, , 1–45 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Collaborative Environment for Authoring Large Knowledge Bases [*]

PETER D. KARP [**] pkarp@pangeasystems.com
Pangea Systems, 4040 Campbell Avenue, Menlo Park, CA 94025

VINAY K. CHAUDHRI vinay.chaudhri@sri.com
Artificial Intelligence Center, SRI International, 333 Raenswood Avenue, Menlo Park, CA 94025

SUZANNE M. PALEY paley@pangeasystems.com
Pangea Systems, 4040 Campbell Avenue, Menlo Park, CA 94025

Abstract. Collaborative knowledge base (KB) authoring environments are critical for the construction of high-performance KBs. Such environments must support rapid construction of KBs by a collaborative effort of teams of knowledge engineers through reuse of existing knowledge and software components. They should support the manipulation of knowledge by diverse problem-solving engines even if that knowledge is encoded in different languages and by different researchers. They should support large KBs and provide a scalable and interoperable development infrastructure. In this paper, we present an environment that satisfies many of these goals.

We present an architecture for scalable frame representation systems (FRSs). The Generic Frame Protocol (GFP) provides infrastructure for reuse of software components. It is a procedural interface to frame representation systems that provides a common means of accessing and modifying frame KBs. The Generic KB Editor (GKB-EDITOR) provides graphical KB browsing, editing, and comprehension services for large KBs. Scalability of loading and saving time is provided by a storage system (PERK) which submerges a database management system in an FRS. Multi-user access is controlled through a collaboration subsystem that uses a novel optimistic concurrency control algorithm.

All the results have been implemented and tested in the development of several real KBs.

Keywords: Knowledge bases, knowledge base management systems, knowledge representation, storage management, concurrency control, application programming interface (API)

1. Introduction

Collaborative knowledge base (KB) authoring environments allow multiple, geographically distributed users to collaborate in the development of large KBs. The first generation of FRSs [26] provided only single-user KB authoring environments whose engineering limitations constrained the size of the resulting KBs, and did not permit distributed KB access. This report describes a next-generation, reusable environment for collaborative KB authoring that consists of the following components:

[*] To obtain the software described in this paper, contact the second author at Vinay.Chaudhri@sri.com.
[**] The work presented in this paper was done while the first and the third authors were at SRI International.

- The Generic Frame Protocol, which provides infrastructure for software and knowledge reuse. It is a procedural interface to FRSs that provides a common means of accessing and modifying frame KBs.
- The GKB-EDITOR, which provides KB browsing and editing services for large KBs.
- The Storage Subsystem, which provides scalable storage of frame KBs in a commercial DBMS.
- The Collaboration Subsystem, which provides optimistic concurrency control over the KB updates made by multiple distributed KB authors.

This work makes a number of contributions to the field of knowledge base management systems. We identify design requirements for collaborative KB authoring environments, and present an architecture that satisfies those requirements and an implementation of that architecture. The architecture includes both the Storage Subsystem, for expanding the storage capabilities of FRSs, and novel optimistic concurrency control techniques for coordinating KB updates made by multiple simultaneous users. The GKB-EDITOR completes this authoring environment by providing four different viewer and editor utilities for browsing and modifying large KBs. All of these tools have been used in the development of several real-world KBs, including a planning ontology, and the EcoCyc biochemical-pathway KB, which contains 12,000 frames.

These results are made more significant because the implementations have been reused across several FRSs. The Storage Subsystem has been used with LOOM [33], THEO [36], and OCELOT[39]. (In principle, the storage system can be reused with a larger set of FRSs. The reuse examples given here are based on the implementations done by us.) the GKB-EDITOR has been used with LOOM, THEO, SIPE–2 [53], OCELOT, and partially with ONTOLINGUA [22] and CLASSIC [5]. These reusability results support the generality of our approach, and provides the most substantial example to date of the type of software reuse envisioned by the Knowledge Sharing Initiative [41, 37].

2. Design Requirements

We derived the design requirements for KB authoring environments from our experiences with KBs in several domains. We present two scenarios of the use of these environments to illustrate the motivations behind these requirements.

Scenario 1 involves the distributed development of a planning ontology (An ontology for a domain specifies commonly used terms and their definitions. For example, an ontology of a University domain may define terms such as student and professor [21].) by multiple DARPA contractors who are distributed throughout the United States, as part of the DARPA/Rome Laboratory Planning Initiative. Contractors at roughly a dozen sites might want to access the LOOM KB that implements a planning ontology, to browse the current state of the ontology, to add new class definitions interactively, and to edit existing definitions. The LOOM classifier runs during the editing process to infer relationships among new and existing classes. Users might also execute ontology translators to convert the planning ontology to other forms, such as a relational database.

Scenario 2 involves a biological KB that describes the biochemical reaction network within the *E. coli* cell. The KB models biological objects such as enzymes and biochemical pathways. Expert biologists from around the world interact with the KB in different ways. Some biologists want to update the KB by editing that region of the KB that falls within their expertise. Other biologists require only read-only access to the KB. They will browse and query the KB using a graphical interface. Others will execute qualitative and quantitative simulation programs that model the metabolism of the cell. Other scientists will execute programs that redesign the biochemical network of *E. coli* for commercial purposes in bio-technology. Still other users will evaluate queries over the KB to answer scientific questions.

We have extracted the following design requirements from these scenarios.

- The system should efficiently support several distinct patterns of KB operations:
  - Interactive browsing and editing sessions that access a small portion of a KB. Editing sessions may last hours or even days, and include changes to many different frames.
  - Computations such as simulation, design, and expert systems that repeatedly access significant subsets of the KB.
  - Traditional database like queries that return small amounts of data to the application.
- The system should be capable of handling next generation of KB sizes that will range from 10^4 to 10^7 frames.
- The system should support multi-user access. The updates made by multiple users may interact, and they must be synchronized. Users may be geographically distributed, and may access the KB by using long-distance Internet connections.
- The schema-evolution capabilities of most FRSs should be retained. KB schemas are usually larger, more complex, and more dynamic than those for databases. As KBs grow, we expect their schemas to grow also, as well as the need to modify class and relation definitions dynamically.

3. Architecture

Our system architecture is shown in Figure 1. The GKB-EDITOR is a graphical tool for interactive KB editing and browsing. The GKB-EDITOR is reusable across multiple FRSs because all of its KB-access operations are implemented using the Generic Frame Protocol (GFP). GFP provides a uniform procedural interface to FRS applications. Our architecture includes a storage subsystem that allows KBs to be stored within an ORACLE DBMS. Frames are retrieved on demand from

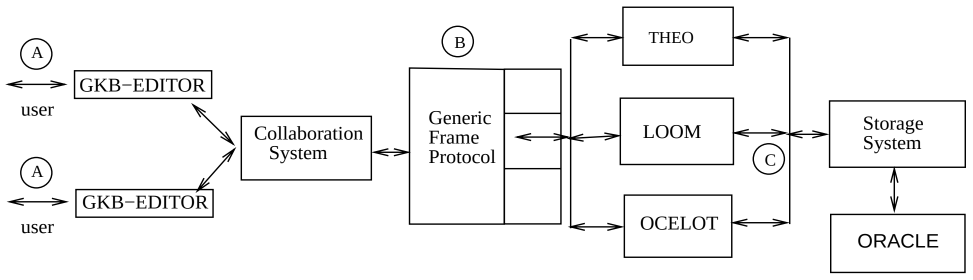


Figure 1. System architecture

the DBMS into an FRS that acts as a client. The storage subsystem records what frames have been modified, and can incrementally save modified frames to the DBMS. Updates from multiple users are synchronized using a collaboration system based on a concurrency control method that detects the conflicts between the updates made by a user and recent updates performed by other concurrent users. Unlike traditional database techniques that keep an item locked for the whole duration of a transaction, we use locks only while the conflict-free updates are being deposited in the database.

The architecture of Figure 1 allows distributed operation because network links can be inserted between components at several places. We can insert a network link at (C) by transmitting Structured Query Language (SQL) calls to the DBMS server over a network. We can insert a network link at (B) by creating a remote-procedure call version of GFP, in which every GFP call is sent over a network. We can insert a network link at (A) by allowing the X-window graphics of the Gkb-Editor to flow between an X client and X server.

In most functional implementations of our system, the network link has been inserted at C. The Gkb-Editor, collaboration system, GFP, and the FRS act as a client program and the DBMS acts as a server. This approach allows computationally intensive applications such as planners, design programs, and simulators to operate in parallel on multiple client machines, on regions of the KB that are cached in the local memory of those clients. AI applications tend to make a very large number of KB accesses, sometimes accessing the same frames repeatedly.

4. Generic Frame Protocol

The goal of GFP is to allow reuse of knowledge and software by mixing and matching knowledge and software components in a large AI system. By reuse of knowledge, we mean transforming a knowledge base or ontology developed in one FRS in another FRS, for example importing an ontology developed in Loom into Ontolingua. By reuse of software, we mean an easy porting of the software across FRSs, for example, reusing a system such as the Gkb-Editor across multiple FRSs. GFP accomplishes its goals by defining a set of Common Lisp generic functions that constitute a common application-program interface (API) to FRSs. These functions constitute a wrapping layer for FRSs; that wrapping layer is implemented by creating FRS-specific methods that implement GFP operations.

4.1. Design Principles of GFP

The following principles guided our design of GFP.

- **Simplicity:** The protocol should be simple and reasonably quick to implement for a particular FRS, even if this means sacrificing theoretical considerations or support for idiosyncrasies of that FRS.
- **Generality:** The protocol should apply to many FRSs, and support the most common FRS features.
- **No legislation:** The protocol should not require substantial changes to an FRS for which the protocol is implemented. That is, the protocol should not mandate the method of operation of an underlying FRS.
- **Performance:** Inserting the protocol between an application and an FRS should not introduce a significant performance cost.
- **Consistency:** The protocol should exhibit consistent behavior across implementations for different FRSs, that is, a given sequence of operations within the protocol should yield the same result over a range of FRSs.
- **Precision:** The specification of the protocol should be as precise and unambiguous as possible.
- **Language independence:** Ideally, GFP should be independent of programming language.

Satisfying these objectives simultaneously is impossible because many of them conflict. Different FRSs behave differently, and unless we mandate a minutely detailed behavioral model for FRSs (which no developers will subscribe to anyway), we cannot force these systems to exhibit uniform behavior. GFP uses a knowledge model that encompasses many FRSs and is detailed enough to be useful in practice, but is not so detailed as to exclude every FRS from its model. Another example of conflicts among our objectives is that to precisely specify the semantics of the GFP function that retrieves the values for a frame slot, we must specify the inheritance semantics to be used. However, different FRSs use different inheritance mechanisms [26]. Conformance with a specific semantics for inheritance would require either altering the inheritance mechanism of a given FRS (violating the no-legislation goal), or emulating the desired inheritance mechanism within the implementation of the protocol (violating performance and generality, since the inheritance method used by that FRS is inaccessible through the protocol).

Currently, GFP has many Lisp dependencies. Connecting GFP to an FRS implemented in some other language should be straightforward using Lisp foreign-function calls. Also, precision and generality are conflicting goals. If we precisely model various characteristics of the FRSs in GFP, the protocol will not be general, because the detailed features of one FRS may not be shared by another.

4.2. Proposed Design of GFP

The most important decision in the design of GFP was whether GFP should be the "least-common-denominator" or a "superset" of a class of FRSs. In the least-common-denominator approach, GFP would contain only those features that are supported by most of the systems. In the superset approach, it would aim to support the union of the features of all FRSs. Our design is a hybrid of the two approaches described. The core design of GFP is based on the least-common-denominator approach. It can be parameterized in various ways to capture the peculiarities of an FRS that are not covered in the core model. We, however, did not design GFP to be a superset of all FRSs, because that would make the protocol cumbersome, complex, and difficult to use.

The design of GFP is based on a comprehensive survey of FRSs [26]. The survey revealed a large variety of system designs. Some of the differences among these systems were significant, whereas others were superficial. We identified those commonalities that are useful for a broad range of applications. Let us consider the knowledge model of GFP in more detail.

4.2.1. Representational Primitives A *frame* is an object with which facts are associated. Each frame has a unique name. Frames are of two kinds: classes and instances. A *class* frame represents a semantically related collection of entities in the world. Each individual entity is represented by an *instance* frame. A frame can be an instance of many classes, which are called its *types*. A class can also be an instance, that is, an instance of a class of classes (a *meta-class*).

Information is associated with a frame via *slots*. A slot is a mapping from a frame and a slot name to a set of values. A slot value can be any Lisp object (e.g., symbol, list, number, string). Slots can be viewed as binary relations; GFP does not support the explicit representation of relations of higher arity.

Facets encode information about slots. Some facets pertain to the values of a slot; for example, a facet can be used to specify a constraint on slot values or a method for computing the value of a slot. Other facets may describe properties of the slot, such as documentation. Facets are identified by a facet name, a slot name, and a frame.

A *knowledge base*, or KB, is a collection of frames and their associated slots and values. Multiple KBs may be in use simultaneously within an application, possibly serviced by different FRSs. Frames in a given KB can reference frames in another KB, provided that both are serviced by the same FRS.

4.2.2. Inference Mechanisms Our survey of FRSs revealed that three forms of inferences are most prevalent: *subsumption reasoning*, limited forms of *constraint checking*, and *slot value inheritance*. Consequently, GFP supports only these three types of inference.

In GFP, it is possible to specify and query subsumption (or class–subclass) relationships. Some FRSs require subsumption relationships to be specified when frames are created. In contrast, FRSs that perform automatic classification infer the subsumption relationships by comparing class definitions. GFP operations allow the user to interrogate any class–subclass and class–instance relationships, no matter how the relationships were derived.

