FramerD: Representing knowledge in the large

by K. Haase

Content-aware media applications require rich, interconnected descriptions of media content. FramerD is an object-oriented database developed at the MIT Media Laboratory to support just such descriptions in a scalable and distributed manner. Conventional databases, even the newer object-oriented ones, focus mostly on data structures whose components are scalar or literal values, rather than references to other objects. FramerD is optimized to support objects that have components consisting of references to other objects in an environment where the storage of and the operations over the objects are distributed over local- and wide-area networks. This paper discusses the design, implementation, and performance of FramerD, and sketches some current applications of the system.

If the most recent revolution in media systems has been the shift to digital representations of such media content as video and audio, the oncoming revolution lies in the computer representation of the "meaning" of this content. Such representations are crucial if our media systems are to organize and filter the growing mass of available content or to learn from users' actions in response to the information they are provided. Good decisions and good generalizations all rely on good representations; the design of such representations and the infrastructure enabling them is necessary in any future media architecture.

For the last five years, the Machine Understanding Group at MIT's Media Laboratory has been exploring questions about media representations and the infrastructure required to support them. This paper describes FramerD, the current "state of our art" in this work on infrastructure for "understood media."

FramerD is a representation system providing a network protocol for exchanging complex structured objects, a simple persistent object store for managing large databases of complex objects, and a collection of tools for manipulating and indexing large semantic network databases. FramerD grew out of an attempt to apply past work in artificial intelligence (AI) to new media systems. By and large, work in AI before 1990 was characterized by:

- Small problem sets (usually dozens, at most hundreds, of examples)
- Limited domains of discourse (usually some particular human activity or problem class)
- Brittleness (when inputs varied from expectations or domains briefly widened)

While these restrictions are tolerable when tasks can be narrowly defined, they are unacceptable in the more open-ended realm of media systems. Media can be about *anything* and this universality complicates the application of AI techniques to experimental media systems. This complication is further exacerbated by the demands of interactivity. Among Lippman's[1] requirements for "good interactivity" are: interruptability, graceful degradation, and impression of infinity—all of which are confounded by the brittleness and limitations of traditional AI models. In

Figure 1 Layered architecture of FramerD
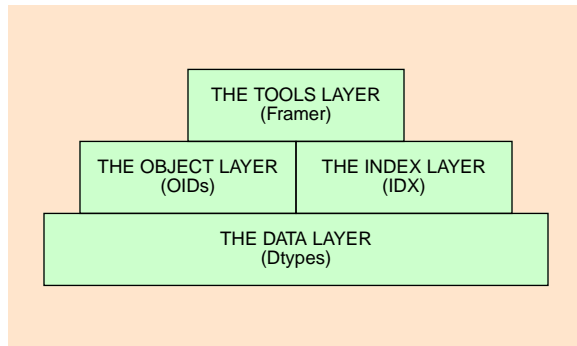
order to prototype intelligent interactive media systems, we required:

1. Large numbers of examples (millions or tens of millions)
2. Broad coverage (more than a single domain)
3. Robustness (in the face of both scaling up [in database size] and scaling out [in domain coverage])

FramerD was designed to address the first and (in part) third of these points. Our work on analogical representations[2] addresses the second and (in part) third points by relying on a rich library of cases to provide breadth, flexibility, and robustness. The current version of FramerD robustly and efficiently supports semantic networks with up to millions of nodes on platforms ranging from workstations to personal computers to high-end personal digital assistants. This scale of representation allows heavily memory-based approaches to understanding, reasoning, and interaction.

FramerD was also designed with portability and distribution of data and function in mind. Part of this was a practical concern: we wanted to be able to move functions (such as background batch processing of databases, real-time interaction with users, or development and prototyping) easily between machines (fast memory-loaded machines, lightweight desktop machines, slower machines supporting good development environments, etc.). At the same time, we did not want this division of labor to get in the way.

These concerns were addressed by introducing a common data level and RPC (remote procedure call) protocol that are used by all the different applications. Our

applications routinely use data and function servers spread across the network. This also allows the construction of very lightweight clients relying on networked servers for most representational processing.

The FramerD layer cake

Like most large software systems, FramerD can be characterized by a layer cake diagram, as shown in Figure 1.

FramerD's layered architecture supports its use as either a data transport and storage system, a data organizing and indexing system, or an inferentially rich expert system. Applications using FramerD at one level can easily make use of higher-level facilities when circumstances or the emerging task structure make it convenient or important.

The lowest level, the data level, provides recursively composable structures and primitives: vectors, lists, numbers, strings, interned[3] symbols, etc. Via the Dtype protocol, these complex objects can be transmitted and buffered much as a representation like Sun Microsystems' XDR (external data representation, which underlies their RPC and NFS [network file system] protocols) handles simple strings and arrays. The data layer also provides for *object references* defined by 64-bit unique object identifiers.

The next layer, the object layer, provides for the mapping of these references into data layer values. This mapping can be imagined as providing a 64-bit address space whose cells are complex structured objects (including pointers elsewhere in the 64-bit space). In parallel with the object layer, the index layer provides for the maintenance of large inverted indices whose keys and values are arbitrary Dtypes. Both indices and object repositories are readily distributed across networks and architectures.

The tools layer uses the object and index layers to provide basic representational facilities via a frame-based language.[4–6] These functions include slot-based inference, object indexing, and fast analogical matching.

A brief history. In 1992, Nathan Abramson at the Media Lab implemented Dtypes, a simple network protocol for exchanging LISP-like objects. This was used in news software being developed at the time. A year and a half later, Klee Dienes took the Dtype package and reimplemented it in C++, adding a LISP

interpreter (for the Scheme dialect of LISP) to allow the use of interpreted programs or scripts within and around the Dtype universe. Recently, this implementation has matured and has been ported to Java** under the guidance of Dan Gruhl.

On a parallel track, the Lab's Machine Understanding Group had been developing a series of AI languages designed to be accessible to a large non-AI community and to support fairly large databases. The first such language, ARLOtje, was developed as a teaching tool but ended up being used in a number of research projects.[7,8] One situational problem was that much of the Lab's work was done (for reasons both cultural and technical) in the C programming language,[9] while the AI tools we developed were generally written in Common LISP.[10]

In the spring of 1992, we began the implementation of Framer,[11] a set of representation tools in LISP and C that allowed data to be shared between them. Framer incorporated a scripting language based on Scheme[12] with the addition of special provisions for dealing with sets and collections of arguments and return values. Framer also was used in a variety of prototype systems at the Media Lab[13–15] and was the structure underlying our first work on relational retrieval from large databases.

The development paths of Framer and Dtypes collided in the spring of 1994, when FramerD was first developed (originally in LISP) by implementing procedures for reading and writing Dtype representations from LISP. This soon led to the implementation of Dtype servers in LISP, allowing technology developed in the Machine Understanding Group (for instance, a broad-coverage natural language parser) to be accessed by Dtype programs written by other groups.

At the time of FramerD's development, the Dtype protocol was being considered as a foundation for the Lab's Media Bank[16] project. This required a database facility for saving and retrieving Media Bank objects. The first version of this technology used the GNU[17] project's GDBM (GNU database manager) library, but this had several problems, the least of which was that GDBM files were not generally portable across machine architectures.

The Machine Understanding Group ended up developing its own portable database manager library and database servers, which were interoperable with existing Dtype servers. At this point, we had also extended

Digital Equipment Corporation's Scheme-to-C system[18] to include access to FramerD Dtype objects and servers. This was used as the basis for a number of Web-based demonstrations and for the next generation of our text retrieval and matching work.

