Managing Persistent Data with Mneme: Issues and Application of a Reliable, Shared Object Interface*

J. Eliot B. Moss *Steven Sinofsky*†

COINS Technical Report 88-30
April 1988

Object Oriented Systems Laboratory
Department of Computer and Information Science
Univerity of Massachusetts
Amherst, MA 01003

Abstract

We investigate issues that arise when attempting to integrate 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 over more traditional database systems. We then 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 then show how Mneme can be used to support software engineering. Finally, we compare Mneme with other current and recent research efforts.

---

*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' present address: Department of Computer and Information Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, 01003; telephone (413) 545-4206; Internet addresses 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.

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).

Our primary motivation for this project is to further the integration of programming language and database functionality. There has been a great deal of work describing and addressing the problems of integrating these two different technologies [Buneman, 1984, Atkinson et al., 1984]. Primarily we are concerned with providing truly seamless integration, with the 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. We intend to further these ideas of persistence to include sharing and reliability.

The object-oriented language framework provides a natural starting point for adding database features to programming languages [Bloom and Zdonik, 1987]. Object-oriented languages (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. Even though OOLs and OODBs complement each other well, there are still many issues that need to be addressed. We seek a tighter integration of the two. If such integration can be achieved, then many programming problems will be greatly simplified.

1.2 Issues

When attempting to integrate traditional database functionality with programming languages, one is presented with several issues that must be addressed. As we address

these issues, we shall see that many problems arise due to the assumptions that have been made. Since we are concentrating on the use of an integrated OOL/OODB in design applications, many of these assumptions can, and should, be modified.

In an OODB, 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 large objects in the form of records, relations, or even files. Traditional databases perform best on records two to ten times larger than objects in a typical OODB. Attempts to use existing databases for object-sized data have proved less than satisfactory. In sum, an OODB must be able to rapidly a fetch large number of these smaller objects. Still, OODBs must cope with large objects, such as unstructured source code strings or image data, in a reasonable manner [Bernstein, 1987].

In an OOL a great deal of emphasis is placed on the relationship between objects. As a result, *pointer chasing* is a common operation in an OOL. For similar reasons, pointer chasing is an important characteristic of design applications, where the relationship between objects representing some design is emphasized. In contrast, traditional databases rely heavily on keys or indices, which are useful for unpredictable queries and set-at-a-time operations, but inefficient for traversing individual relationships. While OODBs should support ad hoc queries, they require additional techniques to provide efficient pointer chasing.

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 than on the relationships between large amounts of data. Queries in a design application are not needed as much as direct object retrieval. The object-oriented model can take advantage of these two observations. Much of optimization previously necessary would be present in direct links between objects, and thus a query would be greatly simplified. In addition, it will be possible to design type-specific optimizations, which can be implemented within the operation code [Bloom and Zdonik, 1987]. The object-oriented paradigm is well suited to such type-specific optimizations. The reader is referred to [Banerjee et al., 1987a] for further discussion of queries in object-oriented databases.

In a design environment, transactions will be both short-term and long-term. Short-term transactions, which include object fetching and updating for example, are the most common. These transactions will usually involve only a very small amount of data. Long-term transactions, typically associated with file check-in/check-out, are also present in a design environment. In an object-oriented setting these long-term transactions can be modeled as a series of short-term transactions coupled with the use of a user-defined atomic data type. We describe this idea further in Section 3.4.

Traditionally, database systems have been able to provide useful subsets of data in the form of files or relations. A user could isolate important records and copy them to a

separate file for later use. This functionality is not currently available in object-oriented systems, but is one that is necessary in a design environment. 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. Mneme will provide such a capability.

The issue of language integration is of primary concern. If an OOL/OODB is to be successful in a design environment, then the integration must be very tight. 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 will permit us to provide a very high level of transparency. The user of Mneme will not have to worry about a great many details in order to use the persistent store. Many aspects of database functionality, however, cannot be totally transparent to the user. For example, in order to access shared data or to recover from crashes, the user is required to understand transaction concepts to some degree.