GFP recognizes type and number restriction constraints on slot values that are specified as facets. GFP relies on the underlying FRS to check those constraints.

Inheritance in GFP is based on the use of *template* and *own* slots. A template slot is associated with a class frame and may be inherited by all the instances of that class. An own slot can be associated with a class or instance frame and cannot be inherited. Inheritance in GFP can be characterized as follows. The value of a slot in a frame F is computed by combining the values of own slots and template values for that slot in F and the superclasses of F, provided those values do not conflict. GFP allows the use of different semantics for "combining" and "conflict," to support a range of inheritance methods.

4.2.3. Parameterization Using Behaviors GFP supports extensions to the core knowledge model to capture the features of FRSs that are not covered by the core model. This diversity is supported through behaviors, which provide explicit models of the FRS properties that may vary. An application program can query the value of behaviors and use the result in executing code specific to the value of that behavior.

For example, some FRSs may allow a user to define new facets. In such cases, in addition to the standard GFP facets, an application program must take into account the user-defined facets. When an FRS allows user-defined facets, we can set the `:user-defined-facets` to true. An application program can query the value of the behavior `:user-defined-facets`, and if returned true, can execute code that depends on user-defined facets.

4.2.4. Set of Operations GFP defines a programmatic interface of common operations that span the different object types in the knowledge model, namely, *knowledge bases*, *frames*, *classes*, *instances*, *slots*, and *facets*. Three categories of operation are supported for each object type: *retrieval operations*, *manipulator operations*, and *iterators*. Retrieval operations extract information about objects and slot values. Manipulator operations create, destroy, and modify objects. Iterator operations loop over a set of objects.

As an example, consider operations on slots. The most commonly used retrieval operation on slots is `get-slot-values`, which retrieves the values of a slot given a frame name and a slot name. The operation `member-slot-value-p` tests whether a given value is one of the values of the slot of a given frame. An example of manipulator operation on slots is `create-slot`, which is used to define a new slot of a class. To operate on all the values of a slot, one may use the iterator function `do-slot-values`. It can have a body consisting of a LISP expression that is evaluated for each slot value.

In addition, GFP supports operations on behaviors. Retrieval operations obtain information about the behaviors supported by GFP in general, the behaviors that a given FRS supports, and the behaviors that are enabled for a particular KB.

Any integrity constraint violation detected during an operation is signaled using GFP conditions. The GFP specification defines a collection of commonly occurring conditions. In most cases, it is possible to map the error message from an FRS to some GFP condition. GFP also provides a generic condition which can be used when there is no mapping from the error message of an FRS to any of the available GFP conditions.

4.3. Implementation

Each GFP operation is implemented as a Common LISP Object System (CLOS) method. We identified a subset of GFP operations, called the *kernel* operations, which can be used to implement every other operation not in the kernel. We have created a default method for all the nonkernel operations that defines them in terms of kernel operations. For example, the default method for `member-slot-value-p` calls the kernel operation `get-slot-values`. The kernel consists of roughly 30 operations, whereas the total GFP operations are over 200. The default methods can be overridden to improve efficiency or to support better integration with development environments. We can create GFP implementations for new FRSs quickly because only the kernel methods need to be implemented.

GFP back ends exist for LOOM [33], OCELOT, SIPE–2 [53], THEO [36], and ONTOLINGUA [22]. The LOOM and ONTOLINGUA back ends have been used in conjunction with the GKB-EDITOR. A read-only back end exists for CLASSIC [5]. (A more complete back end for CLASSIC is feasible, but we did not attempt it because of limitation of resources.)

A network version of the GFP (NGFP) is also available. NGFP allows us to invoke GFP operations on KBs that are not necessarily resident on a local machine. In Figure 1, NGFP supports the network link at B. The NGFP implementation consists of a client and server that may be running on different machines on the internet and may be written using different programming languages. The clients open a connection to the server by specifying its address, port number, user name, and password. Once a connection has been created, the user can create a KB on the remote server. All subsequent operations that use the remote KB as the KB argument are transparently executed on the remote server. NGFP clients are available for C, LISP, and JAVA.

NGFP implementation supports two performance improvement techniques — client side caching and remote procedures. The client caching facility records all the remote GFP query operations and stores their results in a table. As a result, if a query is repeated, it can be answered locally instead of executing it on the remote server. Whenever there is an update, the cache is flushed. Since updates tend to be infrequent, such a simple scheme for client consistency has proven to be adequate. For an update-intensive environment, more sophisticated caching schemes are necessary [17]. NGFP allows a user to define procedures that consist of several GFP operations. The procedures are executed on the server in one network call instead of one network call for each GFP operation, thus improving the performance.

4.4. Logging Facilities

The GFP implementation provides a facility to capture, in the form of a log, all KB update operations executed in a user session. The log can be used in a variety of ways. For example, in Section 7, we describe how we use the logging capabilities of GFP to support multiuser access to knowledge bases. The log can be used for propagating updates between replicated copies of a knowledge base at multiple sites: the updates are transmitted as a log and applied to the local copies of the KB.

The log can also be used by the GKB-EDITOR to support "undo" for user operations: undoing an operation involves applying, to the current state of the KB, the inverse of every operation in the log. For example, the **`add-slot-value`** operation that adds a value to a slot can be inverted by executing a **`remove-slot-value`** operation that removes a slot value. The logging facility in GFP saves any additional information that is necessary for inverting an operation. or example, the **`put-slot-values`** operation stores a new set of values into a slot, removing any previous slot values. Those previous slot values must be restored when the operation is inverted, therefore those values are saved in the log. Since for every GFP operation, there is an operation that is its inverse, and information necessary to invert an operation is stored as part of the log, supporting undo does not require any additional effort for a GFP user.

4.5. Evaluation of the Protocol

In this section, we discuss some of our experiences in using GFP with LOOM and CLASSIC. The problems identified here should not be viewed as an exhaustive list of difficulties in using GFP with these two systems.

The knowledge model of LOOM fits GFP model fairly closely. For most GFP operations, we were able to identify equivalent LOOM functions. Two difficulties that we faced were capturing the concept–definition language of LOOM, and representing context information. Attributes of LOOM classes are specified through complex definition expressions. As GFP does not support concept definition languages per se, we developed a mapping between the LOOM language and GFP facets. For example, the **`:at-most`** concept constructor in the LOOM concept definition language, which specifies the maximum number of values of a slot, maps to the **`:maximum-cardinality`** facet in GFP that has the same meaning. But there is no way to represent the **`:not-filled-by`** construct of LOOM in GFP (this construct specifies values that are not valid for a slot). In FRSs such as Loom, Cyc, and Ontolingua, one can associate a context with each assertion in a knowledge base. For example, one may state that in the context of theoretical computer science, "Karp" is a professor at UC Berkeley, but in the context of knowledge based systems, "Karp" is a scientist at SRI.

Contexts have a variety of uses such as for storing the alternative world states that are generated by a planner. At present, there is no notion of contexts in GFP, and each GFP KB maps to a single Loom context. This representation does not capture the feature that contexts in Loom can be organized into a hierarchy, and can inherit assertions from the parent contexts.

Classic differs from GFP to a greater extent than does Loom. We consider two examples to illustrate the differences. First, in Classic, a slot must be defined before the concept that uses it, whereas GFP assumes the opposite. This dependency makes it difficult to implement the GFP operation `create-slot`, which has two required arguments: slot name and the classes to which it is attached. A possible solution for this problem is to revise the `create-slot` operation to accept only the slot name as the required argument. Second, Classic supports closed world reasoning by offering an operation for "closing" slots on an instance to indicate that there can be no more values of this slot. Such a behavior can be represented in GFP by introducing a new facet called `:closed` that can have values true or false. Introducing a new facet is a way of extending the knowledge model of GFP. The burden of defining the semantics of a new facet is on the implementor and is not guaranteed by the GFP specification.

A natural question that arises while using GFP with different systems is: how does a user know when to use GFP to access an FRS and when to use the API of the FRS? To achieve maximum portability, GFP operations should be preferred when possible. In some cases, the use of the native API of an FRS is inevitable, forcing us to sacrifice portability. For all the systems that we have built using GFP, and which are reported in the later sections of this paper, the source code can be divided into two components: one that uses only GFP, and the other that uses the FRS-specific API. For example, our storage system requires a byte-string representation of a frame which cannot be obtained using GFP and required us to use the LOOM API. The extent of such code in a system is highly application dependent.

In summary, GFP was able to cover a substantial part of the FRS features that we considered. It was not complete in some cases, but we were able to deal with them by minor extensions to our initial design. Therefore, we believe that GFP effectively meets the goal of a generic API for knowledge bases.

Our experimental evaluation of the GFP back end for Loom indicates that the performance penalty for using GFP is acceptable. The overhead incurred by the use of GFP is largest for fast operations and smallest for slow operations. Using a Loom implementation of GFP, we compared the running times of key GFP kernel operations with the corresponding Loom operations. The results showed GFP operations to be 1% to 50% slower (depending on the operation). The high overhead costs resulted for operations without direct counterparts in Loom. For example, GFP provides an operation for retrieving a frame when given a frame name, but Loom has no such operation. Instead, it provides separate operations for instances and classes. GFP operation must consider whether the name corresponds to a class or an instance in order to invoke the appropriate underlying Loom operation. We note that on an absolute scale, the overhead is very small in this case (approximately 0.02 millisecond).

For directly comparable operations, the upper bound on overhead was 35%. Much of the increased execution time results from activities common to all GFP operations. Thus, the overhead is high on a percentage basis for fast operations such as slot value retrievals (35% for a 0.3-millisecond operation), but low for more expensive operations such as retrieving all instances of a class (1% for a 16-millisecond operation).

Overall, GFP has been effective in supporting the collaborative authoring of FRSs. Its extensive collection of operations was the basis of design of the Gkb-Editor (described in Section 5). In fact, the Gkb-Editor was a driving application for the design of GFP. The logging facility in GFP was inspired by the needs of the collaboration system and played a pivotal role in supporting multi-user access. GFP has found several uses at SRI and KSL. It remains to be seen how well it is adopted by the others in the community.

4.6. Limitations of GFP

The major limitation of GFP is a lack of a formal model as a basis for proving that a compliant GFP application will interoperate with multiple FRSs. We have taken the first step to address this limitation: we have specified the GFP knowledge model using logical axioms [7]. The axiomatic specification precisely defines the basic notions in the GFP knowledge model, for example, inheritance of slot values, association between a frame and slots. That is, however, only a part of the solution, because for most FRSs, it is almost never possible to do a GFP implementation that complies to the specification in every respect. Therefore, we need a way to characterize the degree to which a GFP implementation complies to the specification and the guarantees that can be made about the portability of GFP applications.

Another limitation of GFP is lack of support for logical expressions (including support for relations with arity greater than three) and support for querying a KB for an explanation of an answer returned by it. Many description logic systems support rules or triggers that cannot be represented directly using GFP. These limitations make it difficult to use GFP with systems that perform deductive inference.

4.7. Related Work

Both GFP and Knowledge Interchange Format (KIF) [20] seek to provide a domain-independent medium that supports the portability of knowledge across applications. KIF is more expressive than GFP, as KIF is a comprehensive first-order representation formalism whereas GFP captures a subset of first-order logic that represents class hierarchies. ONTOLINGUA [22] is a set of tools for writing and analyzing KIF knowledge bases along with translators for mapping KIF KBs to specific FRSs. KIF and ONTOLINGUA are declarative representation languages; GFP is a procedural interface for accessing representation structures. KIF and ONTOLINGUA are designed for use in sharing a large corpus of knowledge at specification time, through the use of translators. GFP is designed for runtime access and modification of existing KBs. GFP is similar to Knowledge and Query Manipulation Language (KQML) [41] in that it provides a set of operations defining a functional interface for use by application programs. For example, an agent may query an FRS using a KQML "performative." The KQML allows an agent to express the action of querying, but provides no language to express the query itself. (The query could be expressed using GFP.) Thus, GFP is complementary to KQML. GFP plays the same role for FRSs as ODBC (Open Database Connectivity) does for relational DBMSs [19].

Joshua is a system for uniform access to heterogeneous knowledge structures developed with the same objectives as GFP [44]. Joshua is organized around a uniformly accessible database, whose interface consists of two *generic functions* `TELL` and `ASK`. In addition, Joshua supports a *protocol of inference*, for example, operations to add forward trigger rules, to insert a justification and to ask for an explanation. It supports default implementations for its generic functions and protocol of inference. The default implementation can be overridden if necessary. Joshua has been used for reusing truth maintenance systems. Joshua and GFP share many features, for example, a knowledge model consisting of frames, slots, and facets. GFP and Joshua have complementary functionality. GFP has a collection of over one hundred methods to access an FRS whereas Joshua supports only `TELL` and `ASK`. GFP, however, does not have any methods to manipulate justifications and rules.

GFP is undergoing further development under the auspices of a working group formed under DARPA's High Performance Knowledge Bases (HPKB) program where it has been renamed Open Knowledge Base Connectivity [6]. A more detailed description of GFP implementation and examples of using it are also available [42, 43]. Major extensions planned include better support for operations to deal with general logical expressions and support for querying a KB for an explanation of an answer returned by it.