In the spring of 1995, the FramerD implementation was beginning to show some cracks. In particular, we found that it was less efficient than the original Framer for large-scale projects (in Framer, much conscious attention was paid to scaling and storage management issues) and that many common representational functions were quite slow. This led to the implementation of the current version of FramerD. In the following sections we consider its design and rationale.
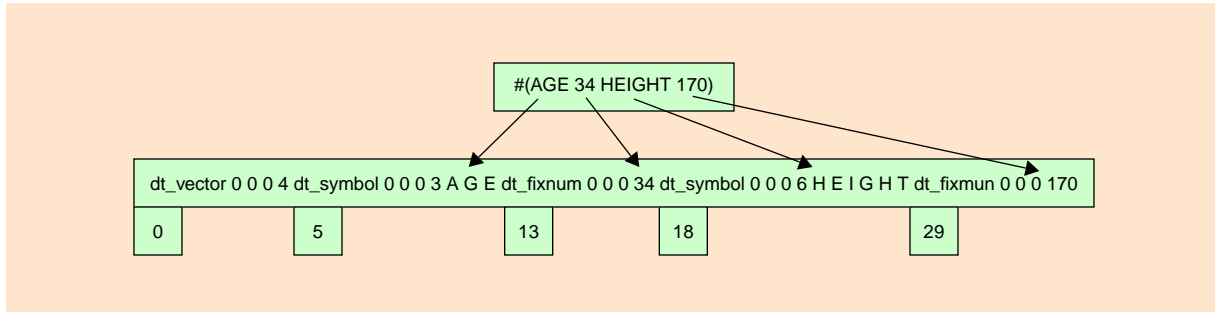
Dtypes: The data level

Dtypes are designed for an environment where programs routinely use data structures with complicated interconnections implemented by address pointers. This is foundational in languages like LISP or Scheme and a common programming practice in C, C++, and numerous other languages. These kinds of structures are especially important for describing large semantic networks, where most data values fall between nodes. Dtypes are designed to provide an external representation for these internal structures and must support:

1. Fast conversion from external format to internal structures
2. Compact representation (in terms of disk space and bandwidth requirements)
3. Portability between C, C++, LISP, and Scheme (at least)
4. Lightweight state-free parsing and production

Requirements (3) and (4) were important because of our desire for architectural and political portability to encourage adoption by other research projects within the Lab, at other laboratories, and within sponsor companies.

Why binary representation? For their internal structures Scheme and LISP already provide a printed representation designed to be easily readable by both people (with some acclimation) and machines. The problem with the format is that some information about the structure, useful for reconstituting objects (for instance, numbers of characters or elements, precision of numeric values, etc.), is implicit, rather than explicit, in the printed representation. For instance, a

Figure 2 Layout of a basic Dtype representation

#(AGE 34 HEIGHT 170)

dt_vector 0 0 0 4 dt_symbol 0 0 0 3 A G E dt_fixnum 0 0 0 34 dt_symbol 0 0 0 6 H E I G H T dt_fixmun 0 0 0 170

| 0 | 5 | 13 | 18 | 29 |

string in the printed representation has the form ""characters"" where the number of characters in the string (and the number of bytes in memory needed to store it) is implicit in the specification.

This printed representation is the only external representation provided by LISP and Scheme. By contrast, schemes like ASCII (American National Standard Code for Information Interchange) and EBCDIC (extended binary-coded decimal interchange code) provide an external representation that, although not easily readable by people, is very readable by machines and can be useful for internal representation as well.

The standard printed representation for Scheme and LISP fails to satisfy requirement (1) for Dtypes. Furthermore, many of the quantities in the programs' internal representations have natural binary representations. For these reasons we chose an external format that was binary: thus, the length of the vector or the value of a small integer can be directly written or read without parsing some intermediate digital representation.

Some machine architectures, for instance Intel** processors and Digital's Alpha** chip, store 4-byte numbers in memory with the least significant byte first, while others (the Motorola 68000 and the Power-PC* chip) place the most significant byte first. In order to ensure portability among machines with different byte orderings, the Dtype representation normalizes all 4-byte codes to place the most significant byte first. Thus the protocol is state-free: it is not necessary to know the byte ordering of the machine to which you are transmitting (or from which you are receiving) a Dtype representation for an object.

Dtype representations. Dtype representations are either basic (consisting of a 1-byte type code followed by a data field) or packaged (consisting of a 1-byte package code followed by another byte or two of extended type code and a data field). The data field has five possible formats:

- Zero length, where the object is completely specified by the type code, is used for representing void values and the empty list.
- Fixed width in bytes, where the data field has a fixed number of bytes based on the type code alone, is used for representing small integers, floating point numbers, object identifiers, and Boolean values.
- Fixed width in Dtypes, where the data field consists of some number of Dtype representations, is used for pairs, compound tagged data types, and error and exception values.
- Variable width in bytes, where the data field starts with 1 or 4 bytes of size information followed by the specified number of bytes, is used for strings, symbols, bignums (arbitrary precision integers), arbitrary packets of bytes, and various multimedia representations.
- Variable width in Dtypes, where the data field starts with 1 or 4 bytes of size information followed by the specified number of Dtype representations, is used for vectors, property lists, and nondeterministic sets.

Figure 2 illustrates a basic Dtype representation. At offset 0 is the type code for a vector. The next four bytes hold the number of Dtype fields that follow. At offset 5 is the type code for a symbol, followed by 4 bytes that hold the length of the symbol, then the symbol itself. At offset 13 is the byte code for a small inte-

Figure 3   Packaged Dtype representation
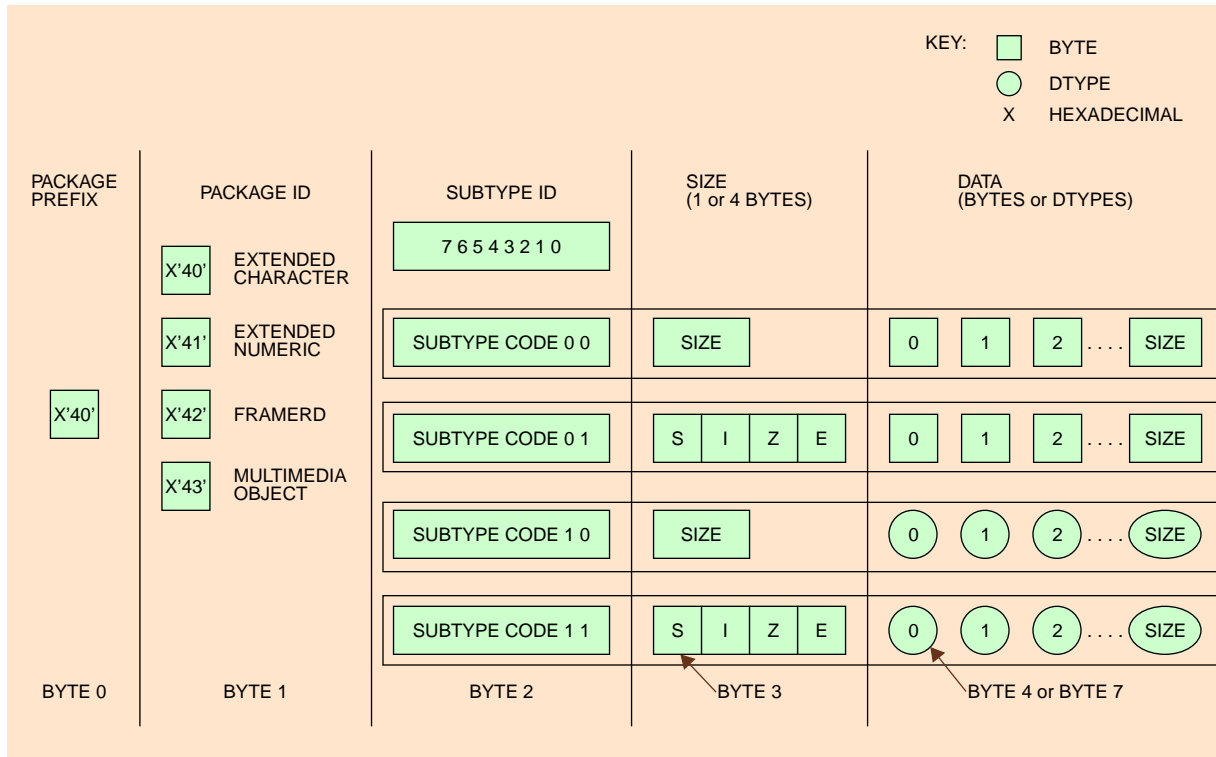
ger followed by 4 bytes that contain the small integer. Another symbol and small integer follow.

