Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface*

J. Eliot B. Moss *Steven Sinofsky*†

COINS Technical Report 88-67
July 1988

Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst

This report is a substantial revision of and supersedes COINS TR 88-30. This version will also appear in the proceedings of the Second International Workshop on Object Oriented Data Bases; Germany; September, 1988.

Abstract

We investigate issues of integrating object-oriented languages and database features. We provide criteria for database/language support in the context of design applications, and consider the advantages of integration. We discuss the design of Mneme, a system that narrows the gap between object-oriented databases and programming languages, both traditional and object-oriented. Mneme includes facilities for sharing, reliability, and clustering of objects, as well as a high degree of flexibility. We compare Mneme with other current and recent research efforts, describe its particular contributions, and present relevant aspects of the design of the first Mneme prototype.

---

*This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, and by Digital Equipment Corporation, Apple Computer, Inc., GTE Laboratories, and the Eastman Kodak Company.

†Authors' address: Department of Computer and Information Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, 01003; telephone (413) 545-4206; Internet: Moss@cs.umass.edu and Sinofsky@cs.umass.edu.

1 Introduction

In this paper we investigate some of the issues that arise when attempting to integrate object-oriented languages and database features. We consider the advantages over more traditional database systems of such integration in the context of design applications such as software development environments. We describe the design of a system, *Mneme* (NEE-mee, the Greek word for memory), that will narrow the gap between programming language data structures and databases. We further describe the advantages and contributions of Mneme, and compare it with related systems.

The presentation is organized as follows. This introduction describes our motivation, the general issues with which we are concerned, and the specific goals we have for the Mneme system. The next section gives an overview of the concepts Mneme provides to clients. This is followed by a discussion of the internal design for the initial Mneme prototype. We then compare Mneme (as conceived more than as currently implemented) with some other well known current and recent efforts, and conclude, summarizing Mneme's advantages and indicating directions we intend to pursue in the future.

1.1 Motivation

We wish to address the needs of the programmer working on large, complex design applications. Such applications require the cooperative use of shared, distributed, persistent data in a production setting. Applications that would benefit from improved integration include computer aided design (CAD), office automation (OA), and document production, in addition to software development environments (SDEs). An important aspect of these applications is that they demand high functionality and high performance simultaneously.

We believe that integrating programming languages and databases has the promise of meeting the challenge of high functionality and high performance, while also better addressing issues of construction and evolution of application programs by making those programs easier to write. There has been a great deal of work describing and addressing the problems of integrating programming language and database technologies [Buneman, 1984; Atkinson *et al.*, 1984]. Primarily we are concerned with providing truly seamless integration, with database operations as transparent as possible. Traditionally the user of database systems is confronted with the problem of translating ideas first to a programming language model and then to a database language model. If the programming language and database language are tightly integrated, then the user need only be concerned with translating ideas into a single language model. We are also concerned with providing persistent data as described in [Atkinson *et al.*, 1981] and [Atkinson *et al.*, 1984]. Persistence should be an orthogonal property of all existing data structures. In addition, the resulting language should be data type complete: all types must have the same rights and privileges.

Object-oriented languages (OOLs) provide a natural starting point for adding database features to programming languages [Bloom and Zdonik, 1987]. OOLs focus on the structure of data and the operations permitted on data, in other words, the behavior of data. In

contrast, object-oriented databases (OODBs) focus on persistent data. Together object-oriented languages and object-oriented databases provide superior support for a wide range of applications. OOLs and OODBs complement each other well, and tight integration of the two will provide the advantages of the object-oriented approach along with persistence. This will considerably simplify programming complex design applications.

While an object-oriented persistent programming language exhibits most of the desired features, there are a number of problems that persistence and object-orientation do not address. As will be discussed, Mneme additionally tackles issues of very large collections of objects, sharing and cooperation, reliability, distributed systems, support for more than one programming language, and support for multiple, heterogeneous data storage servers.

1.2 General Issues

When attempting to integrate traditional database functionality with programming languages for use in design applications, a number of issues must be addressed by any system. Here we indicate our position with respect to some of the most important issues.

Small Objects. In an OOL, we are concerned primarily with a large number of relatively small objects. As an example, we consider individual nodes of an abstract syntax tree to be objects, and the entire tree to be a collection of objects. In contrast, traditional database systems are concerned with a smaller number of relatively larger objects in the form of records, relations, or even files. Traditional databases perform best on records somewhat larger than objects in a typical OOL, and even better on "batch" processing of considerable numbers of records in similar ways. Attempts to use existing databases for object-sized data have proved less than satisfactory. A combined OOL/OODB must be able to fetch a large number of these smaller objects rapidly. Still, the system must cope with large objects, such as unstructured source code strings or image data, in a reasonable manner [Bernstein, 1987].

Pointer Chasing. In an OOL a great deal of emphasis is placed on the relationship between objects. As a result, the traversal of relationships between individual objects – *pointer chasing* – is a common operation in an OOL. When implemented in a relational database, one might call this "relationship traversal". The point is that one is frequently inspecting particular relationships starting from a particular object. Traditional databases are generally not very well tuned to this style of access, and do better on bulk operations, such as relational joins, rather than on selecting one record or field at a time. Not only is pointer chasing significant in OOLs, but it is important in design applications, which typically use structures designed for efficient use within main memory. The generality of the relational model, which is good for ad hoc queries not anticipated in advance, is traded for efficiency on a heavily used data structure. A challenge facing OODB system designers is to provide support for ad hoc queries in addition to efficient pointer chasing. Further, one generally wants to be able to trace relationships in both directions, so OOLs should support relationships in addition to simple pointers or names, as discussed in [Rumbaugh, 1987].

Query Optimization. In order to search a large database efficiently, traditional database systems rely on sophisticated query optimization. Query optimization depends on the fact that the operations defined over a database are simple, few in number, and well-defined. This assumption is not valid under the object-oriented model, since the operations available on a given object depend on the type of that object. In general, design applications depend more on the relationships between types and individual objects than on the relationships between large amounts of data. Queries in a design application are not needed as much as efficient object retrieval, as previously argued. This is good, since query optimization for OODBs is not very well understood yet. Still, the object-oriented model can take advantage of direct links between objects, and is well suited to performing type-specific optimizations, implemented within the code for the type [Bloom and Zdonik, 1987]. Refer to [Banerjee *et al.*, 1987b] for further discussion of queries in object-oriented databases. We believe that querying should be supported, but that it can be layered on top of the basic object-oriented storage manager.