5. Generic Knowledge Base Editor

The knowledge representation community has long recognized the need for graphical knowledge-base browsing and editing tools to facilitate the development of complex knowledge bases. However, the past approach of developing KB editors that were tightly wedded to a single FRS is impractical [29, 32]. The substantial efforts required to create such tools are lost if the associated FRS falls into disuse. Since most FRSs share a common core functionality, a more cost-effective approach is to amortize the cost of developing a single FRS interface tool across a number of FRSs. Another benefit of this approach is that it allows a user to access KBs created using a variety of FRSs through a single graphical user interface (GUI), thus simplifying the task of interacting with a new FRS.

The GKB-EDITOR is a generic editor and browser for KBs and ontologies — generic in the sense that it is easily portable across several FRSs. This generality is possible because the GKB-EDITOR performs all KB access operations through GFP. To adapt the GKB-EDITOR to a new FRS, we need only create a GFP implementation for that FRS — a task that is considerably simpler than implementing a complete KB editor. For the GFP back ends that we have created at SRI, the average effort required has been of the order of four to six person weeks, whereas we have invested more than one person year in the development of GKB-EDITOR. The GKB-EDITOR and the Stanford Ontology Editor [16] have been the most significant applications driving the development of the GFP. They have driven the addition of new operations to GFP, and they have challenged the portability of GFP because the GKB-EDITOR has been used in conjunction with several FRSs.

The GKB-EDITOR contains a number of relatively advanced features, such as incremental browsing of large graphs, KB analysis tools, operation over multiple selections, cut-and-paste operations, and both user and KB-specific profiles.

5.1. Viewing and Editing Knowledge Bases

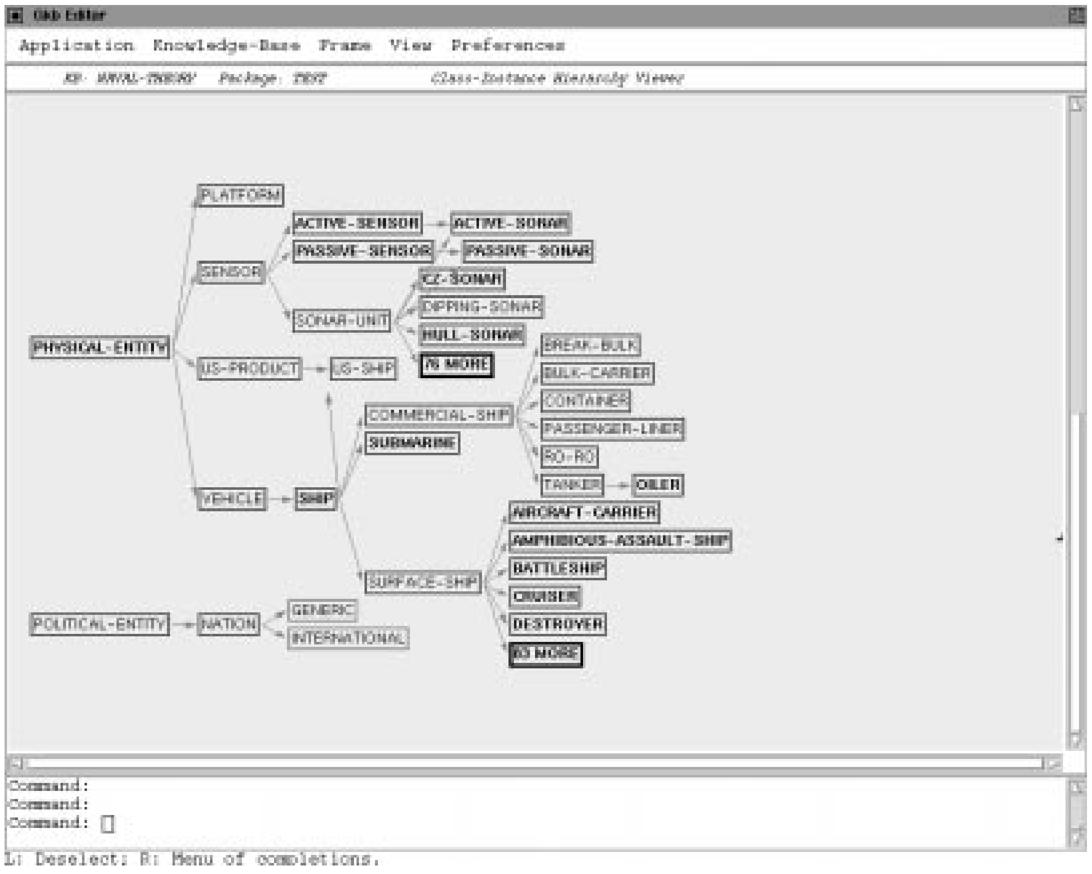
The GKB-EDITOR offers four different ways to view and edit parts of a KB. The user can view the KB as a class-instance hierarchy graph, as a set of inter-frame relationships (this view is roughly analogous to a conceptual graph representation, a semantic network, or an entity-relationship diagram), by examining the slot values and facets of an individual frame, or by viewing a collection of slot values within a spreadsheet. Each viewer is suited for different viewing and editing tasks.

5.1.1. Class-Instance Hierarchy Viewer A sample screen shot of the the class–instance hierarchy viewer is shown in Figure 2. Each node in the graph represents a single class or instance frame, and directed edges are drawn from a class to its subclasses and from a class to its instances. Multiple parentage is handled properly.

The hierarchy is normally browsed incrementally. The roots of the hierarchy graph are either computed or specified by the user, and the graph is expanded to a specified depth. If a particular node has more than a designated number of children, the remaining children are condensed and represented by a single node. Unexpanded nodes are visually distinguished from expanded nodes. The user can browse the hierarchy by clicking on nodes that are to be expanded or compacted.

The hierarchy viewer can also be used to modify the class-instance hierarchy. Operations such as creating, deleting, and renaming frames, and altering superclass-to-subclass links and class-instance links can all be accomplished with a few mouse clicks.

5.1.2. Relationships Viewer It is often useful to visualize other relationships in a KB than the parent–child relationship, such as the part-of relationship. Such

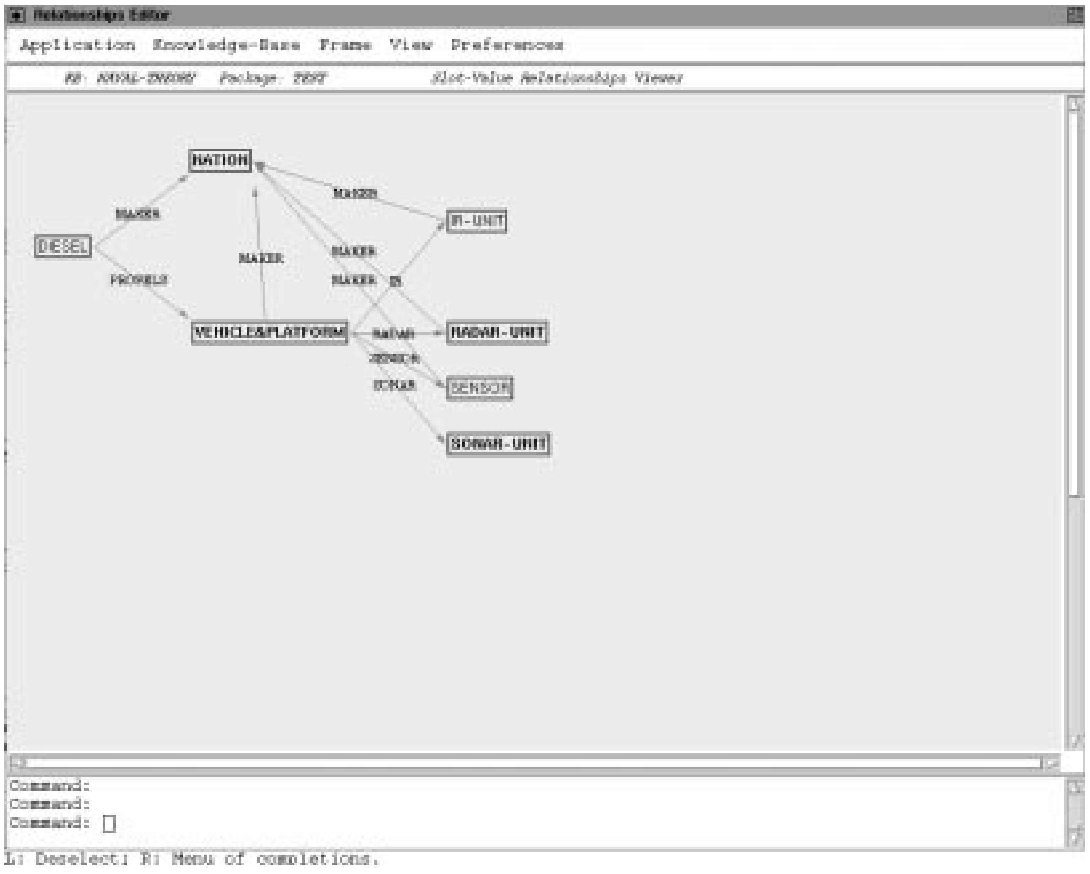


```
Command:
Command:
Command:
L: Deselect; R: Menu of completions.
```

Figure 2. Class–Instance hierarchy viewer in GKB-EDITOR showing a KB in Naval domain developed by University of Southern California's Information Sciences Institute (ISI/USC). The nodes shown in bold can be expanded further. The box containing "76 More" represents that the node Engine has 76 more children that can be expanded.

relationships are encoded in the KB as slot values. For example, if frame B is a value of slot X in frame A, then an edge can be drawn from node A to node B, labeled X (see Figure 3). If we recognize that slot X represents a relationship between frames A and B, then this view is analogous to the view of a KB as a conceptual graph (although our displays do not use all the visual conventions of the conceptual graph community) or to a semantic network. Like the hierarchy view, a relationships view is browsed incrementally. This viewer supports a variety of relationship editing operations.

5.1.3. Frame Editing Viewer The frame-editing viewer allows the user to view and edit the contents of an individual frame. The user may select a frame from the hierarchy viewer or the relationships viewer and display it in a frame-editing viewer, which presents the contents of a frame as a graph. Each slot name forms the root of a small tree; its children are individual slot values and facet values.



```
Command:
Command:
Command:
L: Deselect; R: Menu of completions.
```

Figure 3. Relationship viewer in GKB-EDITOR. The frame Diesel has two slots: Maker and Propels. The slot Maker is of type Nation.

Inherited items are distinguished visually from local items, and cannot be edited (although they can be overridden where appropriate). With each inherited value, the name of the frame from which it was inherited is shown.

In addition to duplicating, renaming, or deleting the viewed frame, the editing operations available in this viewer permit adding, deleting, replacing, and editing of slot and facet values. Slots themselves may be added, removed, or renamed, when classes are edited.

The semantics editing operations in the GKB-EDITOR are modeled after the GFP operations used to implement them. For example, the deletions are performed as specified by the `delete-frame` operation of GFP. The `delete-frame` operation takes an argument that specifies whether all the subclasses and instances of a class being deleted should also be deleted. When deleting a frame through GKB-EDITOR, a user can specify the value of this argument.

5.2. Related Work on KB Editors

Our work builds on ideas from previous graphical browsers and editors that were built for individual FRSs, including KnEd [13], CODE4 [47], HITS [32], and Protege-II [14]. The principal difference between these tools and GKB-EDITOR are that the preceding tools are restricted to use with a single FRS, and that none of the preceding tools have as full a set of features as does GKB. For example, none have all four of the GKB-EDITOR viewer types, and most do not have incremental browsing of large graphs, KB analysis tools, or highly configurable profiles.

Stanford's Ontology Editor [16] is a browser and editor for ontologies [22]. Currently, the Ontology Editor operates only on ONTOLINGUA ontologies, but because it is implemented using GFP, it could be used to browse and edit KBs for a variety of KR systems. ISI/USC has developed Ontosaurus, a browser for Loom knowledge bases (See `http://www.isi.edu/isd/ontosaurus.html`). Ontosaurus uses WWW as its interface, but is limited to only LOOM KBs. The WWW implementation of these two ontology editors is both an advantage and drawback. The advantage is the easy accessibility of the server; its drawbacks result from the many limitations of the HTTP protocol: most information is presented in textual form, rather than graphically; displays cannot be updated incrementally, as they can in the GKB-EDITOR.

5.3. Uses of the GKB-EDITOR

The GKB-EDITOR is in daily use in several KB and ontology authoring projects using LOOM, SIPE–2 [53], and OCELOT KBs. At SRI International, it is being used in DARPA's High Performance Knowledge Base program for the construction of a large KB of a question answering system [48]. The KB is being constructed by extending and combining several other existing KBs. At Pangea Systems, a collaborative group of five people are using GKB-EDITOR to construct the EcoCyc KB describing *E. coli* metabolic pathways and the *E. coli* genome [25]. EcoCyc contains over 15,000 frames and is available to scientists around the world through the WWW. (See URL `http://ecocyc.PangeaSystems.com/ecocyc/`.) The Gkb-Editor is also being used to construct a medical-informatics ontology of 1200 frames that describes clinical-trials results [46]. Feedback from both user communities has contributed significantly to the development of the Gkb-Editor.

6. Storage System

Most existing FRSs such as Loom and Ontolingua fully load a KB into memory before accessing any part of it. To provide persistence, KBs are written in their entirety to flat files on secondary storage. This approach is not scalable because loading and saving time are proportional to the size of the KB rather than to the volume of information accessed in a given session, or to the number of frames modified in a given session.