A final issue that we must deal with is that of garbage collection. This issue arises primarily because of our choice of heap-based languages. 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 Mneme environment.

2 Overview of Mneme

We begin by presenting, in order of importance, our primary goals of Mneme:

- Support for objects, but without pre-empting language decisions such as type systems.
- Flexibility, in at least three ways: the programming languages supported, the back-end storage managers or servers that may be used, and the object management policies available to the Mneme user.
- Good performance, within the constraints imposed by the other goals.
- Access from traditional languages as well as OOLs.

In the remainder of this section, we outline how Mneme will address these goals. In Section 6, we discuss future additions to Mneme, as well as research issues that arise.

2.1 Object-Oriented Storage Manager

Mneme is a tool offering 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++ [Stroustrup, 1986] and Ada[1] [Ichbiah, 1979], as an additional pre-programmed abstract data type. Mneme will also interface with language based tools such as Graphite [Clarke et al., 1986].

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, objects 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 languages' execution mechanisms, thus inhibiting a large amount of back-end processing. We discuss this issue further in this section.

2.2 Flexible Subsystem

Mneme is a flexible subsystem, residing between the application language and the storage manager. Mneme is designed so that both the languages that use Mneme and the storage systems that Mneme uses may be changed easily, in order to meet the needs of the design project. The programming language, or *front-end*, 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, available from the C language [Kernighan and Ritchie, 1978] and systems built using C such as our Smalltalk system [Moss et al., 1988] and Trellis/Owl.

Mneme also has a simple *back-end* interface, through which it communicates with the storage manager being used. Mneme will interface with any storage manager that can provide fixed size storage blocks and simple concurrency control. Storage managers, such as ObServer [Skarra et al., 1987], Exodus [Carey et al., 1986], Camelot [Spector et al., 1986], or bare disks, will be used at the back-end of Mneme. Figure 1 details the overall topology of Mneme.

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.

2.3 Performance

We wish to attain good performance according to two measures: the rate at which useful objects can be fetched, and the rate at which the objects may be accessed once

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

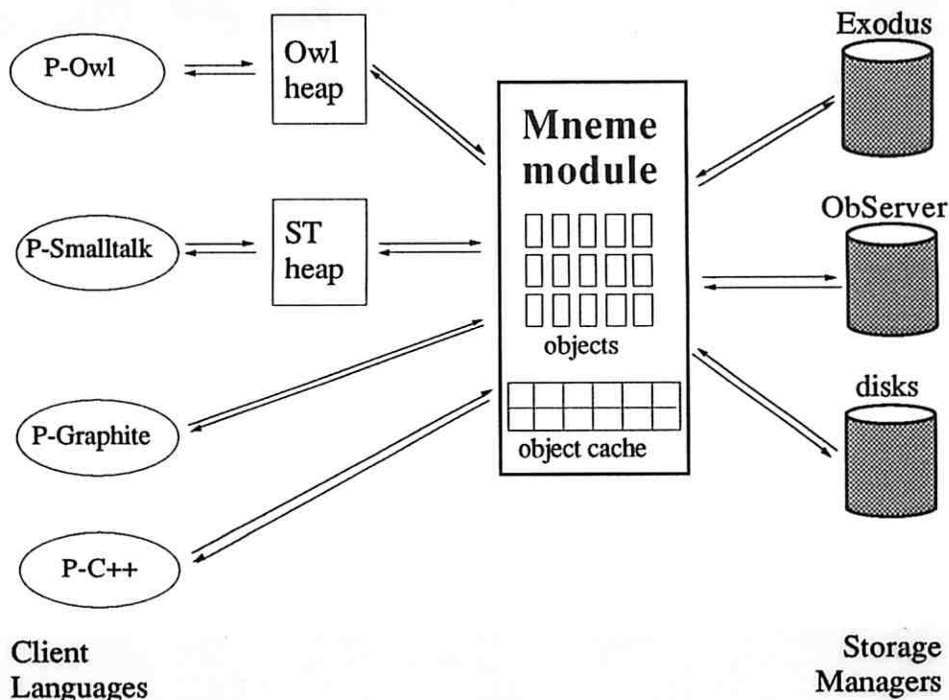

Figure 1: The topology of Mneme.

fetched, that is, once they are in Mneme's buffers. In addition, we are interested in the rate at which transactions against the object store can be committed, which will be the subject of other writings.