One important property of Dtypes is their support for opaque data types. This means that an application can implement its own data types while using standard facilities for the carriage and storage of the data type. The systems doing the carriage and storage do not need to know how the data type is implemented, but can generically handle the Dtype representation generated by the implementing client(s).

Opaqueness is implemented in two ways in the Dtype representation. Most generally, a client data type can be described by the *compound Dtype* type tag followed by a tag (a Dtype) and some data (another Dtype). The tag indicates the type of the object and can be either a symbol or, more portably, an object reference. The data field contains another Dtype structure from which the object can be recreated.

A more efficient and specific form of opaqueness is available through the *package extensions* to the Dtype

representation. The type codes greater than 127 are reserved to identify "packages" of extended types. A packaged Dtype representation consists of a 1-byte package prefix, a 1-byte package identifier, a 1-byte subtype identifier, a 1- or 4-byte size field, and a specified number of either bytes or Dtype representations. The two highest-order bits of the subtype identifier describe whether the size is in Dtypes or bytes and whether the size field is 1 or 4 bytes, as shown in Figure 3. This allows packaged Dtypes to be read, stored, and written without knowing anything about the object represented.

Currently, four packages exist in various implementations:

• Extended numeric types (bignums, rational and complex numbers, polynomials, and integer and floating point vectors)
• Extended character types (ASCII and Unicode** characters and Unicode strings and symbols)
• FramerD types (property lists and nondeterministic sets)

Figure 4 Objects and pools
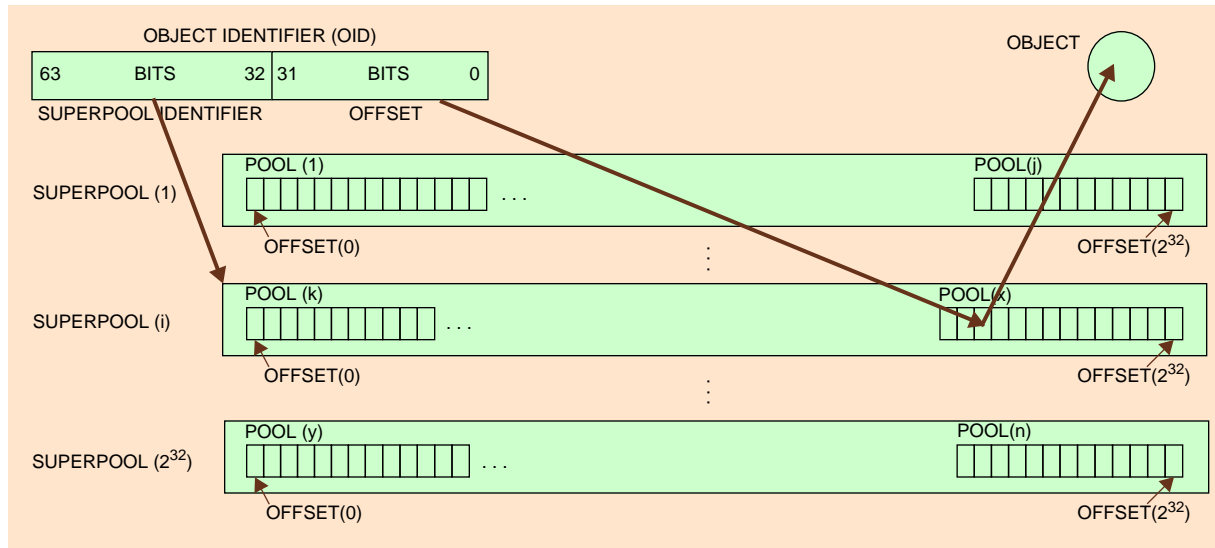
- Multimedia types including standard image and video formats: JPEG (Joint Photographic Experts Group), MPEG-1 (Moving Pictures Experts Group), MPEG-2, AIFF (Audio Interchange File Format), etc.

Three new data types. In addition to the standard LISP data types, FramerD introduces three special types: the object identifier, the feature vector, and the result set. Object identifiers consist of 8 bytes and are used by the FramerD object system, which maps object identifiers into actual data-level objects. The object system's implementation is described in the next section.

Feature vectors are sets of attribute/value pairs, much like LISP property lists[10] and are used for constructing frame and semantic-network representations.

Result sets are unstructured collections of other data-level objects (but not of other result sets). They were first introduced in the design of Framer[11] as a way to avoid having to specify explicitly whether a function or accessor returned or contained a single result or a set of results. When combined with a nondeterministic interpreter,[19,20,11] this allowed the elegant expression of many search and composition operations. FramerD's primitive scripting language supports the same nondeterministic function application.

Related technologies. Nearly every LISP system implements some kind of binary data format, usually used for storing compiled programs and data. The earliest versions were MacLisp's "FASL (fast load) files," and subsequent LISP implementations have provided some similar functionality. Dtypes differ from these systems in being portable and in not supporting any kind of compiled code (except that user objects could be implemented for compiled procedures).

The chief virtue of this Dtype representation is supposed to be its portability, a property that arises as much from political and cultural compatibility as from technical issues. To this end, we are making the LISP, Scheme, and C implementations of FramerD available for personal and research purposes with no fee. The code can be downloaded from the laboratory's FTP (file transfer protocol) site from the directory pub/framerd at the host ftp.media.edu. This site includes sources for the LISP and C versions and executable code for a dozen hardware platforms.

Pools: The object level

The FramerD object system is organized around 8-byte object identifiers (OIDs) used to uniquely identify objects. These identifiers define an address space segmented into disjoint *superpools*, which are further

segmented into disjoint *pools*. Figure 4 shows the relationship between superpools, pools, objects, and OIDs. The first 4 bytes of the OID determine the superpool. The last 4 bytes give the offset within the superpool of the object's address, which points to the object's data. The OID of another object might be included in the object's data to implement an association between the objects.

A pool is identified by its *base* (OID of the first object in the pool) and a capacity (the total number of object addresses in the pool). Capacities come in powers of 2, so the offset in an object identifier can always be split into a pool identifier and an offset into the pool. However, this segmentation is not defined syntactically (as in, for example, Internet addresses) but is determined during the lookup process.

The use of 64-bit identifiers, as opposed to symbolic names, arises from two distinct demands. First, FramerD is intended as an infrastructure capable of supporting and sharing a range of knowledge bases developed by different researchers at different sites. This goal demands some mechanism for separating name spaces, to allow independent development while still allowing references across name spaces. One approach would be to use a simple package system, such as Common LISP's,[10] with a registry for package names. The problem with this approach is that it then becomes necessary to do a pair of string lookups in order to resolve an object reference. With simple 64-bit identifiers, references can usually be resolved by a few array references rather than a pair of string lookups. While this is only a constant-time improvement, it is an important one, since the semantic networks are mostly objects containing object references.