Sharing, Reliability, Cooperation, and Distribution. Since typical database transactions offer concurrency control (concurrency must be considered in any shared system) and reliability features (of obvious value in a design environment), transactions are a good starting point for a framework that includes sharing and reliability. Design applications demand more than traditional databases, though: support for cooperation. It becomes an interesting challenge to provide appropriate primitives for programmers so that they can build a wide variety of design tools that simultaneously prevent unwanted interference, protect application data from the effects of failures, support cooperation, and perform acceptably.

In a design environment, transactions will be both short-term and long-term. Short-term transactions, which typically fetch and update only a few objects, are the most common. These transactions will usually involve a small amount of data. Long-term transactions, typically associated with file check-in/check-out, are also very important, and are related to cooperation in the sense that controlled sharing of intermediate results of long transactions is one of the fundamental means of cooperation.

We model long transactions as a series of short transactions. Short transactions are patterned after those of traditional database systems, and are not designed to support cooperation. For cooperation, the programmer should design a modest number of application-specific atomic data types. An atomic data type is one that provides transaction oriented concurrency control and recovery. The application-specific atomic data types would embody the modes of cooperation desired for that application. A facility for building application-specific atomic data types such that they are integrated with the short-term transaction mechanism will allow virtually any pattern of cooperation while providing concurrency control and recovery as desired. We describe this idea further in Section 3.7.

Since we are concerned with multiple cooperating users, collections of workstations and servers are one of the important hardware environments that must be supported. This is what makes distribution an issue.

Data Modularity. It is important in design systems to be able to isolate and manipulate significant subcollections of the data maintained by the system. This is important

for reasons of resiliency, performance (e.g., local copies in workstations), sharing, access control, archival reference, use with other systems and tools, etc. For example, one might wish to retrieve a copy of the source code for a single module and export it to another machine, perhaps by mailing a floppy disk to another user. Traditional databases are not especially good at this; in fact, file systems provided the best examples. We need to provide similar functionality, but at a level that includes object semantics as opposed to simple bags of bytes.

Transparency. The transparency of language integration is important because it simplifies programming. Thus, it is best that the extension of a language to support a reliable, shared collection of objects be as transparent as possible. It is this desire that guides us towards heap-based languages, such as Smalltalk [Goldberg and Robson, 1983] and Trellis/Owl [Schaffert *et al.*, 1986]. A persistent shared heap offers a very high level of transparency, allowing the user to ignore many details of access to persistent data. Some significant aspects of database functionality cannot be totally transparent to the user, however. For example, in order to access shared data or to recover from crashes, the user is required to understand transaction concepts to some degree.

Garbage Collection. A final issue that we must deal with is that of garbage collection. This issue arises primarily because of our preference for arbitrary pointer structure and automatic storage management. Garbage collection of the transient (non-persistent) heap is an issue that has been given a great deal of attention in the past. Garbage collecting a persistent heap is a difficult task, but solutions for large heaps have been proposed [Bishop, 1977]. We intend to adapt these approaches to the integrated persistent environment.

To summarize, in integrating an OOL and OODB in support of design applications, we believe it is important to address these issues: large numbers of small objects; pointer chasing; query processing; sharing, reliability, cooperation, and distribution; data modularity; transparency; and garbage collection. This list is not exhaustive; it is easy to think of many other issues, some of which are being explored in various other OODB projects, including exception handling, schema evolution, versioning mechanisms, design history, access control and protection, etc. The issues we discussed are simply the ones that we most desire to address in our research.

1.3 Specific Goals of the Mneme Project

Above we discussed issues of relevance to our overall research program. We turn now to our goals in the Mneme project. Mneme is a persistent store, rather than an integrated OOL/OODB. The idea is to use Mneme as a basis for exploring some (but not all) of the issues we are addressing in our longer term research program. We desired a tool that could be used with more than one language, that would take advantage of existing storage managers or network servers, that would enable us to wrestle with the critical low level performance problems related to accessing and manipulating small objects on demand, and that would provide maximum functionality consistent with the other goals. We now consider each of these goals in more detail.

Language Independence. Because of the effort involved in building a prototype, we wanted it to apply to more than one language. We also wanted to avoid undertaking very much language design, or compiler writing or modification at this stage in the research. We believe that we can learn many very useful things without getting involved in the linguistic aspects of OOL/OODB integration. As we will describe in more detail later, we decided that Mneme should provide untyped objects, but it does distinguish between object references and other sort of data within each object. That Mneme objects are untyped is a direct result of our desire for language independence. Since Mneme provides in essence a (persistent, distributed, reliable, shared) heap of objects, it meshes best with heap-based languages such as Smalltalk, Trellis/Owl, or even CLU [Liskov *et al.*, 1977; Liskov *et al.*, 1981]. Mneme can be used with more traditional languages such as C [Kernighan and Ritchie, 1978], C++ [Stroustrup, 1986], Pascal [Jensen and Wirth, 1974; ANSI, 1983], or Ada[1] [Ichbiah *et al.*, 1979], though, with the degree of transparency determined by the extent to which one is willing to modify the language (a little) and the compiler and run-time system (a lot).

Being language independent in the way we chose (a heap memory model) automatically ruled out query processing. Hence, Mneme does not deal with that issue at all. See our discussion of future plans at the end of the paper for more on this point.

Use Existing Software. We wanted to be able to use, evaluate, and compare a variety of existing and planned storage/object managers/servers, including the Exodus storage manager [Carey *et al.*, 1986], ObServer [Skarra *et al.*, 1987], Camelot [Spector *et al.*, 1986], and the Genesis tool kit [Batory *et al.*, 1986]. This would also save effort in building a fully functional prototype, and prevent "reinventing the wheel".

Explore Performance Problems. The primary performance problems we wished to explore are clustering of objects, caching, concurrency control and recovery algorithms, and object addressing and format conversion techniques. Our intuition was that per-object overheads would destroy performance and must be carefully controlled. The generic nature of Mneme (in that it should support multiple languages) somewhat interferes with achieving the best possible performance. This is because Mneme imposes a format on objects which may not directly correspond to that used by the programming language, thus causing extra copying or conversion of data. For the moment we have decided to live with this and study ways of eliminating all or most of the conversion effort for incorporation at a later stage of our work. This is described in more detail in the discussion of future plans.