To obtain scalability, one can use an indexed file that lets us selectively retrieve frames by some key (for example, frame name). An indexed file, however, does not provide fault tolerance: a long update operation, if terminated abnormally, may corrupt the file. An indexed file also does not control concurrent access by multiple clients. Therefore, our storage system design submerges a commercial DBMS within an FRS. Our storage system, PERK (for **PER**sistent **K**nowledge), retrieves frames incrementally, on demand, from the DBMS. Because fetching of frames on demand from the DBMS, or *demand faulting*, is analogous to page faulting in operating systems, we call this process *frame faulting*. PERK tracks which frames have been modified and transmits those frames back to the DBMS when the KB is saved.

Given the basic architecture, other choices must be made: How should the FRS information be organized in the DBMS? In what order should the frames be loaded into the memory of an FRS client? How can we utilize idle periods to speed up the loading time?

6.1. Design of a DBMS Schema

To store a KB in a DBMS, we must define a mapping from the knowledge model of an FRS to the data model of a DBMS. The mapping can be either KB-specific or generic. In a KB-specific mapping, the database schema is different for different KBs [9, 11, 51, 35]. For example, when mapping from a KB to an object-oriented DBMS, KB classes may be mapped to classes in the DBMS. A similar mapping can be designed to tables in a relational DBMS. In a generic mapping, we define one schema that can hold any frame-based KB. One way to construct such a schema is to store an entire frame as a single uninterpreted byte-string in the database.

The main disadvantage of the KB-specific mapping is that neither the relational nor the object-oriented (OO) data models can capture the rich semantics of the knowledge model used by FRSs [4]. For example, the relational model does not allow inheritance of properties from one class to another. The OO data model solves the problem of inheritance, but it does not support the full power of FRSs such as runtime inheritance of default values or production rules attached to slot values or (in many cases) dynamic changes to the schema. In the generic-mapping

Frames			
KB ID	Frame Name	Frame Bodies	Type
1	Physical Entity	...	class
1	Nation	...	class
1	Diesel	...	class
1	Norway	...	instance

Relations		
KB ID	Relation Name	Body
1	Maker	...
1	Propulsion	...

KB Mapping	
KB Name	KB ID
Naval Theory	1
Aircraft	2

Supers		
KB ID	Class Name	Super
1	Ship	Vehicle
1	Vehicle	Physical-Entity

Instance Classes		
KB ID	Instance Name	Class Name
1	Norway	Nation
1	Canada	Nation

Figure 4. Relational schema used to store LOOM KBs in an DBMS, with sample data

approach, we perform all the processing of frames within the FRS so that no frame semantics are lost.

The advantage of a KB-specific mapping over the generic mapping is that the DBMS query-processing engine can operate on KB data on the server side. The server can achieve high efficiency because it can make use of indices, and because it can process the query locally, without transmitting data across a network. An example of such an operation is uniformly increasing the value of a slot for all the frames in a KB. Since the DBMS cannot represent the full semantics of data, however, it will not be able to process the full set of FRS operations.

We chose to design a generic relational schema that can be extended in limited ways to different KBs. That schema consists of five core tables and four tables to support extensibility. An example of the core tables is shown in Figure 4. The *Frames* table contains frame *bodies.* A frame body is a byte-string that provides an FRS with all the information necessary to create the frame. There are occasions when a frame is referenced without its type being known. Therefore, we defined a type field in the Frames table that identifies the frame as a concept or instance. A KB identifier is included in each table, to enable multiple KBs to be stored in the

Indexed Slots	
KB ID	Slot Name
1	Maker
1	Propels
1	. . .
1	. . .

Character Slot Values			
KB ID	Instance Name	Slot Name	Slot Value
1	Akula-Class	Maker	Russia
1	Akula-Class	Propulsion	Nuclear

Figure 5. Tables to support indexed queries on the DBMS server

same Frames table. The *KB Mapping* table associates a KB name with a unique identifier.

One way to store frame bodies in the DBMS is to use the Lisp printed representation of the s-expressions corresponding to a frame body. In practice, we found that the standard Lisp mechanism for reading and writing an s-expression is extremely slow. Therefore, we developed an encoding of s-expressions into strings that is not human-readable but is more efficient to read and write.

The tables *Supers* and *Instances* enable reconstruction of the concept and instance hierarchy. The former lists the superclass–subclass relationships between concepts; the latter stores the relationship between instances and their parent concepts. The tables are indexed on Supers, Classes, and Instances. This information is necessary because for a concept or instance to be defined in an FRS (for example Loom), all parent concepts must already be loaded, or Loom will not be able to classify the new frame. Thus, we must be able to quickly determine the parents of a given frame so that those parents can be retrieved if necessary. These tables can also be used to answer the transitive closure queries on class-subclass relationships without actually faulting in all the frames in the FRS client.

The core tables are adequate if all the queries are to be processed on the client side. In some cases, especially when the answer of a query is a small subset of all possible answers, it may be desirable to process some queries on the server. We can support server-side query processing by restricting queries to only those frame slots whose semantics can be captured by the DBMS schema — such as slots whose values are not inferred using defaults or production rules. We extended the generic schema in Figure 4 in a manner that indexes specific KB slots within the DBMS. Thus, the slots to be indexed can be KB-specific. We defined three additional tables that store slot values, and one table that stores the slot names that should be indexed for a given KB. In Figure 5, we show representative samples for two of the four tables. Each of the three slot value tables contains two columns: one stores a slot name, the second stores a slot value (the three tables are for slot values that are strings, numbers, and long strings, respectively). We separate the types in different tables because retrieval from an index on numbers is much faster than from the one on strings. The long values cannot be indexed, but they should be available so that we can perform a sequential search on them whenever there is a query on a slot with character values.

This approach increases the storage space requirements because slot values are stored as part of the body as well as in the slot value tables. Our scheme, therefore, trades the generality and storage space for the speed of retrieval. PERK uses the frames tables during demand faulting and uses the slot tables for processing queries on the slot values.

6.2. Frame Faulting

When a user begins a session, she first accesses a KB by opening it. During the process of opening a KB, PERK establishes a connection with the DBMS server, and all the root classes and slot definitions (relations) in the KB are faulted into the FRS client. We fault in root classes, because many of them will be faulted in anyway when a frame is requested by an application. We fault in all the relations, because usually a KB contains a small number of them, and faulting them for once simplifies the faulting of class and instance frames.

A frame fault occurs when an application (or the FRS itself) references a frame F that has not been fetched into the FRS client from the DBMS. When faulting a frame into an FRS client, PERK retrieves its body from the DBMS by issuing one or more SQL queries. PERK then calls standard FRS functions to add the frame to the KB.

Among the difficulties we faced in implementing demand faulting for LOOM was that if a frame F refers to some other frame G, LOOM requires that G must exist in the KB. F might refer to G because G is a parent or a child of F, or because G is referenced in a slot of F. If G is a parent of F, and G is not currently in the memory of the FRS client, PERK generates a frame fault for G to allow proper operation of LOOM. But if G is an instance of F, or if G is referenced in a slot of F, we create a place holder (stub frame) for G. G itself will be faulted at a later time if there is some reference to it by the application.

To understand stubs, consider a frame representing a person John, whose father slot has a value Peter. When we load the frame representing John into memory, we do not load the frame representing Peter. Instead, we create an empty frame, known as stub, whose frame name is Peter. As soon as the application requests for any information other than the name of the frame representing Peter, for example for the age of Peter, a frame fault is generated and it is loaded from the DBMS. Using stubs is also known as *pointer swizzling* and is a common implementation technique for object-oriented databases [30].

The classifier code in LOOM is not re-entrant, that is, while we are classifying a concept A into the subsumption hierarchy, we cannot always start classifying another concept B until classification of A is complete. As a result, frame faults generated while classification is in progress cannot be serviced, because if they are, the classifier may find some of the internal data structures in an inconsistent state. Therefore, before we start classifying a concept, we must make sure that all the concepts that will be referenced in the process are already faulted in so that no frame faults are triggered while the classification is in progress.

We found it difficult to predetermine all the frames that will be used while defining a given concept because there is no document describing the classification algorithm of Loom. We addressed the problem by testing our system with several real KBs and identifying common problem cases.

For example, when a concept A is referenced in the definition of concept B, we can notice it while parsing the definition of A, and demand fault A before processing B. Another simple case arises while dealing with the super concepts: before a concept is loaded, we make sure that its direct supers have been loaded. It is easy to determine supers because the supers of a concept can be determined using the Supers table.

A somewhat more involved case arises while classification is in progress. Suppose concept A has two subconcepts B and C. When we load A, we will create stubs for B and C. When there is a demand fault for B, we fetch it from the DBMS. While B is being classified, the classifier considers classifying B under C, causing a demand fault for C. No matter whether we load B or C first, while classifying one of them, the other one will have to be faulted in. In this particular instance, we know that B cannot be classified under C, because otherwise this fact would have been stored in the Supers table in the database, and we would have faulted C before faulting B. Therefore, we modified the classifier to not consider C during the classification process.

Inverse relationships need special treatment because of the circular dependencies that they may generate. For example, consider a frame A whose slot S has value B. If S' is an inverse of S, B will have a slot S' with value A. Suppose we retrieve A when B is not loaded into Loom. If, while loading A, there is a frame fault for a concept C (perhaps C appears in the definition of A) that has B as one of its slot values, we cannot define C because B is not yet defined. We could not go ahead and load B, because one of the slot values for B is A, and any attempt to load B will trigger a cyclical frame fault for A. We addressed the problem by delaying the assertion of the slot values for C until both A and B have been faulted in.

Even though we have empirically tested frame faulting for several KBs, we have no way to guarantee that incremental loading has not changed the behavior of classification or that we have taken into account all possible situations under which a frame fault may be generated during classification. A principled study of classification in conjunction with demand faulting remains a problem open for future research.

6.3. Prefetching

We can decrease the overall latency of PERK by decreasing the number of demand faulting operations it performs. The number of demand faults will decrease if frames that would have been demand faulted are already in memory at the time the demand fault would have occurred. We can achieve that state of affairs by *prefetching* frames from the DBMS before they are demanded by the application, assuming that the cost of prefetching is less than the cost of demand fetching or that the prefetching can be performed during user idle time.

In our current system, once a frame has been brought into the client memory, it is never discarded. Under this assumption, if prefetching occurs only during client idle time, it is likely to improve performance since it will eliminate some future demand-fetch operations. Prefetching, however, does place an additional load on the DBMS server; a large number of clients prefetching from the same server will ultimately decrease overall system performance. Prefetching can also lead to a greater number of page faults on the client.

A prefetching system must decide which frames should be prefetched. It is better to first prefetch those frames that are likely to be demand-faulted in the near future. The principle of locality suggests that the frames most likely to be referenced in the future will be related to those referenced most recently. We consider three types of frame relationships: a frame that fills a slot in a recently fetched frame X, a frame that is a subconcept of X, and a frame that is an instance of X. Since concepts are more likely to be accessed than instances, the subconcepts of X are the first candidates for prefetching. Next, we prefetch the frames that fill some slot of X. We do not prefetch instances of X, however, because the number of instances of a class can be large and the probability of access to a prefetched instance is low. More details on the implementation of prefetching may be found elsewhere [27].

6.4. Implementation

We have implemented PERK in conjunction with the LOOM, THEO, and OCELOT FRSs, which are all implemented in LISP. Most of the PERK code is also written in LISP. We have used the ORACLE DBMS as our persistent store. In our initial work we had experimented with the EXODUS extensible object-oriented DBMS developed at the University of Wisconsin, Madison [18]. We implemented separate prototype storage systems based on EXODUS and on ORACLE, and evaluated them empirically [28, 27]. We found that ORACLE is easier to work with from a practical point of view, because SQL provides a higher level of interaction than does the extensive C++ programming necessary to interact with EXODUS.

The PERK design ensures that the DBMS is transparent to a client application. A client application accesses the FRS using GFP. To store a KB using PERK, the client application creates a KB of a type that is designated for this purpose, for example, `ocelot-oracle-kb`. PERK implements GFP methods for basic KB operations such as creating, opening, and saving KBs. Frame faulting, as described earlier, is triggered by the **coerce-to-frame** operation of GFP because every time an application accesses a frame, **coerce-to-frame** is called. PERK does not allow an application to query the DBMS directly. Thus, integrity constraints are maintained by the FRS.

PERK interacts with ORACLE by using Oracle Call Interface (OCI). The OCI allows a user to embed SQL calls in an application program. The OCI calls appear as foreign function calls in our storage system. We were able to reuse some of the code for OCI calls from the Intelligent Database Interface (IDI) [35]. We had optimized the IDI to reuse the parsed representation of queries and the buffers used to store the query results. The implementation has been thoroughly tested with both Lucid Common Lisp and Allegro Common Lisp environments.

6.5. Empirical Results

The experimental setup for evaluating the storage system consists of test KBs and software used to conduct the experiments. We considered two sets of KBs. The first set consisted of two real KBs that we obtained from the Information Sciences Institute at USC, and which covered the naval and aircraft domains. The KB in the naval domain had 125 classes, 310 instances and 52 slots. The KB in the aircraft domain had 98 classes, 102 instances, and 63 slots. The maximum depth of the class hierarchy for both of these KBs was 7. The second set was randomly generated so that the characteristics of KBs approximated real KBs. All of the KBs in the second set had 100 concepts with just one super each. The concept hierarchy in each KB was the same, regardless of the number of instances. Different test KBs were generated with 1000, 2000, 3000, and 5000 instances. We refer to these KBs as RKB-1000, RKB-2000, RKB-3000, and RKB-5000, respectively. Instances averaged five slots apiece, with an average of two fillers per slot. Half the slots were filled by integers and the other half were filled by symbols.