A second demand is that *gensyming*, the creation of new object identifiers, needs to be efficient. For instance, our text-processing system generates a node for every phrase of the text it is processing, and some representation schemes require a node for every slot value generated. When programs are generating representations, as opposed to simply using them, such generation must be fast. A standard approach to gensyming is the use of a simple counter to generate object names with a numeric component, but when this is done, it is still necessary to make sure that the generated name does not collide with any existing object.[21] This assurance requires a search to confirm that the gensym is unique. In contrast, with numeric identifiers a counter can be used directly as part of the object name, ensuring uniqueness. A simple way to

think about the object database is as a flat 64-bit address space whose locations contain arbitrary Dtype representations. Object identifiers make it possible to transmit and share *references* to objects without having to send or necessarily share the objects themselves. This is important for the kind of richly interconnected structures involved in content understanding. For instance, in most semantic networks, there is some path between any pair of nodes: if we loaded a node whenever it was referenced, it would be necessary to load the entire network whenever we loaded a single node. An object system thus needs to provide:

1. Lazy reference (an object can be referred to without being present)
2. Distributed interoperable data access (to files and across networks)
3. Lazy storage (an object will not be updated unless it has changed)
4. Access control (for security and revision control)
5. Data integrity (in the distributed environment)

FramerD currently provides functions (1) through (3) to our satisfaction. It currently provides (4) by a simple locking mechanism for files of objects or individual objects located on a server. An area of current development, (5) is partially resolved by exclusive locking and partially by the use of journalling to ensure server integrity.

Varieties of lazy reference. When we make a distinction between a reference to an object and the object itself, we need to also make a distinction between the operations we are supporting on references and the operations that require an actual object.

For instance, a minimal operation on references would be the ability to pass references as arguments and return them as values without relying on access to the actual argument. There is not much value in having a distinction between references and objects without this minimal assumption. However, beyond this point, choices must be made.

For instance, is fast identity comparison supported for object references? If it is, we must keep a table of object references and look up new references in the table to ensure that the same reference will turn into the same object pointer. If fast identity comparison is not supported, we must go through some calculation (probably less than loading the whole object) to determine reference identity. Another reason for maintain-

Figure 5 Object marks and tags: Morphology
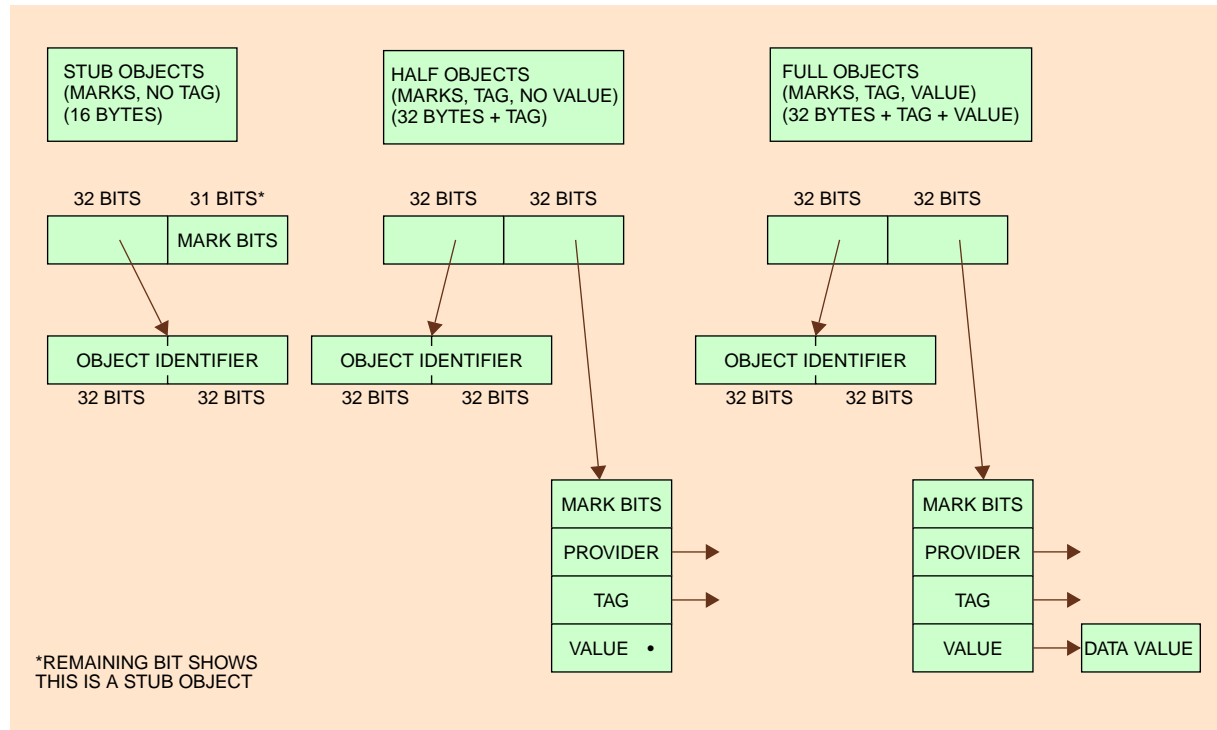
ing reference identity is space allocation: if we ensure that all pointers to the same object are associated with a single memory address, we are ensured that each object will take a constant space.

Maintaining object identity introduces the object reference as a subclass of objects in general: object references are objects whose full description is not in memory. They are "virtual objects" in the same sense that blocks of memory swapped to disk are "virtual memory."

If we are maintaining object identity, we have the question of whether there are any other operations (besides simple comparison) that we want to support on object references. In FramerD, we provide two: marking and tagging. Marking consists of associating object references with a small number of global sets; the membership of an object in a set is determined by a bit field associated with each reference. (See Figure 5.) These sets are explicitly allocated and deallocated and are local to the process manipulating the reference.

Tagging is like marking in that it provides an ephemeral annotation for object references. While marking involves bit fields, the tag is a pointer to some data-level object (possibly another object reference). Tagging provides a reusable mapping, from objects to values, that is independent of the object's structure and does not require that the object itself be loaded.

In FramerD, we implement three levels of object "presence":[22]

1. A stub object consists of the object identifier and a bit field.
2. A half object consists of the object identifer, a bit field, a tag field, and fields for supporting the retrieval of the object's actual data-level value.
3. A full object is a half object whose structure has been loaded (either from disk or the network).

This three-level division was arrived at based on experiences with the earlier Framer implementation. There is a basic trade-off between more levels of presence and more steps and structure for object access. We

examined how the performance and memory requirements of applications changed with different models of object presence. We found that fast set operations were important enough to merit some basic marking capability on all object references; other algorithms required object tagging, so we implemented that as a second level; and finally, the value of the object reference is loaded only when that value is needed for some operation.

What is it good for? We can see the value in this multilevel data representation by considering the following algorithm for maintaining a flat index of objects. We assume a function, *keys(X)*, that maps from an object X to a set of key features associated with X, and an inverse function, *coindices(k)*, from key features to the objects possessing them.

This algorithm implements a variant of the vector-space retrieval model,[23] where objects are considered only if they have exactly two keys in common, and each coincidence of keys is weighted by the inverse frequency of the key in the corpus. By using marks and tags, this algorithm computes this function with much less memory than would otherwise be required.