Since Mneme is intended to be used with a variety of languages and in a variety of applications, we felt it very important that Mneme provide means to specify and change policy decisions and algorithms, in order to tune performance. This desire led us to determine a policy interface in the architecture, and to require that more than one policy be possible at the same time (but applying to different sets of objects). These ideas will be explained in more detail later.

For reference and comparison, we have used the following as our specific performance

[1] Ada is a registered trademark of the Department of Defense.

goals:

- Object fetching: Assuming average object size to be 30 to 40 bytes, as is our experience with CLU, Smalltalk, and Trellis/Owl heaps, we would like to be able to fetch about 10,000 objects per second. Clearly this will require good object clustering, good physical clustering on secondary storage media, and low system overhead in performing any necessary I/Os.

- Object access: Given that all the relevant objects are resident, we would like to be able to perform at least 100,000 object field/slot accesses per second. This is necessary to support activities such as dragging objects on a workstation screen.

- Transaction commit rate: We have not devised either a good measure or a performance target, but we mention this we feel it necessary to obtain a moderate to high rate of commits for short transactions between a single client and a single server.

Maximum Functionality. We wanted Mneme to provide the maximum OOL/OODB functionality consistent with its being a persistent object store, independent of any particular programming language. That is, we wanted to address as many of the OOL/OODB integration issues as possible. The requirements we settled on are:

- Sharing, Reliability, Distribution: Mneme would support short transactions within a distributed system.

- Cooperation: We would provide facilities for building application-specific concurrency control and recovery.

- Object Size: Mneme would support objects of virtually any size, transparently to the client.

- Object References: The object references would normally be the size of a pointer in the programming language, for ease of integration with languages, as well as to conserve storage. At the same time, the total number of objects that can be stored must be limited only by available resources, not by a lack of address bits.

- Garbage Collection: Automatic storage reclamation must be possible, but was not required. This demands the ability to locate and examine object references within objects.

- Data Modularity: A natural and easy to use data modularity facility was required.

Let us briefly mention some of the advantages that accrue to Mneme if these goals are met. Language independence is mainly a labor saving device: it will encourage Mneme to be used more broadly and give us more information with which to evaluate it. Using the same subsystem across languages will also allow more meaningful comparisons to be made, since fewer factors will be different in the comparisons. We will also see just how

important query processing really is by trying to get along without it. Getting a handle on the basic performance issues of dealing with large numbers of small objects is clearly important, no matter how seamless the integration of language and database, though some very sophisticated optimizations are ruled out since we have no linguistic notation to analyze and optimize. Exploring policies, and supporting multiple policies, are both likely to be quite important. The functionality required (sharing, reliability, distribution, cooperation, etc.) will make Mneme a very desirable vehicle for a number of applications, and Mneme's policy, language, and storage manager flexibility will serve to increase its range of applicability. In sum, the underlying advantage of our goals is that they are directed towards maximum flexibility and learning in research, while simultaneously providing a really useful tool to the widest variety of potential clients.

2 Overview of Mneme

We now consider the interfaces provided by Mneme. Figure 1 details the system architectural view of Mneme, which is useful in understanding the interfaces and the relationships Mneme has with other system components.

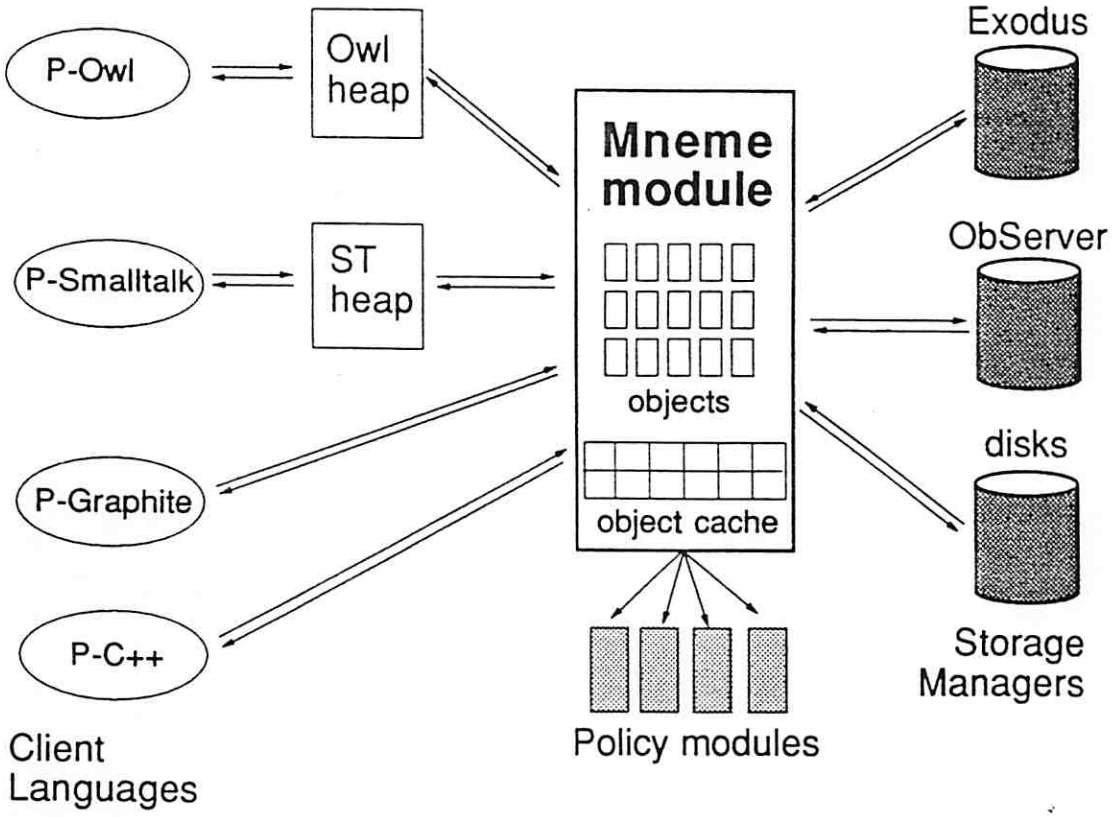

Figure 1: How Mneme fits into overall systems.