Our specific goal for the rate of fetching useful objects is 10,000 objects per second, assuming an average object size of 30 to 40 bytes. This goal depends on having an efficient back-end storage manager, on good clustering and caching of objects for the application, and on low per-object overhead in processing fetched objects. Clustering and caching are being addressed by permitting a variety of strategies and heuristics to be used in the same system. At present we depend on the application or client language to find or define the appropriate strategy or strategies.

Because Mneme provides a simple and general format for objects, which likely does not exactly match the format used by any particular client language, in some cases Mneme objects must be converted to the client language object format. This is expected to incur some amount of overhead, but appears to be avoidable only by specializing to the client language and integrating more closely with it, sacrificing Mneme's generality and flexibility.

Our target object access rate is 100,000 references to slots of objects per second, which should be adequate for interactive graphics displays. Once fetched, objects will be converted into heap based objects of the client OOL, resulting in only slight performance degradation over an OOL without persistent objects. One function that may slightly degrade performance is the necessity to check for object residency when invoking operations on objects, which is unavoidable in a persistent object system. A residency check is the

client language equivalent of a page fault, and we call it an *object fault*. Handling object faults will not slow most OOLs down very much, possibly not at all, if the language implementation uses an object table. Elaborating this point, however, is beyond the scope of this paper.

2.4 Support for Traditional Languages

The front-end call interface that Mneme provides addresses the issue of supporting traditional languages reasonably well. The result can be thought of as providing an additional data type, `PersistentObject`, to the programmer. If one is willing to undertake some language design and considerable compiler and run-time system modification, persistent objects can be added as a new type, or a new class of types (persistent records, persistent arrays, persistent sets, persistent lists, etc.) along the lines of the E language [Richardson and Carey, 1987].

Through such extensions one can provide a persistent version of any type, but true orthogonal persistence is independent of type. If persistence is orthogonal, then there are not two types, `Record` and `PersistentRecord`. Rather, any record may be persistent if it needs to be. Persistence is frequently defined in terms of reachability through chains of pointers from persistent roots, not unlike the reachability used in garbage collection.

Orthogonal persistence can be implemented more easily for *heap based* languages; languages that (at least conceptually) create and store all data in a dynamic storage area, the heap. Smalltalk and Trellis/Owl are procedural object oriented languages that are heap based, but do not provide persistence. CLU [Liskov et al., 1981, Liskov et al., 1977] is also a heap based language, but it is not object-oriented, nor does it provide persistence. Argus [Liskov and Scheifler, 1983] is an extension of CLU, which provides a form of persistence. The various Lisp object systems are also heap based, since Lisp itself is heap-based.

Orthogonal persistence for a heap based language is achieved by modifying the memory management subsystem of the client language to handle object faulting and use Mneme as its object store. This can be accomplished, for Trellis/Owl and Smalltalk at least, without changing the client language or its compiler in drastic ways, though it requires more effort than just providing Mneme as a package that the application programmer invokes explicitly.

3 Design of Mneme

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

3.1 Files

A *file* contains a collection of *objects*. Within a given file, an object is identified by a unique, non-time-varying identifier, which we call a *persistent identifier*, or PID. PIDs are

about 30 bits long, so a file may contain on the order of one billion objects. In general, a file can be accessed simultaneously by multiple users; concurrency control is discussed in more detail below. Mneme also supports references between objects that reside in different files, with cross-file reference controls available to Mneme users if desired.