The core of the algorithm in pseudocode is:

```
To find objects similar to object Q:
Let A be a marked set of objects, initially empty
Let score(x) be a tag on object x
Let N be the maximum value of |coindicies(k)|
For all the keys k in keys(Q)
   for each e in coindices(k)
     if (score(e)) is defined
        set score(e) = score(e) + N/|coindex(k)|
     otherwise if e is in A,
        set score(e) = N/|coindex(k)|
     otherwise place e in A
return the scored objects sorted by their score
```

The total number of references accessed by this procedure is

$$\left| \bigcup_{k \in keys(Q)} coindices(k) \right|$$

If it were necessary to load each of these objects, retrieval would be quite expensive. For instance, in our current 10-million-word text database, the total number of distinct references accessed for a simple query is in the hundreds of thousands; loading an object can take anywhere from milliseconds on up, depending on the latency and bandwidth of the disk or

network. Such comprehensive algorithms are thinkable only with the ability to mark and manipulate objects *without* loading them.

Distributing storage. As mentioned earlier, FramerD divides its address space into disjoint chunks called *pools*; pools are used to manage the storage of individual objects. Each pool has a *provider*, which is either a disk file or a network server. The pool for an object can be found in *log N* time (where N is the total number of pools).

For a variety of reasons, the current implementation limits pool sizes to integral powers of two and forbids pools to cross 32-bit boundaries. This simplifies the process of locating the pool for a particular object identifier.

Pools are also used to organize access and concurrency control. Access control means managing the relationship of clients to pools and the operations they are permitted to perform on both the pool and the objects it contains. Concurrency control means managing conflicts between requests.

A simple, portable, file-locking mechanism is used to control modifications to file pools. Different processes can easily share read access to the same file, but only a single process can modify the objects stored in the file. Because individual processes cache object values (gaining a significant performance enhancement), this locking mechanism provides no guarantee of read/ write consistency. However, it does guarantee that changes to objects in file pools will not be lost by different processes overwriting the same object. When files are not being shared, the file-locking mechanism is usually transparent to users, since modifications of objects automatically lock the corresponding file pool. Only when a program terminates abnormally without clearing its locks, or when files are actually shared, does the locking system for file pools become visible.

File pools can also be declared read-only by the local protection facilities of the file system, in which case the protections on the file extend to the objects stored in that file.

Networked pools operate differently, supporting locking at the level of individual objects. Whenever an object provided by a remote server is about to be modified, a lock request is issued to the server, together with an identification for the current process. If the lock request succeeds, the server returns the

object's current value and notes internally that the object is locked. When the object is saved (either explicitly or when the session ends normally), this lock is freed through a request validated by the pro-

> **Fine-grained transaction control is primarily important in the presence of critical conserved quantities.**

cess identification. In addition to this normal freeing mechanism, a lock can be forcibly removed, though this may lead to inconsistencies. The server does not allow two processes to share a lock on the same object; one must relinquish the lock before another can claim it. The automatic update of the current value when an object is locked guarantees that changes will not be lost due to caching of outdated values.

While most servers support the locking scheme above, servers can also be *read-only* or *chaotic* (unadvisedly providing write access without locking). A read-only server with an extended repertoire of commands can provide carefully limited access to a pool of objects. For instance, in our text-understanding work, one server provides a "rhizome" database consisting of unique entries for English words and their possible meanings. The rhizome server is read-only, but it also provides specialized commands for creating new word entries. This approach keeps the database from generating inconsistencies or redundancies but still allows remote clients to extend the database as needed.

FramerD does not provide any more fine-grained concurrency control, such as the transaction-based mechanisms used in conventional databases. This sort of mechanism is difficult to implement in FramerD for a variety of reasons. First, lazy reference makes heavy use of caching, which makes operations on an object, once it is loaded, local and fast. Requiring that actions go through a central arbiter would confound this feature. The integration of inference into the database also complicates the definition of an "atomic" transac-

tion. In noninferential schemes, it is easy to tell whether two changes conflict: they disagree on an assignment to some logical location. However, in inferential schemes, recognizing a conflict may involve an arbitrary amount of computation. Finally, FramerD is likely to be used in contexts with intermittent connectivity, where we cannot (for instance), assume transaction commitment within any particular window. Though there may be transaction schemes that address all of these problems, we do not have one. However, fine-grained transaction control is primarily important in the presence of critical conserved quantities (like account balances); for other kinds of data, FramerD's locking and sharing of databases may suffice.

Data integrity. The locking scheme just described ensures that different client programs will not inadvertently erase each other's changes. However, changes might be lost in other ways. In particular,

1. A client program may terminate abnormally before writing its changes to a file or server.
2. A server may terminate abnormally (possibly because of a client request) before changes have been committed to the disk files or to other servers that it references.
3. A client or its server may terminate while changes are being transmitted from one to the other.

FramerD addresses these issues as follows:

Issue (1) is a problem when the client has locally cached data that have been changed. However, we argue that the integrity of these data is the client's problem and that it may, in fact, be better that the incomplete state of the client program be lost. The client may choose to checkpoint itself in some fashion, but that is not the responsibility of FramerD. The remaining problem here is the interaction of crashed clients and the locking mechanism. When a client terminates in an untimely fashion, its outstanding locks may not be cleared. Several functions exist for cleaning up after such terminations, but further development is warranted to make such occasions infrequent.

Issue (2) is a serious problem, since a client can have sent all of its changes to the server, but if the server crashes without saving those changes, they will be lost without the client's knowledge. To address this, we introduce the notion of journal servers. A journal server saves every transaction to a file and periodically (based on both real time and number of transac-

tions) reconciles its state on disk with the saved transactions. When a journal server starts up, it checks for transactions in the journal file, indicating an unexpected termination. At this point it reprocesses saved transactions in order to bring the database up to date, saves its state, and clears the journal file. To provide useful debugging information, the journal server writes out requests before they are processed and replies after they are processed. While reprocessing the requests in the journal file, the server is able to report the transaction during which the unexpected termination occurred. Furthermore, it compares the past results of transactions with new results to ensure that the current state is consistent with that reported to clients.

Issue (3) is especially tricky and currently is not addressed. The likely solution will be to allow bundled transactions, where a series of changes are sent together to a server, and if a response is received, all of the changes will have been recorded.

FramerD: The tools level

FramerD implements a frame-based "representation language language" (RLL) derived from those described by Greiner[24] and Haase,[25] which are in turn derived from frame languages developed by Stefik[6] and Goldstein.[5] Following Stefik, we refer to "frame objects" as "units" whose "slots" contain "values." Other implementations have referred to "slots" as properties or attributes, but we choose "slots" to emphasize their implementational, rather than epistemological, character. Slots are used in implementing properties, attributes, relations, etc.

In previous representation languages and representation language languages, objects were uniquely associated with symbolic names and slots were also uniquely identified by such names. This reflected an underlying implementation that used LISP symbols and their "property lists" to associate units with values. Among representation language languages, this convention was used to define the properties and implementation of particular slots using the unit with the corresponding name: the unit "Color" defined the restrictions, inferences, and presentational information for the "Color" slot.

The association of a particular symbol with an object was useful in providing a human handle on a system's objects. However, for the reasons of namespace collision and fast gensyming already described, FramerD adopts numeric identifiers but provides a second layer for indexing symbolic names to objects.

Units in FramerD are objects (with 64-bit identifiers) whose values are "feature vectors" mapping "feature names" to values. The feature names are either symbols or object references. When the feature name is an object reference, accesses to the attribute are mediated by the corresponding object. This mediation can include the automatic generation of values for a slot, the execution of side-effects when the slot's value is changed, specialized tests for membership in a slot, or specialized display of values.