2.1 The Client Interface

The client interface consists of a number of routines that may be called to access Mneme facilities. Depending on the language being supported, calls might be avoided or reduced via inline expansion, a possibility with Ada, or via macros, as in C and systems built using C such as our Smalltalk system [Moss *et al.*, 1988] and Trellis/Owl.

Via the client interface, Mneme offers a simple and efficient abstraction of objects. These objects are similar to objects found in traditional OOLs, such as Smalltalk and Trellis/Owl. The objects of Mneme, however, are not biased towards any particular OOL. Mneme objects are intended to be used as the basis for an automatically managed persistent store interfacing to the heap of the OOL. The objects provided by Mneme also can be used from more traditional languages, such as C++ and Ada, as an additional pre-programmed abstract data type. Mneme will also interface with language based tools such as Graphite [Clarke *et al.*, 1986]. Any given copy of Mneme most likely interacts with a single program or tool; data sharing comes about through the use of shared storage managers.

Mneme is truly object-oriented, in the sense that it is based on objects allocated in heap storage. An object can have a set of references to other objects. These references are called *slots*. In addition, an object can have a *data* area consisting of a number of uninterpreted bytes. The objects provided by Mneme reside in a reliable, shared store of objects.

Mneme avoids pre-empting language decisions by defining a very general kind of object but not defining a type system or code execution model for Mneme objects. This implies that Mneme objects must be brought to the language's execution mechanism, thus inhibiting a large amount of back-end processing. This point is discussed further with our future plans.

The client interface actually deals with several other concepts in addition to objects. These will be introduced in later sections.

2.2 Storage Manager Interface

The storage manager interface is the means by which Mneme actually stores and shares data persistently. Mneme supports multiple storage managers, possibly simultaneously. The basic features required of a storage manager are that it store and retrieve sequences of bytes (we call such sequences *segments*), and that it provide some concurrency control and recovery features.

2.3 Policy Interface

Mneme also offers flexibility in the storage management policy and strategy used by a client language. Sets of related objects can be managed by the strategy that is best suited to those objects. Mneme provides a collection of built-in strategies, as well as a facility for users to define and exploit additional strategies. Objects are grouped into *pools*, where each pool has an associated *policy module*. This is explained in more detail later.

2.4 Buffer Manager

Although the buffer manager is actually part of Mneme, it is built as a separate module and is conceptually distinct from the rest of the system. As of this writing we are still working on the details of its relationship to the other system components.

3 Client Concepts

Mneme presents several abstractions to its clients: objects, object identifiers, files, roots, handles, transactions, and pools.

3.1 Objects

A Mneme object has two primary components: an array of slots and a array of bytes. A byte is simply an uninterpreted 8 bit quantity. The bytes part of an object may not contain references to other objects. A slot is the size of a pointer (32 bits on typical machines), and may hold a reference to an object. A slot may also be empty (represented by 0) or hold an *immediate value*. Immediate values are a concession to heap based languages that allow integers and other non-pointer data within slots; CLU, Smalltalk, and Trellis/Owl operate in that way. The sign bit is used to distinguish references (positive) from immediate values (negative).

In addition to slots and bytes, objects have a few attribute bits, which we use to mark them as read-only or as having other special properties. Mneme provides routines to determine the number of slots and bytes an object has, and to fetch and store slots, bytes, and attribute bits. Objects can grow and shrink in either the number of slots or bytes. This is done by truncating or appending empty slots or zero bytes as necessary.

3.2 Object Identifiers

Every object currently accessible to the client has a short name, the size of a pointer, called a *client identifier*, or CID for short. An application can thus name and manipulate up to about 2^{30} objects at a time. The exact value of a CID is not the same as what is stored within a slot of an object inside Mneme. This is because the overall space of objects is not bounded, hence objects cannot be assigned a short unique identifier. What we actually do is best explained step by step, as we go. We call the reference values stored in slots inside Mneme *persistent identifiers*, or PIDs.

3.3 Files

A *file* contains a collection of objects. Files are the unit of data modularity in Mneme, and correspond roughly with the usual operating system notion of file, even to the extent that operating system files might be a reasonable way of implementing Mneme files. Within

a given file, each object has a unique PID value, and objects in that same file can refer to the object simply by using the PID. Thus, individual files can contain up to about 2^{30} objects. Since there is essentially no limit on the number of files, there is no limit on the total number of objects that Mneme can store. Mneme files may have new objects added and removed over time.

For those cases in which an object needs to refer to an object in a different file, we support cross-file references. Cross-file references are implemented by having a reference to a specially marked object in the current file, appropriately called a *forwarder*. The forwarder then contains information detailing the file and object desired. It is possible to have a variety of kinds of forwarders, depending on the nature of the names and whether their interpretation or meaning is static or dynamic. We have not yet completed this aspect of the design, but expect to rely on some kind of external name service mechanism so as to avoid developing our own.

The modularization of the Mneme object base into a collection of files has other advantages beyond avoiding a limit on the total number of objects. Having separate files allows parts of the object base to be separated for processing by particular tools, and files can form a natural unit of transfer between systems and organizations. That is, files provide a natural way of avoiding a monolithic database. Files also allow convenient distribution of data; they map nicely onto existing operating systems and servers; they are a familiar concept; and they enhance reliability by providing storage fault isolation and containment. The partitioning of the object world into files is also important for storage reclamation and garbage collection.

Files can have string named, string valued attributes associated with them. These attributes can be used for correctness checks (e.g., "I am a file made by a Smalltalk system", "I belong to Jane", etc.), for policy parameter storage, and so on.

3.4 Roots

Mneme provides a vast sea of objects, and is designed to allow one to trace paths of object references through this sea. One must, however, have some means for getting started and oriented within the sea of objects. Every Mneme file has a *root object*. To start using Mneme one opens an appropriate file (determined by the application and/or the user) and begins at that file's root. This is similar to the notion of the root directory in a file system. In addition to being able to obtain the CID of the root object of a file, Mneme clients can change which object is the root of the file. Files start without roots, so if the client fails to designate a root, then the root fetching operation will fail.

Note that in the present scheme there is no way to identify or name a Mneme object uniquely from *outside of* Mneme. We are considering adding features for such external persistent identifiers.