Experiments were run using Loom 2.1, running on Allegro Common Lisp 4.3. Similar experiments were done for the Theo FRS, and the results are available elsewhere [27]. Both the FRS and the Oracle server were running on the same workstation, a SPARCstation, model Ultra 1 with 64 MB of physical memory running Solaris 5.2. Lisp was restarted before every trial, to avoid caching effects, and a garbage collection was executed immediately before timing. The DBMS server was configured with a large enough buffer so that the KB could be memory resident. All the experiments assume a warm start for the DBMS server. Overall elapsed times were measured using the Lisp **time** function. Each trial was repeated enough times so that the confidence intervals on the mean elapsed time were less than 5% of its length. These experiments measured demand fetching times only; the prefetcher was disabled.

6.5.1. Experiment 1: Interactive Browsing The first experiment considers a scenario where users are interested in interactively browsing and editing a KB or an ontology by using a tool such as the Gkb-Editor. The users in this scenario perform the following operations frequently: (1) opening an existing KB (2) initiating a browse of the class hierarchy, (3) searching for a node and displaying its contents. Therefore, we measured the times to perform these operations for our test KBs.

In the second and third columns of Table 1, we show the time to open a KB by using a flat file system and PERK. The flat file numbers are obviously higher, because the whole KB must be loaded. While opening a KB, PERK loads only the root classes, the slot definitions, and any classes that may be needed in the process. The benefits of PERK can, however, depend on how interconnected the KB is. The Naval theory KB is highly interconnected, and while faulting in the root classes and

Table 1. Interactive browsing time (seconds)

KB	Time to Open a KB		Time to Initiate a Browse	
	Flat file	PERK	Flat file	PERK
Naval Theory KB	5.75	2.78	2.20	2.80
Aircraft KB	3.36	0.86	0.77	2.34
RKB-1000	100.96	1.45	0.876	9.99
RKB-3000	218.40	1.43	1.08	11.35
RKB-5000	355.00	1.44	1.32	22.58

relations, several other concepts are referenced and must be faulted in. As a result, for the Naval Theory KB, the flat file system and PERK differ by only a factor of 2, whereas for RKB-5000, the difference is two orders of magnitude. The time to open a random KB does not change much with its size, because in each case, the number of root classes is the same. The experiment suggests that with PERK, a user does not have to face a long startup time before beginning to work with the KB.

In Table 1, we show the time to start a typical browse that starts from all the root classes in the KB and expands the subclasses and instances up to a maximum depth of 3. Depth is defined as the length of the shortest path from the root to a leaf node. The actual number of frames that are displayed on the screen is different for each KB. We can see that the time required by PERK is significantly higher. That is natural because PERK faults in frames that are necessary to start the browse, whereas for the flat file system all the frames are in memory. The time to initiate a browse while using PERK is still considerably smaller than the time to open a KB while using the flat file system, and the waiting periods for the user are better spread out over a period of time.

The time to search for a frame and display its contents are not reported here, but they follow a characteristic similar to the time required to initiate a browse.

6.5.2. Experiment 2: Batch Queries In the second experiment, we consider a scenario where users are interested in running complex programs over a KB. The access pattern in such a case can be modeled by a query that randomly references a certain fraction of the KB. We did not consider a more detailed access pattern (such as depth-first search) because for the type of result we want to show here, it would not have made any difference. Each reference faults in at least one frame from the DBMS server. In our experiment, we assume that there is no intermediate time between successive frame references. In practice, there will be some time between the requests, and during the intermediate time the prefetcher could have already brought in the frame that is needed next. Therefore, our results present a worst case scenario.

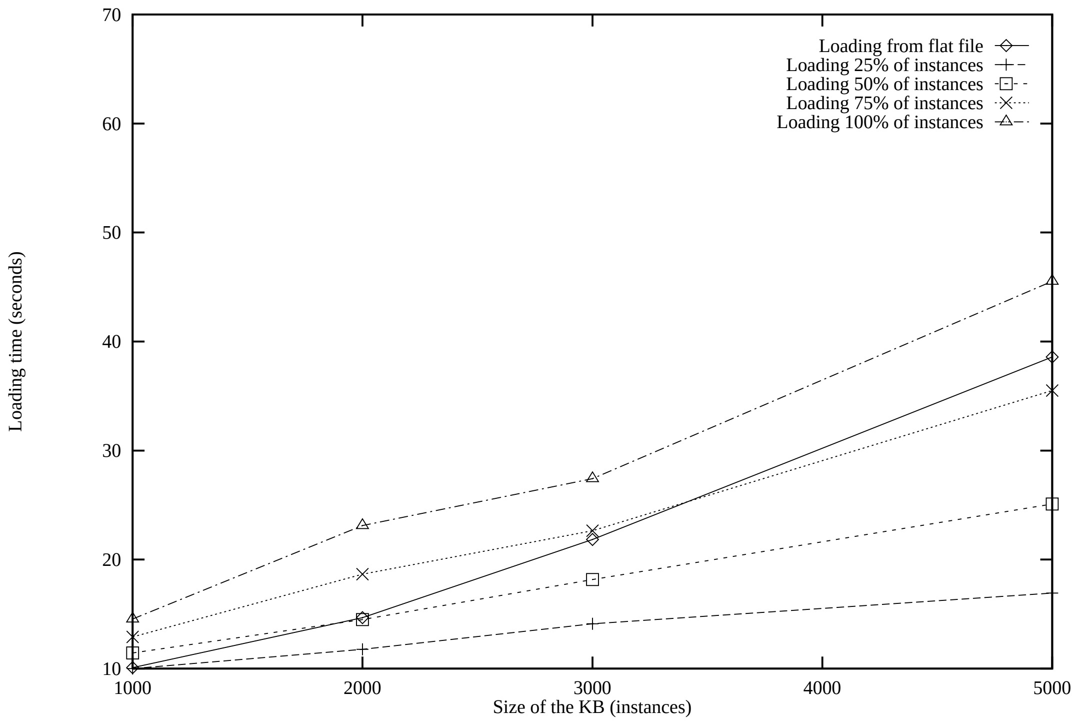


Figure 6. Frame loading time. The solid line shows the time required to load entire KBs of varying sizes from a flat file. The dashed lines show times required to fault in frames from the DBMS due to references to instances by an application. All times refer to total elapsed times. The vertical ordering of dashed lines in each graph and in its legend are the same.

Figure 6 lets us evaluate the relative merits of loading frames from the DBMS versus loading from flat files. We consider only random KBs here because they let us study the effect of varying the KB size on the loading time. Clearly, when a small fraction of the KB is to be referenced in a given session (25%), the DBMS outperforms the flat file. When a larger fraction of a KB is to be referenced (75%), for smaller KBs, it is better to use a flat file, but for larger KBs, loading from the DBMS is faster. For example, for a KB of size 5000, it is faster to reference 75% of the frames from the DBMS than from a flat file.

6.5.3. Experiment 3: Batch Updates We measured the time required to save updates to some number of randomly chosen instances from KBs of various sizes. To be consistent with traditional LOOM behavior, updates are not written as they occur. Rather, we wait until the user issues a command to save updates, and then all are written in a single transaction. We varied the percentage of frames updated

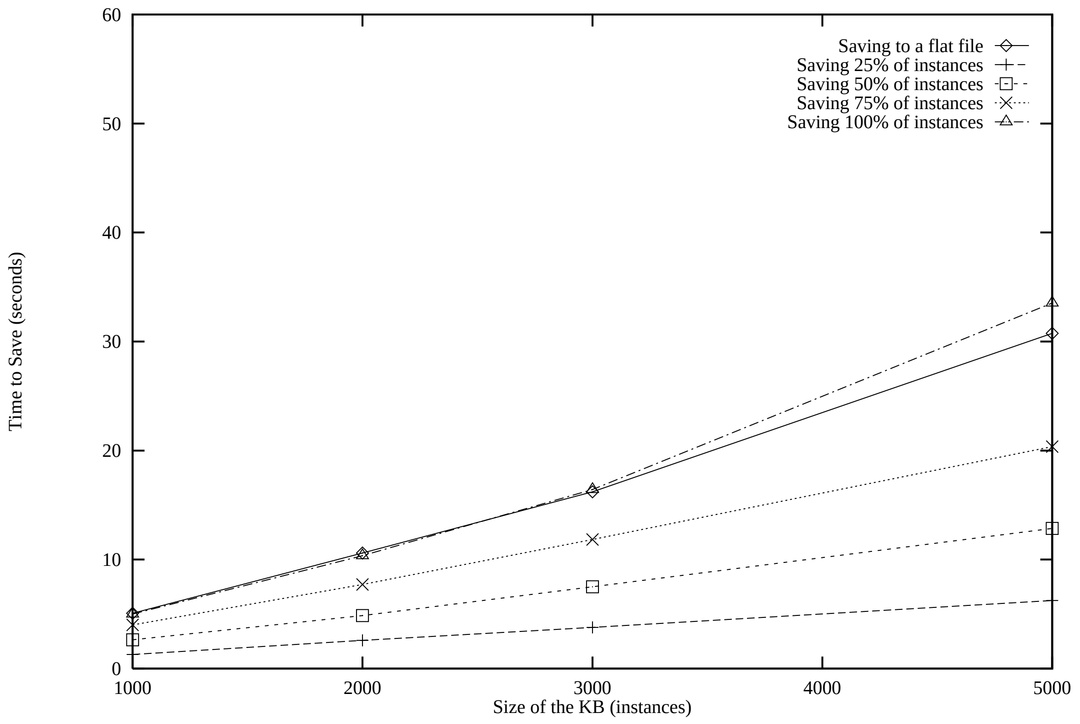


Figure 7. Update times. Each data point represents the time to save a given fraction of instances. The solid line shows the time required to save entire KBs of various sizes to LOOM flat files. The dashed lines show, for KBs of various sizes, the times required to store a given fraction of updated instances to the DBMS. In most cases, saves to the DBMS are faster than saves to the flat file.

between 0 and 100. Selected results are shown in Figure 7. For comparison, we have included the time to save KBs of varying sizes to LOOM flat files (the time is constant for a given KB regardless of the number of frames updated in that KB).

Figure 7 demonstrates that, as expected, our architecture achieves the goal of saving KB changes in time linearly with the number of updates. In most cases, the DBMS outperforms the flat file system for saving the updated instances.

6.6. Limitations of PERK

A limitation of PERK is that once frames have been loaded into virtual memory, there is no way to flush them out. Thus, PERK cannot deal with KBs that lead to a process whose size exceeds virtual memory. We have not encountered this problem with any of the KBs we have worked with so far. Therefore, we believe that the experimental results presented here are of interest to a sufficiently large class of KBs.

The problem of flushing frames from memory is a problem that must be addressed in the long term for scalability of PERK. Flushing frames from memory is not straightforward because frames can refer to other frames by pointers and displacing a frame from memory can invalidate references to it. Special purpose schemes have been developed to solve this problem [30]. We believe that some of the existing schemes can be incorporated into PERK. Some of the schemes require that additional information be maintained while initially loading a KB which can affect the empirical results presented in this paper. Thorough experimentation to study the effect of flushing frames on load time is beyond the scope of this paper and is left for future work.

Our experience in using PERK with LOOM showed that there can be many unexpected interactions between a storage system and the inference capabilities of an FRS making it difficult to design a generic storage system. Therefore, if PERK were to be used with another FRS, a different interactions with that the inference capability of that FRS may arise and would have to be addressed.

The design of PERK takes into account the class–subclass relationships and the slot values of an FRS, but ignores many other aspects, for example, truth maintenance information and deductive rules. Addressing these aspects of FRSs is left open for future work.

6.7. Related Work

The Knowledge Engineering Environment Connection couples the KEE FRS with a relational DBMS [1], and the IDI couples LOOM with a relational DBMS [35]. Both systems employ a KB-specific mapping . The advantage of this architecture is that it allows existing information from a database to be imported into an AI environment. Its drawback is that the storage capabilities of the FRSs are not enhanced transparently, as in our approach. Users of KEEconnection (and of the IDI) must define mappings between class frames and tables in the DBMS; KEEconnection creates frame instances from analogously structured tuples stored in the DBMS, and can store instance frames out to the DBMS. However, only slot values in instance frames can be transferred to the database — class frames cannot be persistently stored using database techniques and cannot be accessed by multiple users. Our approach allows *all* information in a KB to be permanently stored in the DBMS.

Groups at IBM and at MCC have coupled FRSs to object-oriented DBMSs [34, 2]. The IBM effort differs from our approach in that a KB is read from the DBMS in its entirety when it is opened by a K-REP user, which we believe will be unacceptably slow for large KBs.

Markowitz and Chen have developed a system called *OPM* that implements an object-oriented data model on top of a relational DBMS [9]. They have developed automated methods for mapping an OPM object-oriented schema into a relational schema, and for mapping queries in the OPM query language into queries to the underlying relational schema.

Borgida and Brachman have connected the CLASSIC FRS to a database by using a two-phase approach. In the first phase, they load the individual objects and associated facts into an FRS client through a back-door where the usual CLASSIC inference machinery is turned off. Instead, many inferences are performed by the DBMS by asking the appropriate queries. In the second phase, for the inferences that cannot be computed by the database, they use queries to eliminate from consideration those instances that will not add any new information, so that the inference needs to be performed on only a small subset of all the instances. Their approach is geared toward a system that has numerous instances already stored in the database, a complex conceptual schema available in the FRS, with a goal of rapidly loading those individuals into the FRS. Their approach cannot take a complex CLASSIC schema and store it in the database.