Basic operations. The basic operations on units and slots are:

1. Getting the values for a slot
2. Testing whether or not the slot includes a value
3. Adding an object to the values for a slot
4. Removing an object from the values for a slot

The default behaviors for (1), (3), and (4) simply access the value associated with the slot in the feature vector. The default behavior of (2) uses the behavior of (1) (it gets the values and then checks for membership).

When the slot is itself an object, the default behaviors are modified by "demons" associated with the slot description. These demons are expressions in a scripting language stored in simple symbolic slots named (respectively) get-methods, test-methods, add-demons, and remove-demons. When a value is gotten, tested for, added, or removed, the corresponding demons are executed. The expressions are in a restricted subset of Scheme with no provision for procedure definition. These expressions can be easily transformed into or generated from ALGOL-style expressions, but no such interface currently exists.

Behavior-modifying expressions are evaluated in an environment where variables relating to the ongoing operation are bound: unit, slot, and data are bound to the unit and slot being operated upon and the value directly associated with the slot in the feature vector. For testing and modifying operations (e.g., (2) through (4) above), the variable value is bound to the particular value being tested for, added, or removed.

For example, the following objects implement the slot's advisor and advisees as inverses, using add and remove demons appropriately:[26]

```
The object @Advisor
   Obj-name: "Advisor"
   Add-demons:
      (add value @Advisees unit)
   Remove-demons:
      (remove value @Advisees unit)
The object @Advisees
   Obj-name: "Advisees"
   Add-demons:
      (add value @Advisor unit)
   Remove-demons:
      (remove value @Advisor unit)
```

The extension language provides conditionals and local binding as well as a number of special representation operators. It is easily extended by procedures in the native environment (LISP, Scheme, or C) and expressions containing undefined symbols or procedures (in the current environment) are simply ignored (possibly with an error).

Stack tracking. One problem with demon execution is the unbounded recursion that results when one demon invokes another, which then reinvokes the first demon. We handle this by the novel approach of dynamically tracking active slot operations and transforming recursive repetitions of the same operation into null operations. Thus, if we assert

<@Ken @Advisor @Marvin>,

that in turn asserts the inference

<@Marvin @Advisees @Ken>,

which then asserts

<@Ken @Advisor @Marvin>,

but since that operation is already active, it does nothing. This same principle, applied to getting the value of slots, allows the specification of recursively defined methods that may be quite efficient. For instance, the following three definitions implement length, width, and area slots:

```
@Length
   Obj-name: "Length"
   get-methods:
      data
      (/ (get unit @Area)
         (get unit @Width))
@Width
   Obj-name: "Width"
```

```
   get-methods:
      data
      (/ (get unit @Area)
         (get unit @Length))
@Area
   Obj-name: "Area"
   get-methods:
      data
      (* (get unit @Length)
         (get unit @Width))
```

Even though the slots' methods refer to one another recursively, a calculation path fails whenever it requires a recursive reference. Note also that one of the methods simply involves accessing the data associated with the slot (through the variable data). These slots can be seen to operate in the following cases, where the nonitalicized values were given and the italicized values were computed automatically.

```
@Rectangle1
   Obj-name: "Rectangle1"
   @Area:    100
   @Length:  10
   @Width:   10
@Rectangle2
   Obj-name: "Rectangle2"
   @Area:    35
   @Length:  7
   @Width:   5
@Rectangle3
   Obj-name: "Rectangle3"
   @Area:    242
   @Length:  11
   @Width:   22
```

When a slot description has multiple methods, they are combined disjunctively: the results of all of a slot's "get-methods" are combined with one another, and testing for a particular value returns true if any of the slot description's "test-methods" returns true.

An extremely simple form of inheritance is implemented by the works-like slot of slot descriptions. If a slot description has no specialized methods for a particular operation but it does have a works-like slot, the value of this slot (which should be another slot description) is checked for its own specialized methods, which are then used for the operation.

The inference capability of the systems are at what Lenat and Guha[27] call the "heuristic level"; they consist of procedures that operate on data structures directly. The particular modes of inference depend on the basic functions provided.

Nondeterminism. Another novel and important feature of the extension language is its use of "nondeterminism" in evaluation. Procedures operating on "result sets" automatically iterate over the elements of the set. Result sets are either stored in values explicitly, generated by user procedures of various sorts, or explicitly introduced by the procedure either. The implicit iteration means that an expression like:

(get (get unit (either 'father 'mother)) (either 'brothers 'sisters))

returns the aunts and uncles of the person described by "unit" without having to explicitly express the iterations or worry about the singularity or multiplicity of slots such as "mother" or "brothers." The scripting language also provides set operations (union, intersection, and difference) applicable to result sets that use the marker bits described earlier to achieve linear-time performance over sets of objects.

The use of nondeterministic results adds an interesting twist to the kinds of interdependent slot definitions just demonstrated. If a description is inconsistent, the values of the slots contain result sets corresponding to the possible consistent values for each slot, given the other slot values:

```
@Funny_Rectangle
   Obj-name: "Funny Rectangle"
   Area:        100  125
   @Length: 25   20
   @Width:   5     5
```

Again, nonitalicized entries are given and italicized entries are derived. This is not, of course, a full-fledged constraint reasoning system, since there is no identification of which combinations of these six values are actually valid. Nonetheless, the very presence of multiple values identifies an inconsistency that the removal of a given value will repair.

Inference. An inference is the appearance of structure in the database that was not originally literally described. In a classic example, if the system knows someone's parent and that parent's parent, it can determine that a grandparent relation exists, even though no such relation was literally given. Inferences like these can be done either "eagerly" or "lazily": eager inferences happen as soon as their preconditions are asserted; lazy inferences happen only when the inferred fact is requested by the user. Practically, lazy inferences are implemented by the get and test methods and eager inferences by the add and remove demons.

For instance, suppose we want to implement slots for mother, father, and parents (their union). An eager way to implement these slots would be to have mother and father be slots with demons that add values to or remove values from the parents slot:

```
@Parents
   Obj-name: "Parents"
@Mother
   Obj-name: "Mother"
   add-demons:
      (add unit @Parents value)
   remove-demons:
      (remove unit @Parents value)
@Father
   Obj-name: "Father"
   add-demons:
      (add unit @Parents value)
   remove-demons:
      (remove unit @Parents value)
```

Whenever we asserted that X is Y's mother, the "add demons" for the mother slot would assert that X is also Y's parent. If we subsequently removed X from Y's "mother" slot, we would likewise remove X from Y's parents slot.

A lazy way to implement these slots would be to have mother and father have no demons associated with them, but to have the "methods" slot of parents combine the mother and father slots:

```
@Parents
   Obj-name: "Parents"
   get-methods:
      (get unit @Mother)
      (get unit @Father)
@Mother
   Obj-name: "Mother"
@Father
   Obj-name: "Father"
```

In this case, the parents slot would be generated only when needed, rather than always being updated whenever the mother or father changes.

One problem with eager inference is *truth maintenance*; if two different inference paths lead to the same conclusion, we must be certain that invalidating one does not invalidate the other. The standard solution in FramerD is to store one value on a slot for each justification (each separate invocation of add by users or demons) and to remove one value for each justification. This approach, while it usually works, manifests some annoying problems when a user removes an inferred value without removing its antecedent.