3.5 Identifier Conversion

Though it is not a concept visible to the client, there is obviously some conversion going on between CIDs and PIDs. This is handled as follows. First, assume that each file predeclares the maximum PID value it will contain (we will see how to relax this assumption in a moment). A particular incarnation of Mneme within a given client maintains an abstract address of CIDs by assigning each file a contiguous non-overlapping chunk of CID values. Such a chunk can be described by its first CID, which we call the *base*, and the number of CIDs in the chunk, which can be any number greater than the maximum PID value for the file. To convert PID to CID we simply add the base for the file; to convert CID to PID, we determine the file (conceptually by searching a table describing the CID range for each file, but actually implemented more like a virtual memory page table mechanism, as discussed later) and subtract the file's base from the CID to get the PID. Given the file and the PID, we can then locate the object as necessary.

There are several ways to deal with the maximum PID value assumption. One is to stick with a hard limit on the file. Note, though, that we just need a limit assigned as we *open* the file, so we could allow a file to grow without bound, but just limit its growth within one session of use. Finally, if we do not care for a limit even during a session of use, we can assign the file another, larger, chunk of CIDs when it grows too big. Old CIDs that have been handed will still translate to the correct objects, but we now use the new base for all CIDs handed out in the future. In any case, the PID to CID conversion is always a simple addition, and the CID to PID conversion involves relatively inexpensive table indexing (to be described later).

3.6 Handles

A Mneme object is manipulated by obtaining a *handle* on the object. A handle is requested by supplying Mneme with the CID of the desired object. While a handle is held on an object, the holder is guaranteed logically exclusive access to the object. Obtaining a handle generally forces the corresponding object to become resident. Handles also allow us to use a number of internal object formats (e.g., small objects with a contiguous representation, and large objects with a tree representation) without sacrificing efficiency. A handle stores a more time-efficient run-time representation of an object than a CID, though it consumes more space. Basically, handles prevent doing the CID to PID conversion, object presence check, and object location determination more than once for a whole series of operations on the same object.

3.7 Transactions

Mneme provides a basic transaction facility as well as features for supporting more sophisticated transaction management. We first present our transaction *semantics*, followed by our implementation strategies. Within a transaction, concurrency control is on a per-object basis, allowing shared read and exclusive update. In addition to concur-

rency, transactions are units of recovery, with Mneme supporting atomic, all-or-nothing, transaction commitment.

Since locking individual objects is likely to be prohibitively expensive, in the implementation we can lock objects in groups – probably in terms of segments, the units of transfer with storage managers. This is not visible to the client, since segments are not visible through the client interface. The only possible problem is that coarse granularity locking might lead to unexpected deadlock. There are a number of ways to avoid this problem; one is to allow the client to designate objects that must be independently lockable, and guarantee that the objects are in different segments. Whether locking is actually done on segments or objects is a policy decision, determined by the actions of the relevant policy module. This will be discussed more later.

The implementation has considerable latitude in concurrency control strategies. As mentioned, the granularity can be made more coarse without violating the basic semantic guarantees. A variety of algorithms may also be used, with read/write locking and optimistic concurrency control being two prominent alternatives. Further, since policies may be mixed, any set of mutually consistent concurrency control policies can be used at one time.

To encourage transparency, clients do not lock objects explicitly. Rather, observing properties of or reading slots or bytes from an object are considered reads, and changing the object in any way is considered a write. Whether and when a lock is actually acquired is up to the policy module, except that the basic semantics must be preserved. One policy module might lock objects exclusively as soon as they are referenced (very pessimistic about other clients modifying the object and interfering), while another policy module might use optimistic concurrency control and check consistency only at transaction commit time.

With regard to recovery management, logs, shadows, or combinations may be used, and we will address group commit and related optimizations as well. Mneme's client and server interfaces make no assumptions about distribution or replication of files or objects, so additional variation is permitted in those areas.

Since serializability at the object level is sometimes overly restrictive, especially when implementing protocols for cooperation, Mneme provides some support for more sophisticated serializable, or even non-serializable, interaction. The basic concepts include ***volatile objects***, ***object logs***, and ***event notifications***. A volatile object is accessible to other clients and may be changed whenever the client does not have a handle on it, regardless of transactions. This provides the basic mutual exclusion mechanism necessary for building arbitrary concurrency control. Further, a client's uncommitted changes will be visible to other users, and vice versa. Thus, handles provide mutual exclusion on volatile objects, which is the only concurrency control and recovery that volatile objects naturally enjoy.

To support reasonable use of volatile objects, Mneme also provides object logs, wherein past or intended changes can be recorded, so that when a transaction commits or aborts, it can complete or clean up its manipulations of volatile objects. As an example, consider a directory implemented as a volatile object. It is desirable to increase concurrency by allowing non-conflicting access to the directory rather than locking the whole directory for

the duration of entire transactions. When making a change, such as adding a new entry, the directory code acquires a handle on the directory. It then performs the update, marking the change as tentative, and releases the handle. The tentative change is then visible to other clients, though the directory code should hide it, probably by forcing clients to wait until the entry is not tentative. This would provide serializable directories, without holding locks on the whole directory for the duration of entire transactions. As an alternative, the directory code could let the change show through, providing non-serializable behavior similar to a check-in/check-out recording data structure, which, rather than blocking, might reply, "Joe already has that checked out; do you really want to go ahead?" The object log is used to record the tentative change, so that is the transaction fails, the change can be removed, and if the transaction commits, the change can be made permanent rather than tentative.

An event notification mechanism is desirable for efficiency, so that clients waiting for resources can be notified of their availability rather than busy waiting or polling.

In sum, volatile objects, object logs, and an event notification system allow one to build application-specific atomic objects, along the lines of [Weihl and Liskov, 1985] and [Spector and Schwarz, 1984], or even non-atomic or non-serializable objects to support various forms of cooperation.

3.8 Pools

A *pool* is a collection of objects managed according to the same storage allocation and object management strategies. A Mneme file consists of one or more pools, with pools partitioning the objects of the file. The pools of a file may have different strategies, as may the pools of different files. We will provide some built-in strategies, but the sophisticated Mneme user will be able to devise new pool types and add them to the system. Pools avoid imposing any single object management strategy, which is bound to fail for some applications. Another advantage of pools is that different subcollections of objects can be managed in different ways, rather than all objects being managed the same way.

