

[CK] Copeland, G., and S.N. Koshafian, A decomposition storage model, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1985.

[CM] Copeland, G., and D. Maier, Making Smalltalk a database system, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1984.

[Da] Date, C.J., *An Intoduction to Database Systems, Volume 2*, Addison-Wesley, 1983.

[DKL] Derret, N., W. Kent, and P. Lynbaek, Some aspects of operations in an object-oriented database, *Database Engineering* 8:4, December 1985.

[DK] Dolk, D.R., and B.R. Konsynski, Knowledge representation for model management systems. *IEEE Transactions on Software Engineering*, 10:6, November 1984.

[Ea] Eastman, C.M., System facilities for CAD databases, *Proc. IEEE 17th Design Automation Conference*, June 1980.

[EM] Emond, J.C., and G. Marechad, Experience in building ARCADE, a computer aided design system based on a relational DBMS, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

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

[Gr] Gray, M., Databases for computer-aided design, In *New Applications of Databases*, G.Garadarin, E. Gelenbe eds., Academic Press, 1984.

[Ha] Haynie, M.N., The relational/network hybrid data model for design automation databases, *Proc. IEEE 18th Design Automation Conference*, 1981.

[HJ] Hewitt, C., and P. de Jong, Open systems, In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, M.L. Brodie, J. Mylopoulos, J.W. Schmidt eds., Springer-Verlag, 1984.

[HL] Haskin, R.L., and R.A. Lorie, On extending the functions of a relational database system, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1982.

[JSW] Johnson, H.R., J.E. Schweitzer, and E.R. Warkentire, A DBMS facility for handling structural engineering entities, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[KK] Kaehler, T., and G. Krasner, LOOM - large object-oriented memory for Smalltalk-80 systems, In [Kr].

[Ka82] Katz, R.H., A database approach for managing VLSI design data, *Proc. IEEE 9th Design Automation Conferenceaf1*, 1982.

[Ka83] Katz, R.H., Managing the chip design database, *IEEE Computer*, 16:12, December 1983.

[Kr] Krasner, G., *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[LP] La Croix, M., and A. Pirotte, Data structures for CAD object description, *Proc. IEEE 18th Design Automation Conference*, 1981.

[LoP] Lorie, R., and W. Plouffe, Complex objects and their use in design transactions, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Ma] MacLennan, B.J., A view of object oriented programming, Naval Postgraduate School NPS52-83-001, February 1983.

[MOP] Maier, D., A. Otis, and A. Purdy, Object-oriented database development at Servio Logic, *Database Engineering* 18:4, December 1985.

[MP] Maier, D., and D. Price, Data model requirements for engineering applications, *Proc. International Workshop on Expert Database Systems*, 1984

[MS] Maier, D., and J. Stein, Indexing in an object-oriented DBMS, manuscript in preparation.

[MNP] McLeod, D., K. Narayanaswamy, and K.V. Bapa Rao, An approach to infomation management for CAD/VLSI applications, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Mo] Morgenstern, M., Active Databases as a paradign for enhanced computing environments, *Proc. Conference on Very Large Databases*,1983.

[MBW] Mylopoulos, J., P.A. Bernstein, and H.K.T. Wong, A language facility for designing database-intensive applications, *ACM TODS* 5:2, June 1980.

[Ni] Nierstrasz, O.M., Hybrid: a unified object-oriented system, *Database Engineering* 8:4, December 1985.

[PKLM] Plouffe, W., Kim, W., R. Lorie, and D. McNabb, A database system for engineering design, *Database Engineering* 7:2, June 1984.

[PL] Powell, M.L., and M.A. Linton, Database support for programming environments, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Si] Siddle, T.W., Weaknesses of commercial data base management systems in engineering applications, *Proc. IEEE 17th Design Automation Conference*, June 1980.

[SMF] Spooner, D.L., M.A. Milican, and D.B. Fatz, Modelling mechanical CAD data with data abstractions and object-oriented techniques, *Proc. 2nd International Conference on Data Engineering*, February 1986.

[SSB] Stemple, D., T. Sheard, and R. Bunker, Abstract data types in databses: specification manipulation and access, *Proc. 2nd International Conference on Data Engineering*, February 1986.

[We] Weisner, S.P., An object-oriented protocol for managing data, *Database Engineering*, 8:4, December 1985.

[Za] Zaniolo, C., The database language GEM, *Proc. ACM/SIGMOD International Conference on the Management of Data*, May 1983.

[Zd84] Zdonik, S.B., Object management systems concepts, *Proc. ACM SIGOA Conference on Office Information Systems*, 1984.

[Zd85] Zdonik, S.B., Object management systems for design environments, *Database Engineering* 8:4, December 1985.

[ZW] Zdonik, S.B., and P. Wegner, Towards object-oriented database environments, Brown Univeristy TR, 1985.

Trademarks