For instance, suppose we assert that Hera is Poseidon's mother, which leads to the assertion that Hera is also Poseidon's parent. We then step in and retract the derived parent relation without retracting the original motherhood relation. Time passes and we learn, in classic soap opera style, that Hera used to be a man and was actually Poseidon's father. This asserts the parent relation all over again, but when we subsequently retract the motherhood relationship, it erroneously retracts the parent relation derived from fatherhood. A full solution to this would involve keeping track of why each value is present, but this would involve substantial overhead that we currently choose to avoid.

Because of this problem and the overhead of eagerly making structural inferences, lazy inference is usually preferable. However, in some cases, eager inferences are necessary for reasons of efficiency and completeness. For instance, inverse slots (children and parents, husband and wife, left and right) are best implemented eagerly because computing them "after the fact" requires iteration over very large sets (e.g., finding X's husband might involve searching all people to find one whose "wife" slot is X).

FramerD provides a handful of functions to simplify the writing of methods and demons. First, the functions get, test, add, and remove implement the basic operations listed above. The function pathp searches for a path between two objects through a particular slot (or set of slots). The function either, described earlier, introduces a set, and the functions union, intersection, and difference combine sets in straightforward ways (actually union and either are identical).

Indexing. As mentioned earlier, FramerD provides an alternative method for mapping object names to objects. This indexing facility is quite general, allowing any data-level object to be mapped to a number of other data-level objects (including object identifiers). The facility is designed to make adding a new mapping for an existing key very efficient. This makes the facility good for maintaining large inverted indices. The approach uses hash tables and combines (optional) in-memory caching with data stored on disk, allowing indices to be incrementally modified but quickly accessed. It also allows large tables to be used without a corresponding large "footprint" in memory.

The indexing facility can be used for caching values of complicated procedures or for guaranteeing a cor-

respondence between externally unique lexical identifiers and internally unique object identifiers. For instance, in a prototype intelligent electronic-mail system, the indexing facility is used to maintain a map to the descriptions of qualified addresses and message identifiers.

This general facility is also used to implement a *frame indexing* system that indexes frames by slot values. It ranks frames by similarity, based on overlapping slot values, and has been used quite effectively with our 7-million-frame database. An active area of current research is a comparative evaluation of different methods for computing similarity based on overlapping slot values. We have found that the conventional approaches of information retrieval (for instance, weighting by inverse frequencies) are not optimal for object indexing, since quite common slot values may still be important distinguishing characteristics for objects. Given this, we are exploring formal models for object indexing and their possible implementations.

Performance

Evaluation of FramerD's performance is difficult without some precise characterization of the tasks to which it is being applied, and there is, as yet, no standard set of benchmark tasks for large knowledge bases. To make a preliminary evaluation, we implemented a simple benchmark in C, intended to be characteristic of the operations usually performed over semantic networks. We then monitored the program's performance over a series of random trials, tracking time expended, objects loaded, and objects referenced (to measure the effect of object caching).

The benchmark operation, count-common, operates on a hierarchical ontology and counts the number of parent nodes in common to two children. As with many operations over semantic networks, this operation requires many links to be followed, but it also includes repeated references to nodes higher in the hierarchy, making caching especially advantageous. We also use the marker bits attached to objects to speed up the algorithm. In future work, we will implement program variants that do not use either caching or marker bits, in order to assess how efficacious the corresponding performance improvements really are, though some conclusions can be drawn from the benchmark performance itself.

The ontology used is derived from WordNet[28] and consists of 214 317 nodes, comprised of 122 584

Figure 6 Seconds per frame reference over 250 trials
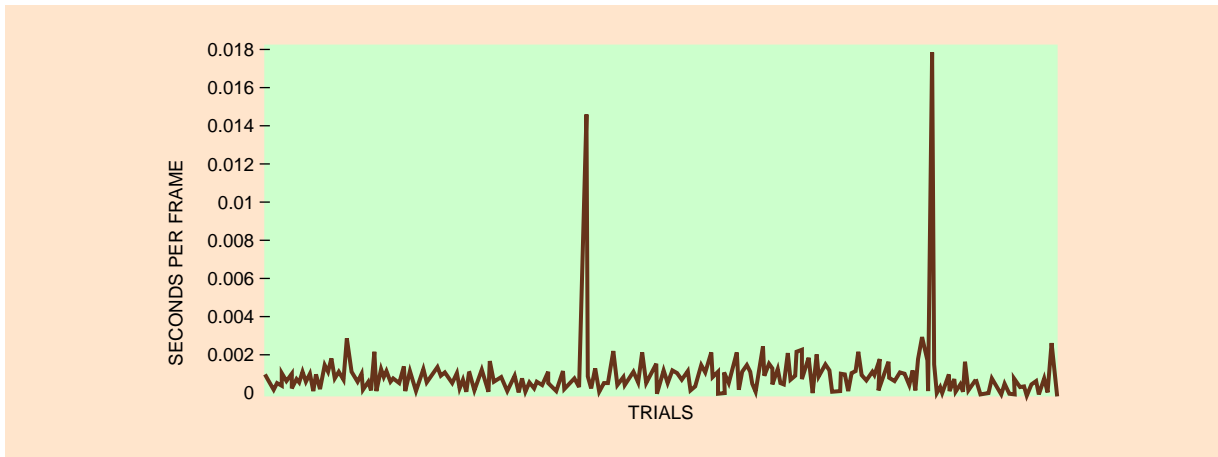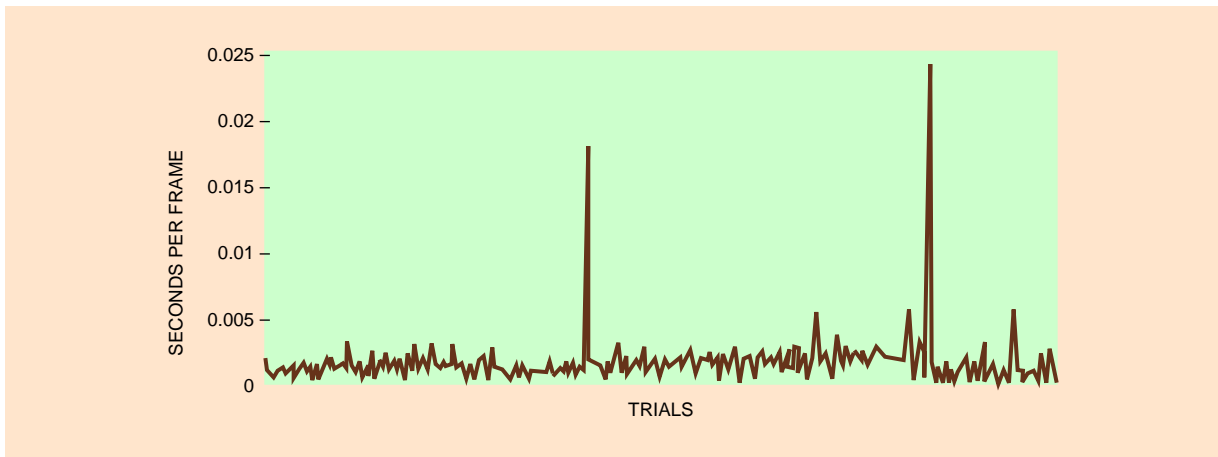

Figure 7 Seconds per frame loaded over 250 trials

nodes representing individual words and collocations (e.g., "fly" or "fly fishing") and 91 733 nodes representing *synsets*: sets of synonyms that can be used interchangeably in particular contexts and thus represent possible word meanings. The relations in Word-Net are both represented separately and conflated into a single hierarchical relation used for the benchmark. These objects are relatively small, averaging about six slots with an average of eight values each.