Since the same PID may be used in more than one file, users of Mneme are presented with *client identifiers*, CIDs, to name objects. These identifiers are short (about 30 bits), but uniquely identify each object available during a given session of interaction with Mneme. This design allows one client session to manipulate on the order of one billion objects. Note that Mneme imposes no size constraint on the *total space* of objects, since there is no limit on the number of files that may be constructed.

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.

3.2 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 without sacrificing efficiency. A handle stores a more time-efficient run-time representation of an object than a CID, though it consumes more space.

As one means of "getting started" in accessing objects within a file, each file has a *root object*. Operations are provided to obtain a CID or handle for the root object of a file, as well as to change which object is the root of a given file. The root object for a file will generally be some form of dictionary object, though a client may install any object whatsoever as the root of a file.

3.3 Objects

Objects are the fundamental abstraction provided by Mneme. Objects have three parts: *slots*, *bytes*, and *attribute bits*. Slots are 32 bits wide, and can hold a reference to another object: a PID when stored, a CID when presented to the user. A cross-file reference is represented by a reference to a specially marked object in the same file; these specially marked objects are called *forwarders*. Instead of containing an object reference, a slot may

be empty, which is indicated by the value 0. A slot may also contain an immediate value. Immediate values are always negative; thus the sign bit is used as a tag and determines the meaning of the remaining 31 bits in a slot.

The bytes part of an object is simply an array of uninterpreted 8-bit values. Bytes must not be used to store references to other objects for at least two reasons. First, separating the slots allows automatic and transparent CID/PID translation. Second, restriction of the location of object references leaves open the option of garbage collecting the Mneme object space. Garbage collection is not mandatory, however, since explicit deletion of objects is also supported.

Attribute bits are used to mark special properties of an object, such as being read-only or being a forwarder. We provide at least 8 attributes in every object format; if more than 8 bits would be useful we can provide additional object formats with more attribute bits.

An object can have an arbitrary number of slots and/or bytes. The initial number of slots and bytes of an object is determined when the object is created, but the sizes can be changed during the lifetime of the object. Large objects might not have contiguous representations at a low level, but the handles of Mneme will hide that fact from the user.

Mneme objects are typeless: any types for Mneme objects are imposed by the user. Mneme does, however, allow for the type tags used by many heap based languages, by offering an efficient storage representation for objects having a single slot. In sum, Mneme provides a general and useful kind of object, but maintains considerable neutrality as to the semantics of objects. Clients that assume different semantics should not use the same files. To help guard against such errors, Mneme files can have properties, string values, associated with them, which the client may inspect and update to support a variety of semantic checks.

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

Since locking individual objects is prohibitively expensive, in the implementation we group objects into segments, transferring and performing concurrency control on segments rather than objects. Segments are also the basis for object clustering. The implementation has considerable latitude in concurrency control strategies. 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. 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 user interface makes 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, Mneme provides some support for more sophisticated serializable, or even non-serializable, interaction. The basic concepts include *volatile objects* and *object logs*. A volatile object is accessible to other users and may be changed whenever the user does not have a handle on it, regardless of transactions. Further, a user's uncommitted changes will be visible to other users, and vice versa. In effect, handles provide mutual exclusion on volatile objects, and that 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 users. The change can be hidden by the directory code, providing serializable directories, or the code can let the change show through, providing non-serializable behavior.

Thus, volatile objects and object logs allow one to build user-defined 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.

3.5 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 pool types, 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.

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 back-end immediately, or it might be retained until the back-end requests its return.

The pool concept is crucial to making Mneme useful across a range of design appli-

cations, because the ability to extend policies and to use multiple policies simultaneously will be necessary in achieving adequate performance.

4 Comparison With Other Work

We compare Mneme with 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.

4.1 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 functional-

ity, 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.

4.2 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., 1987b], 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.