7. Collaboration Subsystem

The development of large KBs and shared ontologies requires the collaboration of multiple people who make simultaneous contributions. Most existing FRSs, however, are single-user systems and allow only one person at a time to update a KB, and therefore are inadequate to support collaborative work. It is neither viable to maintain multiple copies of the KB for each of its users, nor feasible to restrict access to the KB to one user at a time. Using an existing commercial DBMS is an unacceptable alternative for the following reasons. Commercial DBMSs control the operations of multiple users by locking the data items that are accessed. The net effect is to isolate the database in a way that each user gets the illusion that she is the sole user of the database. KBs are highly interconnected, and locking one object may require locking several other objects. Transactions in a KB usually access a large number of entities, for example, while performing an inference using backward chaining. If we use locking to control concurrent access, large portions of a KB need to be locked for long periods of time, thus limiting the potential concurrency. The locking model of controlling concurrent access may work well for short online transactions, but in a collaborative environment, it prevents users from working together. Therefore, we need a facility that makes users aware of each other and *helps* them to work together instead of *preventing* them from creating conflicting updates.

7.1. Proposed Solution

To deal with the problem described above, we allow users to make independent changes to the KB; when their changes are complete, we merge their changes with the KB. The user changes merged with the public copy of the KB are those that do not *conflict* with any updates that have been applied to the KB since the begin time of the user's session.

Our solution is similar to an optimistic concurrency control technique in which the users make independent changes to the database, but one (or more) of the users making a conflicting change must abort the changes and restart [3]. In our framework, instead of issuing an abort, we go a step further and assist the users in identifying the conflicts. Our future work will help the user in resolving their conflicting updates. We use locking only while the conflict-free updates are being deposited into the KB.

Our work so far has considered only those operations that change slot values and do not involve any schema changes. We plan to consider schema changes in our future work.

7.1.1. Model of Collaboration We maintain a *public* copy of the KB that is readable by multiple KB developers, but cannot be modified directly. All updates to a KB occur in *private workspaces* (or simply *workspaces*). A developer who wants to modify an existing public KB creates a private workspace as a child of that public KB; by default the workspace contains all of the information present in its parent, but is not necessarily obtained by copying the parent KB. The developer then modifies the workspace by creating new frames, and updating or deleting existing instance frames. Those updates do not affect the public KB, and a private workspace can be accessed by only a single user at a time. A developer who has brought the workspace to a satisfactory state, *merges* the workspace with the newest state of the public KB. The merge operation detects and resolves conflicts (inconsistencies) between the updates made in the private workspace, and updates by other users from which the public KB has been derived.

Figure 8 illustrates these concepts more clearly. The boxes labeled K1–K4 represent successive states of a public KB. The boxes labeled W1–W4 represent private workspaces. Each workspace has a single parent that is a public KB. Each state of a public KB either is the initial state, or is derived from a merge of the previous state of that public KB with a private workspace. For example, K2 is derived from a merge of K1 with W2.

Because K1 is the parent of W2, and is also the public KB with which W2 is merged, no conflicts can occur during that merge. However, when W1 is merged with K2 to create K3, conflicts may occur between the modifications made in W2 and the modifications made in W1.

Each workspace has the full functionality of an FRS. As the developers make updates in a workspace, the integrity constraints are checked by the FRS with respect to the state of the KB in that workspace. While the updates are being made to a workspace, no effort is made to enforce consistency across workspaces. Consistency is enforced at merge time. Merging will require detections and resolution of conflicts (which we term *arbitration*), and creation of a new state of the public KB.

The users always work with the state of the public KB with which they started. For example, in Figure 8, even after KB is in state K2, the user of workspace W1 does not see any changes and continues to work with K1.

In a general solution, it possible that W1 and W2 have different FRSs. Our work so far has not considered this possibility. We assume that all workspaces use the same FRS.

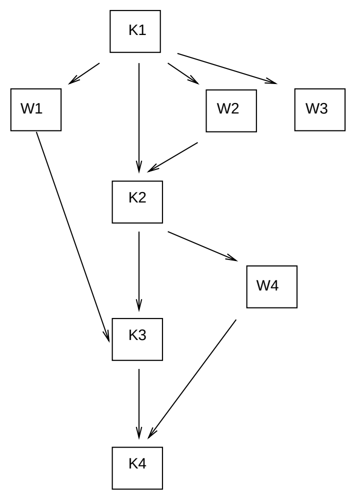


Figure 8. A sample set of relationships among public KBs (K1–K4), and private workspaces (W1–W4)

7.1.2. Recording User Updates We assume a KB that uses the knowledge model of GFP (Section 4.2.1); that is, it consists of classes, instances, slots, facets, and values. The user interacts with the KB by means of GFP operations. A transaction is a sequence of update GFP operations.

With the public copy of the KB, we also record a log, called the *net-log*, of all the changes that have occurred since a given time in the past. When the updates of a transaction are merged with the public copy, conflicts are detected by comparing the operations in the transaction with the operations in the net-log. For detecting conflicts, only those net-log operations need to be considered that occurred since the begin time of the transaction being merged.

A log consists of a sequence of log records. Each log record is a list with two elements. The first element is a list containing the name of the GFP operation and the arguments with which it was invoked. The second element is the list of any values that are being overwritten by the current operation; it has a non-null value only when some value is being overwritten and the old value is not available as part of the GFP operation itself. For a `put-slot-value` operation, we must record the old values of the slot. For a `replace-slot-value` operation, the old value of the slot is one of the arguments of the GFP operation, and we do not need to record it separately. The old values are useful for inverting an update operation when necessary.

7.1.3. Implementation of KB States When the user requests a value from the KB, the FRS must return a value that corresponds to the state in which the user started his or her work. For example, consider a KB state KB_1 that contains a class `person` with a slot `friend` and an instance `John` whose slot `friend` initially contains the value `Peter`. Suppose a user updates the KB to add a new value `Adam` as a friend of `John`. Let the resulting KB states be K_2. If another concurrent user who started when the KB was in state K_1, and queries for the friends of `John`, she should get `Peter` as an answer even if the query is executed when the public KB is in state K_2. There are two approaches to implement such behavior: *delta* and *interval.*

In the *delta* approach, we physically store only one state of the KB, and use the log to compute other states of the KB. The delta approach can be implemented in two ways: positive and negative. In the *positive delta* approach, we store the oldest state and apply the necessary log records to the KB to compute a later state with respect to which the query is to be answered. In the *negative delta* approach, we store the newest state of the KB and negatively apply the updates to answer the queries with respect to the older states of the KB. If we apply the negative delta approach to the example considered in the previous paragraph, we store K_2, negatively apply the entries in the log that are between the KB states K_2 and K_1 (which in this case is an operation adding `Adam` as a friend of `John`) to obtain the desired answer (`Peter` is a friend of `John`). Saving the updates from a workspace is more expensive with the negative delta approach, because in addition to saving the log of changes, we must save the updated frames. On the other hand, purging a log is more expensive for the positive delta approach, because in addition to deleting the log entries, we must save the new state of the updated frames. In general, more users are interested in the recent states of the KB, and therefore the negative delta approach is likely to be more efficient for answering the queries than the positive delta approach.

In the *interval* approach, we associate an interval with each entity in the KB indicating the duration for which it exists in the KB. For example, if the value of a slot salary is 20K in interval t_1 and 25K during interval t_2, the KB contains both of these values. We associate the interval t_1 with the first value and t_2 with the second value. To evaluate any queries on the slot salary, we can use a temporal join index [45]. While the query is evaluated, only the answers that are valid for the duration specified in the query are returned. In the example considered in the previous paragraph, we get only `Peter` as an answer, because that is the only value valid for the KB state K_1.

Over a period of time, the delta approach accumulates a log and the interval approach accumulates past values of entities. Therefore, both approaches require a purging process to remove the information that is no longer necessary. In the negative delta approach, one needs to periodically purge the net-log, and in the interval approach, one needs to periodically purge the old versions of entities. Clearly, the interval approach requires more storage space than the delta approach, because even the entities that have been deleted must be stored. The delta approach has higher runtime cost because the desired answer must be computed by evaluating the relevant log records. If we assume that the purging overhead is comparable for the two approaches, we are faced with the classical computing versus storage tradeoff. A more detailed evaluation of the two approaches is planned for future work. In our current system, we have implemented the negative delta approach.

7.1.4. Definition of Conflict We say that two operations o_1 and o_2 are *conflict-free* if, for any KB state, executing o_1 followed by o_2 leaves the KB in the same final state and returns the same values for each as executing o_2 followed by o_1. For example, inserting two frames with different names is conflict-free. But if a user wants to change a slot value from 2 to 5 and another user wants to change that same value from 2 to 10, then their operations are not conflict-free.

It is useful to consider alternative (and perhaps weaker) notions of freedom from conflict. We say that operations o_1 and o_2 are *partially conflict-free* if, for any KB state, executing o_1 followed by o_2 leaves the KB in the same final state but may not necessarily return the same values for each one of them as executing o_2 followed by o_1. For example, if two users were trying to create a frame with the same name, then their operations leave the KB in the same state but the user who executes the operation first is successful and the operation of the second user is a no-op. We call such operations *partially conflict-free*. Under many situations, it is acceptable to have partially conflict-free operations.

7.1.5. Conflict Detection For conflict detection with respect to a transaction T, we examine the portion of the net-log that contains operations executed after the begin time of T. To check the conflicts between a user transaction and the net-log, we must check for each operation in the transaction, if the net-log contains an operation that performs a conflicting update. Each update GFP operation can, in general, involve multiple updates. For example, the `put-slot-values` operation deletes the old slot value(s) and inserts several new slot values. Furthermore, the number of possible GFP operations is large (over 200), so that analyzing conflicts between all possible combinations of GFP operations would be cumbersome. Therefore, before analyzing the conflicts, we translate the operations into a canonical set of three operations — `INSERT`, `DELETE`, and `REPLACE` — on the nodes and edges of the underlying KB graph (defined below). A `REPLACE` operation modifies an existing KB value. Since the number of operations in the canonical set is considerably smaller than the number of GFP operations, conflict analysis is substantially simplified.

We view the KB as an undirected graph. The nodes represent classes, instances, slots, and values. The edges represent class-subclass, class-instance, class-slot, instance-slot, and slot value relationships. Thus, there are four types of nodes and five types of edges. We use the generic term *entity* to denote either a node or an edge.

In Figure 9, we show a KB KB_1 in a graph form. `Employee` and `Person` are classes and represented as nodes. The subclass relationship between them is represented by an edge. O1 is an instance of `Employee`. `Person` has one slot called `Name`. It is inherited by `Employee`, which has two local slots — `Manager` and `Salary`. The slot values for the instance O1 are shown as nodes with edges from the slot nodes for O1. Since a slot can appear several times in the graph, to identify a slot node uniquely, we must associate it with the frame node it is attached to. For example, the slot `Salary` of the frame O1 could be identified as (O1, `Salary`).

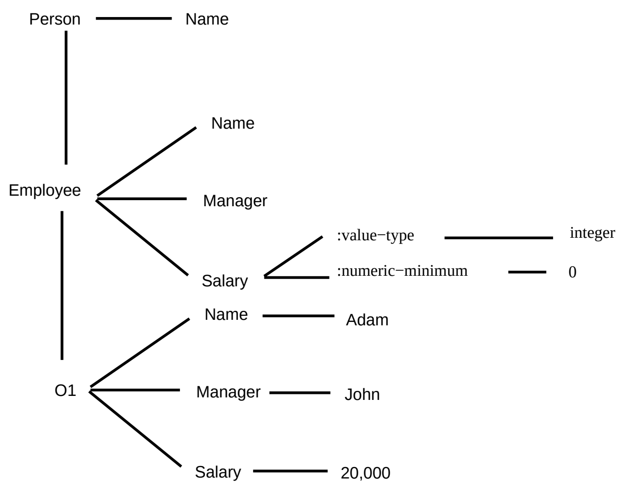


Figure 9. The KB KB_1

A replace operation for edges is not defined. An insert operation on a node is always accompanied by an edge insertion. For example, in the KB KB_1, if for frame O1 we add the value of 20000 for the slot `Salary`, we are inserting a value node 20000 and a slot value edge from the node (O1, `Salary`). Similarly, a node deletion is always accompanied by an edge deletion. To delete Adam's `Salary`, we delete the node associated with the value and at the same time delete the edge between the (O1, `Salary`) slot and the value node. We are interested only in *sensible operations* [8]. Sensible delete (insert) operations are those that delete (insert) an entity only if it exists (does not exist) in the KB. A replace operation is sensible if it is applied to an entity that exists in the KB. The set of all operations is the cross product of the set of entity types and the set of operations.

With the above model, the conflict analysis between GFP operations reduces to conflict analysis between graph operations. When the operations involve distinct nodes or edges, they are trivially conflict-free. Therefore, we consider only the situations in which the operations involve the same entity.

Table 2 shows the conflict matrix for operations such that both of them operate on either a node or an edge. To explain the table, and to keep the discussion concrete, consider the case of slot operations. For two slot operations to conflict, they must refer to the same frame and involve the same slot; that is, if the operation is on a slot value X, its frame name and the slot name must be the same in the two operations under consideration. Consequently, for showing the operations in Table 2, we omit the frame name and the slot name.

When both operations attempt to insert the same value in a slot, they are partially conflict-free, because only one of them succeeds. If one operation inserts a slot value and another deletes it, the operations cannot both be sensible, because for a value to be inserted, it must not exist in the KB, and for it to be deleted it must exist in

P – Partially conflict-free, N – Not conflict-free, "*" – operations not sensible

Table 2. Conflict matrix for node operations or edge operations. For example, Delete(X) and Replace(X,Y) conflict, because the effect of executing them in different order leaves the KB in different states, because only the operation that is executed first succeeds.