Here is an example showing the utility of multiple strategies. Suppose that an application has a small number of high traffic, frequently updated objects, and a large number of objects that are not changed frequently and are rarely touched by multiple users during the same period of time. Transaction management efficiency might well be maximized in this situation by treating the two sets of objects quite differently. The high traffic objects might be placed in a pool that locks objects exclusively when they are first referenced. The low traffic objects could be in a pool managed using optimistic concurrency control. We need only insure that we get a consistent serialization order across all strategies.

The discussion of possible transaction implementations indicates some of the factors that pool strategies can control: the concurrency policy and granularity, and the details of recovery. Other important policies that pools define include: object clustering, storage allocation, object or segment cache loading and cache replacement strategies, and prefetch techniques. Volatile objects belong to volatile pools. Even within volatile pools, there is considerable latitude in policy: when a handle on a volatile object is released, the object

or segment might be returned to the storage manager immediately, or it might be retained until the storage manager requests its return.

The pool concept is crucial to making Mneme useful across a range of design applications, because the ability to extend policies and to use multiple policies simultaneously will be necessary in achieving adequate performance.

4 The Initial Design of Mneme

The main parts of the design of Mneme that are not obvious are the location and faulting in of objects, the storage manager interface, and the policy interface. It is easy to dispense with the storage manager and policy interfaces. The storage manager interface must deal with files, provide segments of reasonable sizes (though their size need not be modifiable), support concurrency control on segments (or ranges of bytes within segments), and offer simple (traditional) transactions. The storage manager interface also requires event notification support and some operations to assist in effective uses of large buffer caches (cache coherence verification). As of this writing the details are not available.

The policy interface is also not yet available in detail, but we have determined its basic style. A policy module will consist of a number of routines, one routine for each of a number of specified events, such as "object to be created", "handle requested", and so forth. A policy routine must carry out the requested action using parameters provided in the call, any data in the buffer cache, storage manager routines, etc. We may also arrange things so that handles point to a vector of routines (determined from the pool for the object corresponding to the handle), and each call on a handle indexes into this vector to find the actual routine to call.

An initial object location and faulting mechanism has been designed, and it works as follows. The space of PIDs for each file is broken into "chunks" of 1024 PID values. Such a "chunk" is called a *logical segment*. Note that a logical segment does not refer to any specific number of *bytes*, only to a particular set of objects that all have PIDs with the same high order bits. The segments transferred between the buffer cache and the storage managers are called *physical segments* to distinguish them from the logical segments. Every physical segment has some set of logical segments assigned to it, and a logical segment is associated with exactly one physical segment.

Objects are located by splitting the PID into two pieces; the high order bits indicate the logical segment and the low order bits the specific object within the logical segment. Information about the logical segment is found in a *logical segment table*, which is indexed directly by logical segment number. The logical segment table indicates which *physical segment* contains the logical segment, and, if the physical segment is resident, where the logical segment's information actually resides within the physical segment (probably via a direct pointer to the information, in the middle of the physical segment data in the buffer cache). The logical segment information includes a table of self-relative pointers to the 1024 objects of the logical segment; the low bits of the PID are used to index this object table and find the object data.

In addition to the logical segment table, each file has a physical segment table indicating the storage manager's name for the physical segment, the segment's size, where (if anywhere) the segment is resident, and so forth. A non-resident segment is detected by a null pointer in the logical segment table when trying to find an object from its PID. At that time the physical segment is fetched, the physical and logical segment tables updated, and the reference processing continued.

The actual CID to object conversion does not proceed by building a PID and then looking up the object. In one design we concatenate the logical segment tables of all the open files, and index directly with the high order CID bits. This has the same effect as converting to a PID and then indexing, but is faster and avoids a table lookup to determine the file.

The design we are incorporating first is a bit simpler, but more restrictive: no file is allowed to have more than 2^{20} objects, and we use the bits beyond the 20th bit to mean a file number within the current client session. The lookup algorithm extracts the highest order bits to index a table of file information and locate the logical segment table for that file. The middle bits are then used to locate the specific logical segment, and the low bits determine the object within the logical segment. In sum, one indexes the file table, then the file's logical segment table, then the logical segment's object table. The CID format is shown in Figure 2 and the indexing scheme is illustrated by Figure 3.

F = File #	L = Logical Segment #	I = Object #
10–12 bits	10 bits	10 bits

Figure 2: Format of a Client Identifier

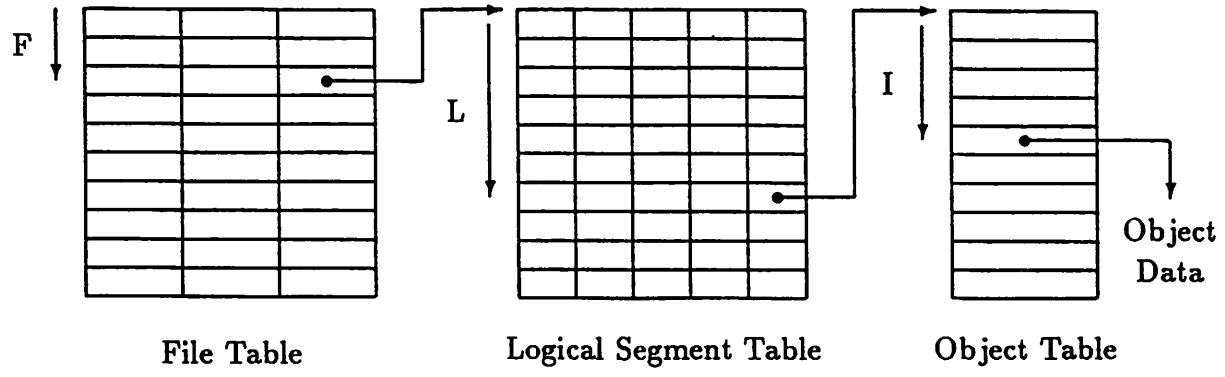

Figure 3: Three Index Object Lookup Scheme

While this three index scheme is a bit slower than the two indexes used in the concatenated logical segment table design, the three index design is much simpler to implement. It may also turn out to be fast enough: initial estimates show that an object can be located

from its CID in about 10 instructions, which corresponds well to our 100,000 references per second goal on current hardware.

The resulting object location mechanism is very similar to the virtual address translation mechanism used by many paged virtual memory systems, and is amenable to hardware speedup perhaps, though a translation lookaside buffer is not likely to be as effective.