5 Software Engineering Applications

There are at least three ways in which Mneme is related to software engineering. First, to the extent that Mneme is integrated with a programming language, it simplifies writing applications in that language that deal with long lived data. In fact, the primary motivation

for Mneme is to make it easier to build such applications. Thus, Mneme can be viewed as a software engineering support tool of itself, if the word *tool* is construed broadly.

Mneme also provides the facilities to build better tools more conveniently. Specifically, we see Mneme as an important part of tools that use structured data to assist in the software development process. Mneme's support for small objects and pointer chasing make it an attractive facility for storing such items as parse trees and module interconnection graphs. In fact, Mneme provides the basic object management facilities required for Persistent Graphite, an extension to Graphite [Clarke et al., 1986] currently under development. Graphite is a tool for building Ada programs that deal with graph structures. The improvements Persistent Graphite offers over Graphite include better handling of graphs with shared substructure, object faulting rather than monolithic reading and writing of entire graphs, concurrent use, reliability, and distribution.

This functionality is achieved without true language integration; i.e., the Ada compiler and run-time system internals are not modified. Persistent Graphite itself was a bit difficult to program, but users of Persistent Graphite need not concern themselves with this fact. Tools that are built with an integrated persistent programming language will not only provide convenient and semantically elegant interfaces to their users, but will also be easier to build. Mneme assists in both efforts.

A third way in which Mneme can be applied to software engineering is as the memory system for an entire integrated software development environment rather than support for individual tools. The multiple file capability and the support for concurrency, reliability, distribution, and cooperation all make Mneme an attractive base for an environment. If the environment is heterogeneous in the sense of using multiple programming languages, then the flexible interface that Mneme provides is especially advantageous. Mneme's neutrality with respect to type systems is also beneficial for experimental software engineering systems, since it allows the experimenter to devise new type systems, even ones in which the types of object can change dynamically. In conclusion, Mneme supports software engineering by providing enabling technology for language-database integration, for individual software development tools, and for heterogeneous and integrated software development environments.

6 Conclusion

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 both the front-end and back-end interfaces of Mneme. We are simultaneously designing and implementing heap interfaces for Trellis/Owl and for our VAX[2] Smalltalk. Implementation will then proceed with the second and third phases, in which we provide for reliability and then sharing.

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

6.2 Areas of 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?

References

[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*, pp. 245–310, Cambridge University Press, Cambridge, England, 1984.

[Banerjee et al., 1987a] 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.

[Banerjee et al., 1987b] 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 Transactions on Office Information Systems 5*, 1 (January 1987), 3–26.

[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.* Technical Report TR-86-07, University of Texas at Austin, Department of Computer Sciences, March 1986.

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

[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, October 4–8). ACM, New York, 1987, 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*, pp. 225–243, Cambridge University Press, Cambridge, England, 1984.

[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 International Conference on Very Large Databases* (Kyoto, Japan, September 25–28). ACM, New York, 1986, 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, August). IEEE, New York, 1986.

[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, September 23–26). ACM, New York, 1986, pp. 26–37.

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

[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., ch. 14, pp. 251–270, Addison-Wesley, 1983.

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

[Liskov and Scheifler, 1983] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381–404.

[Liskov et al., 1977] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8 (August 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.* Technical Report, 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 Transactions on Office Information Systems 5*, 1 (January 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, December 27–29). ACM, New York, 1987, pp. 208–219.

[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, September). ACM, New York, 1986, pp. 9–16.

[Schmidt, 1977] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems 2*, 3 (September 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, September 23–26). ACM, New York, 1987, pp. 196–204.

[Spector and Schwarz, 1984] Alfred Z. Spector and Peter M. Schwarz. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems 2*, 3 (August 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.* Technical Report CMU-CS-86-166, Carnegie-Mellon University, Department of Computer Science, 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). ACM, New York, 1986, pp. 340–355.

[Stonebraker et al., 1976] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of Ingres. *ACM Transactions on Database Systems 1*, 3 (September 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 (April 1981), 25–33.

[Weihl and Liskov, 1985] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 244–269.