	Insert(X)	Delete(X)	Replace(X,Y)	Replace(X,Z)	Replace (Z,X)
Insert(X)	P	*	*	*	N
Delete(X)	*	P	N	N	*
Replace(X,Y)	*	N	P	N	*

the KB, which is a contradiction. Therefore, we do not analyze conflicts for such situations.

The conflicts resulting from operations replacing the slot value X of an entity can lead to three situations: two operations trying to replace X with Y, one operation replacing X to Y and another replacing X with Z, or an operation replacing X to Y and another operation replacing Z to X. When two operations try to replace a value of slot from X with Y, only one of them will succeed, but they will leave the KB in the same state, and therefore they are partially conflict-free. When one of the operations wants to replace X with Y and another X with Z, they conflict, because they have different effects on the KB. Finally, the situation when one operation replaces X with Y and another operation replaces Z with X cannot occur in practice, because for the former operation to happen, X must exist in the KB but for the latter operation, X must not exist, which is a contradiction. However, the operation replacing Z to X conflicts with the operation that inserts X, because executing them in a different order will have different effects on the KB. An operation that deletes X and another operation that replaces X with Y conflict because one of them intends to retain the item in the KB and the other intends to remove it.

Let us now briefly see how Table 2 is applicable for class operations. Two operations inserting a class into the KB are partially conflict-free. Replacing a class name has implications similar to modifying a slot value. For example, two operations replacing a class with different names conflict, but when replacing with the same name they are partially conflict-free. Table 2 also generalizes to edge operations. To specify an edge operation, however, we do need to specify both the end points, and two edge operations conflict only if they involve the same end points. Thus, two Insert(X,Y) operations are partially conflict-free. Since the replace operation is not defined for edges, only the first two columns and first two rows of Table 2 are relevant to edge operations.

7.1.6. Merging User Logs The merge process proceeds in the following steps: log translation, log simplification, log modification, conflict detection, conflict resolution, and log concatenation. Log translation transforms the user log, which is originally represented as a sequence of GFP operations, into a sequence of operations on the underlying KB graph. Log simplification takes a translated log, represented as a sequence of graph operations, and obtains the smallest possible log that has the same effect on the KB as the original log. The simplified log captures the net effect of the changes made by a user in a session. Log simplification is necessary, because while detecting conflicts in a later stage of merging, we do not want to consider any redundant operations. Log modification takes the simplified user log, and modifies it to take into account any changes that have taken place since the time the user started. Conflict detection compares the user log with the net-log and identifies conflicting operations. Conflict resolution resolves the conflicts identified in the conflict-detection phase. Once the user log and net-log are free from any conflicting operations, the user log can be simply concatenated to the net-log within the DBMS.

Log Translation. Typically, each GFP operation translates into a series of operations on the underlying KB graph. As an example of this translation, we consider a few GFP operations. For example, the `put-slot-value` operation translates to the deletion of all the nodes corresponding to old values and insertion of nodes corresponding to new values. As a more involved example, consider the `create-class` operation. The parameters of this operation also include the names of superclasses and slots. If there is only one superclass and no slots, then the operation is equivalent to an insert-node operation, which inserts a node corresponding to the class and an edge between the class and its superclass. In addition, the class inherits all the slots (except for the local slots) from the superclass, which translate into inserting a node and an edge corresponding to each slot. If some of the inherited slots have default values, the translation will contain an insert node operation corresponding to each default value.

Log Simplification. Log simplification uses a collection of simplification rules to obtain a log that is smaller than the original log but has the same effect on the KB as the original log. For example, the insertion of an entity followed by its deletion can be simplified to a null operation. Similarly, if an operation inserts the slot value A, and a following operation replaces A with B, the two operations can be simplified to an operation that inserts the slot value as B. In general, two successive operations on the same slot value can always be simplified to one operation. Therefore, in the simplified log, there is only one operation on each entity.

Log Modification. Log modification changes the user-log to take into account that the KB is not in the same state as when the user started. For example, suppose the initial value of a slot when the user starts is 20. Suppose the net-log contains an operation from another user that adds a second value 30 to this slot. Let the user-log contain a `put-slot-value` operation asserting the value of the slot to be 40, which when executed by the first user, removes the old value of 20 and adds the new value of 40. Since by the time the log is merged, the slot has another value of 30, which should be removed by the `put-slot-value` operation in the user-log, the value 30 must be deleted, and the user should be informed of this additional effect of her operation. For log modification phase to take such effects into account, we must encode in the log some information that can be used during modification. We solved this problem by introducing wild cards in the user log. In the above example, in addition to representing in the log the deletion of slot value 20, we introduce a delete operation in which the slot value is represented as a wild card, meaning that all values of that slot should be deleted. The operation containing the wild card serves as a pattern that is matched against all the insert operations in the net-log. If there is an insert operation on a slot for which all values should be deleted, we generate delete operations for the slot value appearing in the insert operation.

Conflict Detection. Conflict detection principles have already been described in detail in Sections 7.1.4 and 7.1.5. If a conflict is only a partial conflict, it is indicated. Partial conflicts are treated as warnings. It is not mandatory for a user to resolve partial conflicts.

Conflict Resolution. Conflicts detected during the conflict detection step are conveyed to the user. The user is expected to manually resolve the conflicts and resubmit the updates. After the updates are resubmitted, the steps from log translation through conflict detection are repeated. Thus, conflict resolution is an iterative process. The updates are not saved and the log concatenation step not executed until there are no conflicts. Recall that partial conflicts are allowed.

Log Concatenation. Once conflicts have been detected and the user-log modified, it does not contain any operation that conflicts with some operation in the net-log and is ready for concatenation. Log concatenation involves simply appending the user-log to the net-log. At the successful completion of log concatenation, all the user updates are saved to the KB and a new KB state is created.

Over a period of time, the net-log grows in size. To keep its size manageable, we purge the logs that are no longer necessary. The operations of a user-log L_1 can be purged from the net-log if all the users who started their session before L_1 have committed their changes. Alternatively, older log entries can be retained to save the history of KB modifications.

Since the net-log can be large, we store it in the ORACLE DBMS. Our storage system supports the storage and retrieval of the log from the DBMS server. Because the log is stored in a central server, different clients' sessions can retrieve log entries and reconstruct an older state of a KB if necessary.

7.1.7. Correctness The conflict matrix defined in Table 2 should have the property that merging conflict-free updates to the public copy of the KB always leads to *correct* KB states. Correctness is commonly captured by the notion of *serializability* [40].

Serializability can be informally defined as follows. Let a transaction be the set of operations executed by a user. An interleaved execution of a set of transactions is serializable if it is equivalent to some serial execution of the same set of transactions in the sense that it leaves the database in the same state and returns the same answers to each one of its users. The serializability graph of an execution is a graph with a node T_i for for each transaction T_i and an edge (T_i, T_j) if a step of T_i precedes in the execution a conflicting step of T_j. It is well known that an execution is serializable if and only if its serializability graph is acyclic [15].

By induction on the number of transactions in the net-log, we can prove that the merge process described in the previous section produces serializable executions. Let L_i denote the net-log with i transactions. The base case easily follows because L_1 is trivially serializable. For contradiction, suppose L_{m+1} is not serializable. There must be a cycle in the serializability graph of L_{m+1} that includes T_{m+1} and contains an edge from some $T_i, i \leq m$ to T_{m+1}. But that is not possible because the updates of T_{m+1} are conflict-free from the updates of all $T_i, i \leq m$.

We must, however, make sure that the conflict matrix covers all possible cases of conflict between user operations. There are two main classes of conflicts — conflicts between update operations and conflicts between read and update operations. The conflict matrix defined in Table 2 covers all cases of conflicts between update operations and some of the important cases of conflicts between read and update operations.

Traditionally, a read operation on an entity A by a transaction T_1 is assumed to conflict with an update operation on A by T_2. But the two operations conflict only if T_1's read operation influences any values written by it. In practice, it is hard to determine which read operations influence the values written by a transaction. In the KB environment, we have found that transactions read a large number of entities, but most of them do not influence the values written by it. For example, a user may explore a KB using the GKB-EDITOR, but may change the value of only one slot of a frame. In our current work, we assume that a read operation on an entity does not conflict with an update operation on A unless we have reasons to believe otherwise.

We have considered two important situations when a read operation on an entity A conflicts with an update operation on A. The first situation is considered in the conflict matrix defined in Table 2, and arises when a transaction replaces A to B and another transaction replaces A to C. The second situation is considered in log modification when a transaction deletes all the values of a slot, and by the time its updates are merged, a new value of the slot has been added.

In some cases, read operations may trigger inference, whose results may depend on the updates made by a concurrent transaction. We assume that all read/write operations performed by a transaction, direct or triggered, are explicitly represented in the log, which ensures that conflicts caused by any triggered updates are taken into account. Solving this problem in generality can be difficult because the triggered reads and writes depend on the inference method employed in a system.

7.2. Implementation

We have implemented the solution described above. One of the difficulties we had to address in the implementation was that while merging the updates of a transaction T, we need to consider only those operations in the net-log that were applied to it after the begin time of T. It is straightforward to identify such operations if we keep track of the begin timestamp of each transaction. Let us name the subset of the net-log that is of interest to be *sub-net-log*. The *sub-net-log* must be simplified, because we should check conflicts with respect to the net effect of the operations. We used the same procedure for simplifying the sub-net-log as we used for simplifying the user-logs. But while simplifying the sub-net-log, if two (or more) operations are simplified, we must associate the simplified operation with the original operations, because in the event that any conflict is detected, we need to determine who made the conflicting updates.

We performed extensive testing of the implementation for merging, using some sample logs collected for a KB that is under development at SRI. We used about 50 logs generated by 3 users. All the logs were conflict-free, which suggests that most of the time the users will work on disjoint portions of the KB. Log simplification resulted in shorter logs in many cases, confirming its utility in practice. A more detailed evaluation is planned as future work.

7.3. Limitations of the Collaboration Subsystem

Some updates — particularly schema changes — can have wide-spread consequences in a KB and can conflict with many other updates. For example, removing a slot from a class, or changing a constraint on a slot, can easily conflict with changes made to many instances of the class by concurrent users in other workspaces. Future work should address the issue of ensuring that multiple updates that are rejected because of a conflict can be transformed into an acceptable form so that they are not lost.

When conflict-free updates are applied to the public copy of a KB, some new constraint violations may be discovered which were not detected when the update was applied to the workspace. The collaboration system does not take into account such constraint violations. In our future work, such violations will be either identified during the conflict detection phase or by the FRS while the conflict free updates are being committed.

Our approach has not fully addressed the issue of detecting conflicts among facts derived from the inferential machinery of an FRS (semantic conflicts). Conflicts among updates on facts derived from the inverse-slot links are detected using our approach, because these facts are appended to the log maintained by our system. Facts derived using other types of inference are not checked for conflicts.

Other future work will involve providing users with assistance in understanding conflicts, such as identifying which operations within a series of updates are conflicted.

7.4. Related Work

Many approaches have been proposed for supporting a group of designers working on a large project, for example, a software development project or an engineering-design project [38, 24, 10, 31, 50, 23].

Revision Control System (RCS) is a popular system for managing multiple versions in a software project [50]. The software developers can lock a file before changing it, called *check-out*, and deposit their changes when they are done, called *check-in*. As they check-in, their changes are logged, which makes it possible to recreate previous versions of a file and to allow multiple versions of the same file. There are two main differences between our approach and RCS. First, we have a concept of check-in but no concept of check-out. Check-in is a primitive form of merge operation in which there are never any conflicts. Second, RCS manipulates files, and our system manipulates frames, slots, and values.

Concurrent Version System (CVS), a successor of RCS, CVS allows multiple developers to check-out a file and supports them in merging their changes at check-in time. (See `http://www.loria.fr/ molli/cvs/doc/cvs-paper.ps`.) The CVS approach, conceptually similar to ours, is different in that its merge process uses the Unix *diff* utility for comparing text files, whereas, our merge process compares frames, their slot and facet values and their type and superclass relationships.

Hall has developed a model for collaborative work on design databases that is broadly quite similar to what we propose, but differs in a number of specifics [23]. The central difference between Hall's model and our model is that Hall's model relies on the process of change notification to maintain the consistency of each object, whereas our model relies on the merge operation. In Hall's model, inconsistencies can be detected very quickly since change notifications can be sent as users perform updates. In our model, inconsistencies are detected only when an individual user merges his changes into the public KB. A limitation of Hall's model is that two users cannot simultaneously check out the same object for update in different workspaces, which retains some of the disadvantages of locking. Generally speaking, our approaches are complementary, and further knowledge of their relative merits must be gained empirically.

CYC-L [32] provides the most sophisticated knowledge-sharing capabilities of any FRS (but which are nonetheless insufficient for many needs): A user can copy the virtual memory image of a KB stored on a master knowledge server to the user's workstation (a process that is extremely time consuming). When the user changes the KB on his workstation, CYC-L transmits KB updates to the master server. The server maintains its KB in virtual memory, not in persistent storage, and it has no mechanism for controlling concurrent KB updates.

Mays and his colleagues added persistence and sharing capabilities to the K-REP FRS [34]. They developed the notion of relaxing the strict consistency requirements of traditional databases in favor of merging privately developed workspaces (which they call *versions*) that may contain conflicts. Their merge operation is not published, but our private communications with Mays have convinced us that their models of workspaces and of merging are significantly different from ours, and that several approaches to this new paradigm of KB development should be explored.