PIDs are seen to be somewhat location dependent (they always specify a particular logical segment), but the logical segment can be moved anywhere in a file without affecting PID values, and the objects can be moved anywhere within the physical segment containing the logical segment. In particular, the objects in a physical segment can be reorganized and compacted without affecting PIDs. Thus, PIDs have a considerable amount of location independence as well. We believe the scheme represents a good tradeoff between efficiency of locating objects and maintaining location independence.

Object themselves are represented using a header, which contains a format code, size information, and the attribute bits. The header is followed by the slots and then the bytes, padded out to a complete word, as showin in Figure 4. The header is generally one word long; the format code may indicate an object is longer than the limited size field can represent, in which case one or two additional words are used to hold these large sizes.

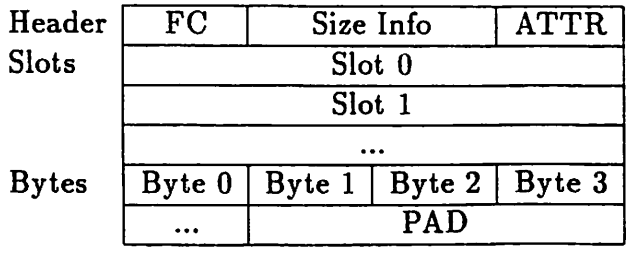

Figure 4: Object Format

A physical segment contains the information pertaining to its logical segments, followed by the object data area. In processing a physical segment upon fetching or storing, a small amount of work must be done for each logical segment, to update the tables, but no per-object work is needed.

5 Comparison With Other Work

We compare Mneme with some representative language and database alternatives, in terms of how they offer support for design tools and environments. Our criteria in evaluating the various approaches are the issues discussed in Section 1.2.

5.1 Integrated Database Languages

An early attempt at integrating programming language and database functionality was the programming language Pascal/R [Schmidt, 1977]. Pascal/R was a major step in the advancement of integrated database languages. The language made use of a consistent type notation, and the program and database had an obvious relationship. Rules of type-checking could be applied to the database, as well as to standard Pascal types. Unfortunately, the database type introduced was not type complete. In fact, a Pascal/R database was permitted to have only the relation type (another type introduced by Pascal/R) as fields, and none of the other standard Pascal types. In addition, the programmer was only allowed to have a single variable of type database. Pascal/R included facilities for taking full advantage of the algebra of relational databases. In general, however, the user of Pascal/R still had to reason with the relationship between two representations of the data: persistence in Pascal/R is neither orthogonal nor type complete. While Pascal/R incorporates some novel concepts in interfacing languages and databases, it does not represent a seamless integration, and its foundation is the traditional relational database. For these reasons it is not a very suitable vehicle for design applications or environments.

A conceptual successor to Pascal/R was PS-Algol [Atkinson *et al.*, 1981; Atkinson *et al.*, 1984]. PS-Algol is in many ways like Pascal/R. A traditional language was extended to include persistence. In PS-Algol, however, the notion of persistence was, for the first time, extended to include type completeness and orthogonality. PS-Algol proved that this view of persistence is feasible. The language, however, did not provide any mechanisms for concurrency or reliability of data, and was therefore ill-suited for cooperative work or design applications.

On the surface LOOM [Kaehler and Krasner, 1983], a large object-oriented virtual memory system, appears to be similar to Mneme in that it extends the Smalltalk heap to include disk storage and does transparent object faulting. In fact, LOOM's goals are substantially different from those of Mneme. In particular LOOM provides a single-user virtual memory or workspace model, and provides no concurrency, reliability, or distribution features. LOOM resembles single user programming environments such as Interlisp [Teitelman and Masinter, 1981] more than it resembles Mneme.

GemStone [Purdy *et al.*, 1987], similar to LOOM, expands the Smalltalk heap to include objects on disk. Unlike LOOM, GemStone does provide considerable database functionality, including queries and an execution model. The GemStone system is distributed, but the database is stored on a single server. In contrast, Mneme will support a multiple server environment, which is desirable for the moderate to large projects typical in design work. GemStone is somewhat specialized to Smalltalk, whereas Mneme is designed to support a variety of languages. In addition, Mneme provides the user the ability to modify and extend object management policies.

5.2 Modern Database Systems

Several modern database systems provide good examples against which we compare Mneme. One is Postgres [Stonebraker and Rowe, 1986], a continuation of the Ingres [Stonebraker *et al.*, 1976] project. Postgres represents an attempt to extend the relational database approach to deal with more kinds of data and to suit a wider range of applications. It is still firmly rooted in relational soil, however. Because Postgres does not provide special support for pointer chasing, and because it is not attempting integration with a programming language, it does not meet the criteria outlined in Section 1.2.

Exodus [Carey *et al.*, 1986] takes an approach different from that of Postgres, providing a core of database functionality and a language, E [Richardson and Carey, 1987], in which to implement revisions or extensions to the system. E does not provide orthogonal or type complete persistence, as is the intent of Mneme. E also does not provide particular support for objects or for control over the storage manager's policies. Further, E is not necessarily intended to be the application programming language, but rather the database implementor's language for databases built using Exodus. While the Exodus storage manager is not suited for use in a distributed system, it does appear useful enough for design applications that we will experiment with it as an underlying storage manager for Mneme.

Genesis [Batory *et al.*, 1986] is a database toolkit, offering a variety of pre-programmed components which may be assembled to devise one's own database system. Again, we will experiment with Genesis components in building disk storage managers for Mneme. ObServer [Skarra *et al.*, 1987] provides objects that are blocks of bytes, with some per-object locking and novel concurrency control, and is intended for use as a server. Neither ObServer nor Genesis provide facilities for language integration.

Finally, Orion [Banerjee *et al.*, 1987a], an object oriented database system, has more ambitions towards integration with a programming language, in this case Common Lisp [Bobrow *et al.*, 1985]. Similar to Pascal/R, Orion's persistence is not orthogonal, and the database type system is rather different from the host language type system.

6 Conclusion

We review Mneme's current status, indicate some questions for further research, and indicate our current thoughts on the next system we would like to build.

6.1 Current Status

We are in the process of implementing the first phase of Mneme. Initially, we will provide for only a single-user, without reliability. This will permit us to experiment with the initial design of most of the client, storage manager, and policy interfaces. We are simultaneously designing and implementing a heap interface for our VAX[2]Smalltalk. Im-

[2]VAX is a trademark of Digital Equipment Corporation.

plementation will then proceed with the second and third phases, in which we provide for reliability and then sharing.