Two hundred fifty consecutive trials were run on pairs of objects chosen at random. The benchmark was run on a 90-megahertz Pentium** processor under Microsoft Windows** NT with 64 megabytes of

memory, and compiled with Microsoft's Visual C/C++**. We ran the benchmarks using both a database stored on the local disk as a file pool and a database served by a remote server (an Alpha** 3000/500 processor). The server was restarted before the benchmark to avoid any server-side caching effects. For each trial, we divided the real time taken for the trial by the number of objects referenced or loaded. In Figure 6, these are plotted against each individual trial. The random seed used to select random objects was set to be the same for the network and local disk trials.

In Figure 7, we see the frame reference times for FramerD with local disk access. The average refer-

ence time over the entire run was 0.0025 seconds. We would expect to see the performance on references increase over time due to caching, but it turns out that loads are so fast that the advantage of caching over a run this small (a few hundred objects) is not substantial. The average time was 0.0048 seconds per loaded frame.

The networked database had identical performance numbers, 0.0025 seconds per frame referenced and 0.0048 per frame loaded. The similarity of this average case performance and the simultaneous appearance of two spikes in the timing data suggests that our timing may be compromised by both the task itself and the precision of the timing function under Windows NT. Nonetheless, this performance is satisfactory for our current applications.

While there is little performance data on large-scale knowledge bases with which to draw a comparison, a recent paper on one approach[29] describes a hybrid database management/frame system that averaged roughly 0.1 seconds per loaded frame, substantially slower than FramerD. However, such comparisons are mostly of practical rather than theoretical significance, since the experimental platforms were substantially different (compiled LISP on a SPARCstation** 10 vs C on a Pentium/90 processor).

Summary

FramerD provides a simple but robust representation for knowledge and databases comprised of millions of objects. Novel elements of its design include:

- A portable binary object format supporting networked computation and multimedia data types
- The separation of naming (maintaining uniqueness) from indexing (determining identity)
- Multiple access libraries allowing programs in LISP, Scheme, C, or C++ to share data and representation
- The division of object storage across multiple files and servers
- The introduction of nondeterminism (the either operator) to simplify slot operations
- An embedded representation language language supporting nonsymbolic slots

We are releasing FramerD to the public to encourage its adoption as a standard for large representational databases. Future work will include the deeper integration of multimedia types and the implementation of generic browsing and manipulation systems.

Cited references and notes

1. S. Brand, *The Media Lab*, Viking Press (1987).
2. K. Haase, "Matching Texts for Information Extraction," *WordNet: An Online Lexical Database and Its Applications*, MIT Press, Cambridge, MA (1996).
3. "Interned symbols" are structures representing strings where lexical equality of the strings implies "pointer equality" of the structures.
4. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, Patrick Winston, Editor, McGraw Hill, New York (1975).
5. I. Goldstein, *FRL: A Frame Representation Language*, AI Memo 333, MIT, Cambridge, MA (1976).
6. M. Stefik, "An Examination of a Frame Structure Representation System," *Proceedings of the Sixth International Joint Conference on Artifical Intelligence*, Tokyo (1979).
7. R. MacNeil, "Generating Multimedia Presentations Automatically Using TYRO, the Constraint/Case-Based Designer's Apprentice," *Proceedings of the IEEE '91 Workshop on Visual Languages*, Kobo, Japan (1991).
8. A. Bruckman, *The Electronic Scrapbook: Towards an Intelligent Home-Video Editing System*, M.S. thesis, MIT, Cambridge, MA (1991).
9. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, NJ (1978).
10. G. Steele, *Common Lisp: The Language, Vol. II*, Digital Press, Newton, MA (1990).
11. K. Haase, "Framer: A Persistent, Portable, Representation Library," *Proceedings European Conference on AI*, Amsterdam (August 1994).
12. J. Rees and W. Clinger "The Revised^3 Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices* **21**, No. 12 (December 1988).
13. G. Davenport and T. A. Smith, "The Stratification System: A Design Environment for Random Access Video," *ACM Workshop on Networking and Operating System Support for Digital Audio and Video*, San Diego, CA (1992).
14. A. Pentland, R. Picard, G. Davenport, and K. Haase "Video and Image Semantics: Advanced Tools for Telecommunications," *IEEE Multimedia* **1**, No. 3, 73–75 (Summer 1994).
15. M. Davis, *Media Streams: Representing Video for Repurposing and Retrieval*, Ph.D. thesis, MIT, Cambridge, MA (1995).
16. A. Lippman and R. Kermode, "Media Banks: Entertainment and the Internet," *IBM Systems Journal* **35**, Nos. 3&4, 272–291 (1996, this issue).
17. GNU (a recursive acronym for "GNUs not UNIX") is a project of the Free Software Foundation to implement quality versions of standard software that can be freely distributed and modified.
18. J. Bartlett, *Scheme→C: A Portable Scheme-to-C Compiler*, Research Report 89/1, Digital Western Research Laboratory, Palo Alto, CA (1989).
19. R. Zabih, D. McAllester, and D. Chapman, "Non-Deterministic LISP with Dependency-Directed Backtracking," *Proceedings of the National Conference on Artificial Intelligence*, Seattle WA, July 13–17, 1987, pp. 59–66.

20. J. M. Siskind and D. A. McAllester, "Nondeterministic Lisp as a Substrate for Constraint Logic Programming," *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C. (July 11–15, 1993), pp. 133–136.

21. For instance, names of the form "eat#1," "eat#2," "eat#3" could be generated to distinguish new objects. However, this runs into collision problems if either a user or a program wants to name an actual object "eat#2," or if two separated users or programs are both generating such names independently.

22. Our current C implementation uses a library of functions for allocating Lisp-like structures. This includes support for maintaining large numbers of simple vectors containing two address-sized values (e.g., four bytes on 32-bit machines, eight on 64-bit machines). An "object stub" consists of one such vector. The first element is either a direct 8-byte object identifier (on 64-bit machines) or a pointer to another vector containing the object identifier. The second element consists of either a pointer to a "detail structure" (aligned on even memory addresses) or an odd integer, which is used to store the bits associated with marked sets.

23. G. Salton, "A Theory of Indexing," *Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics* (1975).

24. R. Greiner and D. Lenat, "A Representation Language Language," *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford, CA, August 18–21, 1980, pp. 165–169.

25. K. Haase, "ARLO: Another Representation Language Offer," B.S. thesis, MIT, Cambridge, MA (1984), also available as MIT AI Lab Technical Report 901).

26. The syntax @Name refers to an object with the mnemonic name "Name"; in the actual implementation, the object is uniquely identified by a 64-bit object identifier though several interfaces hide this reference behind the mnemonic name. In the examples here, tokens with identical printed representations (like the several occurrences of @Width) indicate the same object.

27. D. Lenat and L. Guha, "CYC: Building Large Knowledge-Based Systems," Digital Press, Newton, MA (1990).

28. G. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM* **38**, No. 11, 39–41 (1995).

29. P. Karp and S. Paley, "Knowledge Representation in the Large," *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Francisco, CA (1995), pp. 751–758.

Kenneth Haase *MIT Media Laboratory, 20 Ames Street, Cambridge, Massachusetts, 02139-4307 (electronic mail: haase@ media.mit.edu).* Dr. Haase is an associate professor of media arts and sciences at the MIT Media Laboratory. His undergraduate degree is in philosophy, while both his M.S. and Ph.D. degrees are in electrical engineering and computer science; all three degrees were awarded by MIT. His doctoral work, supervised by Marvin Minsky, was in the area of computational and formal models of representation and creativity. His current work involves knowledge representation, analogy, natural language understanding, and machine and human creativity.