Another proposal, similar to ours, uses "history merging" as a concurrency control mechanism for collaborative applications [52]. The merging algorithm, called the *import algorithm*, computes the minimal subset of a history in the same way we do log simplification. It then looks for conflict-free subsets of the history and tries to merge them. The authors use the conventional notions of conflict and assume a relational model. In contrast, we have used an object model of the database that is more current.

An extensive body of work on concurrency control in partitioned networks is relevant in our context [12]. A partitioned network arises in a replicated database scenario when a subset of the sites get disconnected from the rest because of communication failure. Such a situation arises more naturally in a mobile computing environment where users are disconnected from a central server for extended periods of time [49]. Since databases naturally contain logs to support recovery, our conflict detection and log merging techniques can be easily adapted for both mobile computing and replicated databases.

8. Summary and Conclusions

Our research results advance the state of the art of knowledge base management systems in several ways. The GFP provides a generic API for KBs that constitutes a solid foundation for knowledge sharing and software reuse. The strength of GFP lies in its simplicity and extensibility. GFP unifies many aspects of FRSs and description-logic systems, for example, class–subclass relationships and constraints that can be represented using facets. Several GFP wrappers are now in active use, and GFP is being used in DARPA's high-performance KB initiative (see `http://www.teknowledge.com:80/HPKB/`). The protocol is undergoing some improvements, for example, we are enhancing it to deal with highly expressive KBs encoded in first-order logic. It is now sufficiently mature that the implementors of FRSs can consider it as a serious candidate for an API for their system.

The GKB-EDITOR is the user interface to the collaborative KB-authoring environment. It provides four different viewer/editor utilities for browsing and modifying large KBs: the taxonomy viewer, relationships viewer, frame viewer, and spreadsheet viewer. Each viewer is optimal for different browsing and editing tasks. The incremental browsing capabilities of the GKB-EDITOR help a user to explore large KBs as a starting point for KB reuse. The GKB-EDITOR itself has been reused across multiple domain KBs and across multiple FRSs because it performs all KB access and update operations via GFP. The GKB-EDITOR provides extensive facilities for customizing the user interface, such as changing the color and shape of nodes and edges, displaying inherited and local attributes in different colors, etc.

Our storage system transparently supports efficient storage and retrieval of KBs in a DBMS to provide improved scalability for KBs. Because the storage system utilizes a KB-independent DBMS schema, it can be used with different KBs in a fashion that is transparent to the user. A frame-based schema provides fast demand faulting of frames, and a slot-based schema allows the DBMS to support fast indexed queries. Our prefetching architecture allows additional performance improvements by exploiting idle times. The storage system is portable to other FRSs, because only a small part of it is dependent on an FRS. We have demonstrated its portability by using it in conjunction with the LOOM, THEO, and OCELOT FRSs.

The collaboration subsystem permits multiple users to cooperatively update a shared KB. In our model, users perform updates in independent workspaces that allow them to maintain long-duration sessions without interference from other concurrent users. The user updates are recorded as a log of GFP operations. At the end of a session, the log is simplified to remove any redundant operations. To enable efficient conflict detection, the operations in the user log are translated to operations on a graph that encodes the structure of the KB. Rules for checking conflicts between graph operations are defined as a matrix. Whenever possible, conflicts are resolved or support is provided for resolving the conflicts. The collaboration system has been tested in the context of an ongoing scientific KB project at SRI.

The next generation of KBs will be collaboratively manufactured by teams of knowledge engineers through assembly of prefabricated knowledge components and through semiautomatic population of the instances in a domain of interest. Knowledge will be manipulated by diverse problem-solving engines even though they were developed in different languages and by different researchers. The KBs will be of unprecedented size and will use a scalable and interoperable development infrastructure. Tremendous speedups in the creation of KBs will significantly enhance our ability to quickly solve complex problems. The results described in this paper provide important technologies for developing the knowledge bases of the twenty-first century.

Acknowledgments

This work was supported by DARPA contract F30602-92-C-0115, by Rome Laboratory contract F30602-96-C-0332, and by grant R29-LM-05413-01A1 from the National Library of Medicine. The contents of this article are solely the responsibility of the authors.

References

1. R. Abarbanel and M. Williams. A Relational Representation for Knowledge Bases. Technical report, IntelliCorp, Mountain View, CA, 1986.
2. Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Won Kim, Charles Petrie, David Russinoff, Donald Steiner, and Darrell Woelk. Coupling an Expert System Shell With and Object-Oriented Database System. *Journal of Object Oriented Programming*, 1(2):12–21, 1988.
3. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Welssley Publishing Company, 1987.
4. Alexender Borgida and Ronald J. Brachman. Loading Data into Description Reasoners. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 217–226, Washington DC, 1993.
5. Alexender Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperine Resnick. CLASSIC: A Structural Data Model for Objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 58–67, Portland, OR, 1989.
6. Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proceedings of the AAAI-98*, Madison, WI, 1998.

7. Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. The Generic Frame Protocol 2.0. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA, 21 July 1997.
8. Vinay K. Chaudhri and Vassos Hadzilacos. Safe Locking Policies for Dynamic Databases. *Journal of Computer and System Sciences*, 1998. To Appear.
9. I. A. Chen and V. M. Markowitz. An Overview of the Object-Protocol Model (OPM) and the OPM Data Management Tools. *Information Systems*, 20(5):393–418, 1995.
10. H.T. Chou and W. Kim. A Unifying Framework for Version Control in a CAD Environment. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 336–344, 1986.
11. G.P. Copeland and S.N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, 1985.
12. Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3):341–370, 1985.
13. E.F. Eilerts. KnEd: An Interface for a Frame-Based Knowledge Representation System. Master's thesis, University of Texas at Austin, 1994.
14. H. Eriksson, A. R. Puerta, J. H. Gennari, T. E. Rothenfluh, S. W. Tu, and M. A. Musen. Custom-Tailored Development Tools for Knowledge-Based Systems. Technical Report KSL-94-67, Stanford University Knowledge Systems Laboratory, Stanford, CA, 1994.
15. K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633, 1976.
16. A. Farquhar, R. Fikes, W. Pratt, and J. Rice. Collaborative Ontology Construction for Information Integration. Technical Report KSL-95-63, Stanford University, Knowledge Systems Laboratory, Stanford, CA, 1995.
17. Michael Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems.* Kluwer Academic Publishers, 1996.
18. M.J. Franklin, M.J. Zwilling, C. K. Tan, M.J. Carey, and David J. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of the ACM SIGMOD 1992 Annual Conference*, pages 165–174, San Diego, CA, June 1992.
19. Kyle Geiger. *Inside ODBC.* Microsoft Press, 1995.
20. Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, Stanford, CA, 1992.
21. Thomas R. Gruber. Ontolingua: A mechanism to support portable ontologies. Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory, 1992. Revision.
22. T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
23. K. Hall. *A Framework for Change Management in a Design Database.* PhD thesis, Stanford University Computer Science Department, Stanford, CA, August 1991.
24. Gail E. Kaiser. A Flexible Transaction Model for Software Engineering. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 560–567, February 1990.
25. P. Karp, M. Riley, S. Paley, A. Pellegrini-Toole, and M. Krummenacker. EcoCyc: Electronic encyclopedia of *E. coli* genes and metabolism. *Nuc. Acids Res.*, 26(1):50–53, 1998.
26. P.D. Karp. The design space of frame knowledge representation systems. Technical Report 520, SRI International, Artificial Intelligence Center, 1992. URL `ftp://www.ai.sri.com/pub/papers/ karp-freview.ps.Z`.
27. P.D. Karp and S.M. Paley. Knowledge Representation in the Large. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, pages 751–758, 1995.
28. P.D. Karp, S.M. Paley, and I. Greenberg. A Storage System for Scalable Knowledge Representation. In N. Adam, editor, *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 97–104, New York, 1994. Association for Computing Machinery. Also available as SRI International Artificial Intelligence Center Technical Report 547.
29. T.P. Kehler and G.D. Clemenson. KEE: The Knowledge Engineering Environment for Industry. *Systems And Software*, 3(1):212–224, January 1984.
30. Alfons Kemper and Donald Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal*, 4(3):519–566, 1995.

31. Henry F. Korth and Gregory D. Speegle. Long-Duration Transactions in Software Design Projects. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 568–574, Los Angeles, CA, February 1990.
32. Douglas B. Lenat and R.V. Guha. *Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project.* Reading, MA, Addison-Wesley Publishing Co., 1989.
33. R. MacGregor. The Evolving Technology of Classification-based Knowledge Representation Systems. In J. Sowa, editor, *Principles of Semantic Networks*, pages 385–400. Morgan Kaufmann Publishers, Los Altos, CA, 1991.
34. E. Mays, S. Lanka, B. Dionne, and R. Weida. A Persistent Store for Large Shared Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):33–41, 1991.
35. D.P. McKay, T.W. Finin, and A. O'Hare. The Intelligent Database Interface: Integrating AI and Database Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 677–684. Morgan Kaufmann Publishers, 1990.
36. T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, E. Etzioni, M. Ringuette, and J.C. Schlimmer. THEO: A Framework for Self-Improving Systems. In *Architectures for Intelligence.* Erlbaum, 1989.
37. Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, Fall 1991.
38. Marian H. Nodine and Stanley B. Zdonik. Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 83–94, Brisbane, Australia, 1990.
39. Suzanne M. Paley, John D. Lowrance, and Peter D. Karp. A Generic Knowledge Base Browser and Editor. In *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence*, 1997.
40. Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.
41. Ramesh Patil, Richard E. Fikes, Peter F. Patel-Schneider, Don Mackay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA Knowledge Sharing Effort: Progress Report. In *The Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 777–788, Boston, MA, 1992.
42. James P. Rice. Writing an OKBC Application. Technical Report KSL-98-15, Knowledge System Laboratory, Stanford, CA, April 1998.
43. James P. Rice and Adam Farquhar. OKBC: A Rich API on the Cheap. Technical Report KSL-98-09, Knowledge System Laboratory, Stanford, CA, February 1998.
44. Steve Rowley, Howard Shrobe, and Robert Cassels. Joshua: Uniform Access to Heterogeneous Knowledge Structures. In *Proceedings of the National Conference on Artificial Intelligence*, pages 48–52, Seattle, WA, 1987.
45. A. Shrufi and T. Topaloglou. Query Processing for Knowledge Bases Using Join Indices. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, Baltimore, November 1995.
46. I. Sim. *Trial Banks: An Informatics Foundation for Evidence-Based Medicine.* PhD thesis, Stanford University, 1997.
47. D. Skuce and T.C. Lethbridge. CODE4: A Unified System for Managing Conceptual Knowledge. *International Journal of Human-Computer Studies*, 1995.
48. Barbara Starr, Vinay K. Chaudhri, Adam Farquhar, and Richard Waldinger. Knowledge Intesive Query Processing. In *Proceedings of the 5th International Workshop Knowledge Representation Meets Databases (KRDB'98)*, Seattle, USA, 1998.
49. Douglas Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, 1995.
50. W. F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.
51. P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 101–109, Kyoto, Japan, 1986.

52. Jürgen Wäsch and Wolfgang Klas. History Merging as a Mechanism for Concurrency Control in Cooperative Environments. In *Proceedings of RIDE'96: Interoperability of Non-Traditional Database Systems*, New Orleans, Louisiana, Feb 26-27 1996.
53. D.E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, November 1990.

Contributing Authors

Dr. Peter D. Karp is a Senior Scientist at Pangea Systems. Dr. Karp received his M.S. and Ph.D. in Computer Science from Stanford University. While at SRI International he developed new techniques for building large, shared knowledge bases. He is the principal architect of three SRI technologies: the PERK system for extending the storage capabilities of FRSs and for providing FRSs with multiuser update capabilities using optimistic concurrency control techniques, the Generic Frame Protocol (GFP) for providing standardized and portable programmatic access to FRSs, and the GKB-Editor for providing graphical browsing and editing services for multiple FRSs. At Pangea Systems, Dr. Karp is applying these technologies to the development of scientific knowledge bases for the human genome project.

Dr. Vinay K. Chaudhri has been a member of SRI's Artificial Intelligence Center since April 1995. At SRI, his research focuses on reusable tools for building large shared knowledge bases. He has contributed to the design and development of a collaboration subsystem, persistent store for knowledge bases, the Generic Frame Protocol, and the GKB-Editor. Dr. Chaudhri received his Ph.D. degree from the University of Toronto, Master's degree from Indian Institute of Technology, Kanpur, and Bachelor's degree from Kurukshetra University. His doctoral dissertation *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation* was one of the first attempts to address the issues of concurrent access for knowledge bases. He devised algorithms that can effectively exploit the semantic structure of a knowledge base to offer more efficient concurrency control than would be otherwise possible. His master's thesis was on interactive relational database design and was published as *Lecture Notes in Computer Science* by Springer Verlag. He worked for one and one-half years for Tata Consultancy Services, New Delhi, where he was involved in the development of large-scale information systems.

Suzanne M. Paley is a Scientific Applications Developer at Pangea Systems. Ms. Paley received her B.S. and M.S. degrees in chemistry from Stanford University, and her M.S. degree in computer science from the University of California, Berkeley. Her Master's research, conducted at SRI International, explored means of extending the storage capabilities of frame representation systems by using a relational database as backing storage, including the design and implementation of a novel prefetching mechanism. Also at SRI, she developed a graphical user interface for EcoCyc (an electronic encyclopedia of E. coli metabolism), concentrating on automated displays of biochemical pathways. Other projects include the GKB-Editor, an application for graphically browsing and editing knowledge bases from arbitrary frame representation systems, and CWEST, a toolkit for making CLIM applications available over the World Wide Web.