6.2 Some Questions for Future Research

The flexibility in the design of Mneme will afford us the opportunity to use Mneme as a testbed for many different solutions to the issues raised by integrating database and programming language functionality. These research problems include both database and language issues:

- In an environment such as Mneme, where persistent objects are both shared and reliable, how can one relocate objects, copy objects, and replace objects, all of which are crucial to design applications?
- How can we expand Mneme's notion of pools to include even more sophisticated object clustering facilities, such as those described in [Hudson and King, 1986]?
- Can persistent semantics be added to a language, while simultaneously providing the ability to modify and expand object management policies?
- What is an appropriate set of primitives for supporting cooperative manipulation of objects in distributed systems?

6.3 Future Plans

While the above are interesting research questions, some of which we may pursue, we have more specific plans for the future. It is clear that Mneme provides little more than an abstraction of storage: it has little notion of action. This prevents any query processing or optimization, the distribution of execution across system components (e.g., between client and server, or multiple clients), and so forth. The next project in our research will investigate the design of a *virtual machine*, rather than a store. We wish the virtual machine to be somewhat language independent (i.e., be a target for at least several languages) and to support object oriented languages. It should add execution facilities to Mneme, and possibly replace object access (read, write) with operation invocation. This new system would also allow us to address the issue of format conversion, since we would be designing an interpretation/execution engine rather than just a store.

A persistent, distributed, shared, etc., virtual machine will allow the exploration of many more policies alternatives, such as where code is executed, the extent to which and at what time it is compiled, etc. The system should not only be more interesting, in that it offers more functionality, but also present more challenging research problems. It will also move us further along the spectrum towards the consideration of linguistic issues, and certainly involve us in compilation and optimization problems.

References

[ANSI, 1983] ANSI. *IEEE Standard Pascal Computer Programming Language.* IEEE, New York, 1983. Standard ANSI/IEEE770X3.97–1983.

[Atkinson *et al.*, 1981] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a peristent heap. *ACM SIGPLAN Notices 17*, 7 (July 1981).

[Atkinson *et al.*, 1984] M. P. Atkinson, P. Bailey, W. P. Cockshott, K. J. Chisolm, and R. Morrison. Progress with persistent programming. In *Databases—Role and Structure: An Advanced Course.* Cambridge University Press, Cambridge, England, 1984, pp. 245–310.

[Banerjee *et al.*, 1987a] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Houng-Joo Kim. Data model issues for object-oriented applications. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 3–26.

[Banerjee *et al.*, 1987b] Jay Banerjee, Won Kim, and Kim Kyng-Chang. Queries in object-oriented databases. MCC Technical Report DB-188-87, Microelectronics and Computer Technology Corporation, Austin, TX, June 1987.

[Batory *et al.*, 1986] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. Genesis: A reconfigurable database management system. Tech. Rep. TR-86-07, Department of Computer Sciences, University of Texas at Austin, Austin, TX, Mar. 1986.

[Bernstein, 1987] Philip Bernstein. Database system support for software engineering. In *Proceedings of the Ninth International Conference on Software Engineering* (Monterey, CA, Apr. 1987), IEEE.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Bloom and Zdonik, 1987] Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 1987), ACM, pp. 441–451.

[Bobrow *et al.*, 1985] D. G. Bobrow, 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 Palo Alto Research Center, Palo Alto, CA, 1985.

[Buneman, 1984] Peter Buneman. Can we reconcile programming languages and databases? In *Databases—Role and Structure: An Advanced Course.* Cambridge University Press, Cambridge, England, 1984, pp. 225–243.

[Carey *et al.*, 1986] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th International Conference on Very Large Databases* (Kyoto, Japan, Sept. 1986), ACM, pp. 91–100.

[Clarke *et al.*, 1986] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Graphite: a meta-tool for Ada environment development. In *Proceedings of IEEE Society Second International Conference on Ada Applications and Environments* (Miami Beach, FL, Aug. 1986), IEEE.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hudson and King, 1986] S. Hudson and R. King. CACTIS: A database system for specifying functionally-defined data. In *Proceedings of the Workshop on Object-Oriented Databases* (Pacific Grove, CA, Sept. 1986), ACM, pp. 26–37.

[Ichbiah *et al.*, 1979] J. D. Ichbiah *et al.*. Rationale for the design of the ADA programming language. *ACM SIGPLAN Notices 14*, 6 (June 1979).

[Jensen and Wirth, 1974] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*, second ed. Springer-Verlag, 1974.

[Kaehler and Krasner, 1983] Ted Kaehler and Glenn Krasner. LOOM–large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner, Ed. Addison-Wesley, 1983, ch. 14, pp. 251–270.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Liskov *et al.*, 1977] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977).

[Liskov *et al.*, 1981] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[Moss *et al.*, 1988] J. Eliot B. Moss, Antony L. Hosking, Rajesh Nakhwa, and Steven Sinofsky. Implementing Smalltalk-80 on the VAX. Tech. rep., Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1988. Work in progress.

[Purdy *et al.*, 1987] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 27–47.

[Richardson and Carey, 1987] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementations in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (San Francisco, CA, May 1987), ACM, pp. 208–219.

[Rumbaugh, 1987] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 1987), ACM, pp. 466–481.

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 9–16.

[Schmidt, 1977] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247–281.

[Skarra *et al.*, 1987] Andrea Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object oriented database system. In *Proceedings of International Workshop on Object-Oriented Database Systems* (Pacific Grove, CA, Sept. 1987), ACM, pp. 196–204.

[Spector and Schwarz, 1984] Alfred Z. Spector and Peter M. Schwarz. Synchronizing shared abstract data types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.

[Spector *et al.*, 1986] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The camelot project. Tech. Rep. CMU-CS-86-166, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[Stonebraker and Rowe, 1986] M. Stonebraker and L. A. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Washington, D.C., May 1986), ACM, pp. 340–355.

[Stonebraker *et al.*, 1976] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of Ingres. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.

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

[Teitelman and Masinter, 1981] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer 14*, 4 (Apr. 1981), 25–33.

[Weihl and Liskov, 1985] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Trans. Program. Lang. Syst. 7*, 2 (Apr. 1985), 244–269.