A Frame-Based Language in Information Retrieval

Marybeth T. Weaver
Robert K. France
Qi-Fan Chen
Edward A. Fox

TR 88-25

A Frame-Based Language in Information Retrieval[1]

Marybeth T. Weaver
Robert K. France
Qi-Fan Chen
Edward A. Fox

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

With the advent of the information society, many researchers are turning to artificial intelligence techniques to provide effective retrieval over large bodies of textual information. Yet any AI system requires a formalism for encoding its knowledge about the objects of its knowledge, the world, and the intelligence that it is designed to manifest. In the CODER system, the mission of which is to provide an environment for experiments in applying AI to information retrieval, that formalism is provided by a single well defined **factual representation language.**

Designed as a flexible tool for retrieval research, the CODER factual representation language is a hybrid AI language involving a system of strong types for attribute values, a frame system, and a system of Prolog-like relational structures. Inheritance is enforced throughout, and the semantics of type subsumption and object matching formally defined. A collection of type and object managers called the **knowledge administration complex** implements this common language for storing knowledge and communicating it within the system. Of the three types of knowledge structures in the language, the frame facility has proven most useful in the retrieval domain.

The factual representation language is implemented in Prolog as a set of predicates accessible to all system modules. Each level of knowledge representation (elementary primitives, frames, and relations) has a type manager; the frame and relation levels also have object managers. Storage of complete knowledge objects (statements in the factual representation language) is supported by a system of **external knowledge bases.** The paper discusses the frame construct itself, the implementation of the knowledge administration complex and external knowledge bases, and the use of the construct in retrieval research. The paper closes with a discussion of the utility of the language in experiments.

---

1. This material is based in part upon work supported by the National Science Foundation under Grant No's. IST-8418877 and IRI-8703580, by the Virginia Center for Innovative Technology under Grant No's. INF-85-016; and by AT&T equipment contributions.

1. Introduction.

Information retrieval is a branch of computer and information science that is concerned with the use of computers to aid in the location of relevant information items. This is often distinguished from finding facts, data elements, records, or specific answers to questions, as one might do with a question answering or database management system. Typically, users of information retrieval systems are provided with groupings of data that hopefully will help them acquire knowledge appropriate to their needs or desires. While some systems accommodate collections of documents and make selections at the document level, other systems deal with "full-text" by locating patterns or Boolean expressions combining terms (e.g., word parts, words, or phrases). From a user rather than a system perspective, though, an information retrieval system should be a tool that helps people find what they want, at any level of detail desired, regardless of the location, media, structure, subject matter, or size of the collection that is being considered.

However, while many humans are able to effectively aid others with their information needs, it is only in recent years that a variety of researchers have begun to develop detailed specifications and working prototypes of flexible automatic systems that can perform a number of those services [4]. Most readily available retrieval systems are rather restricted in their capabilities, are limited in how much of the information content is actually represented, and only make use of one or two of the variety of retrieval techniques that Belkin and Croft [3] have described in a recent review of the field. Our article, on the other hand, discusses how a specific system, CODER, uses knowledge representation, retrieval, and "intelligent" approaches, in part to better model the behavior of a human intermediary.

We feel that this effort is especially important, since as was predicted in [18], the emergence of CD-ROM based retrieval systems provides an opportunity to integrate powerful processors, large capacity memory devices, and advanced retrieval methods to handle information stores that human intermediaries would have a great deal of difficulty working with directly. In the last several years, optical disc and CD-ROM publishing has certainly grown, but there are still few if any intelligent information systems to provide adequate access [21].

The CODER (Composite Document Expert/extended/effective Retrieval) system is a testbed for determining how useful various artificial intelligence (AI) techniques are for increasing the

effectiveness of information storage and retrieval (ISR) systems. The system has three components, as illustrated in Fig. 1: an analysis subsystem for analyzing and storing document contents, a central spine for manipulation and storage of world and domain knowledge, and a retrieval subsystem for matching user queries to relevant documents. This paper focuses on the formalism, constructs and techniques defined in the spine and used by the analysis and retrieval subsystems to maintain knowledge about the CODER system, its documents and its users.

1.1. Hypotheses.

The CODER system is designed to address several hypotheses regarding the usefulness of AI methods in information storage and retrieval systems. Those hypotheses are based on the belief that the much needed improvement of information retrieval effectiveness requires a new approach to document analysis as well as retrieval. We hypothesize the following:

- *Users can perform more effective retrieval when structured knowledge is employed. The hierarchical organization of documents as well as concepts such as names, dates and addresses represent important and useful structured forms of knowledge.*

- *The knowledge engineering paradigm can be applied to information storage and retrieval systems. Logic programming is useful in implementing this paradigm, and can be effective in building such systems.*

- *A distributed expert-based information system (DEBIS) is the most efficient, and perhaps the only, way to build these systems. The resulting modularity should aid both implementation of and experimentation with the system.*

Knowledge engineering methods are applied in CODER in connection with the representation system described in this paper. Briefly, the representation system supports retrieval as follows:

1) Frame structures are defined by system administrators using programs found in the knowledge administration complex located in the spine.

2) For each document, the analysis subsystem analyzes the content and organization of documents, and stores the results of its analysis in frame objects.

3) The retrieval subsystem can then fill or partially fill temporary frame objects during a retrieval session with information found in or inferred from user queries.

4) Problem description frames built during the retrieval session are matched to the frame objects stored by the analysis subsystem to help narrow the set of

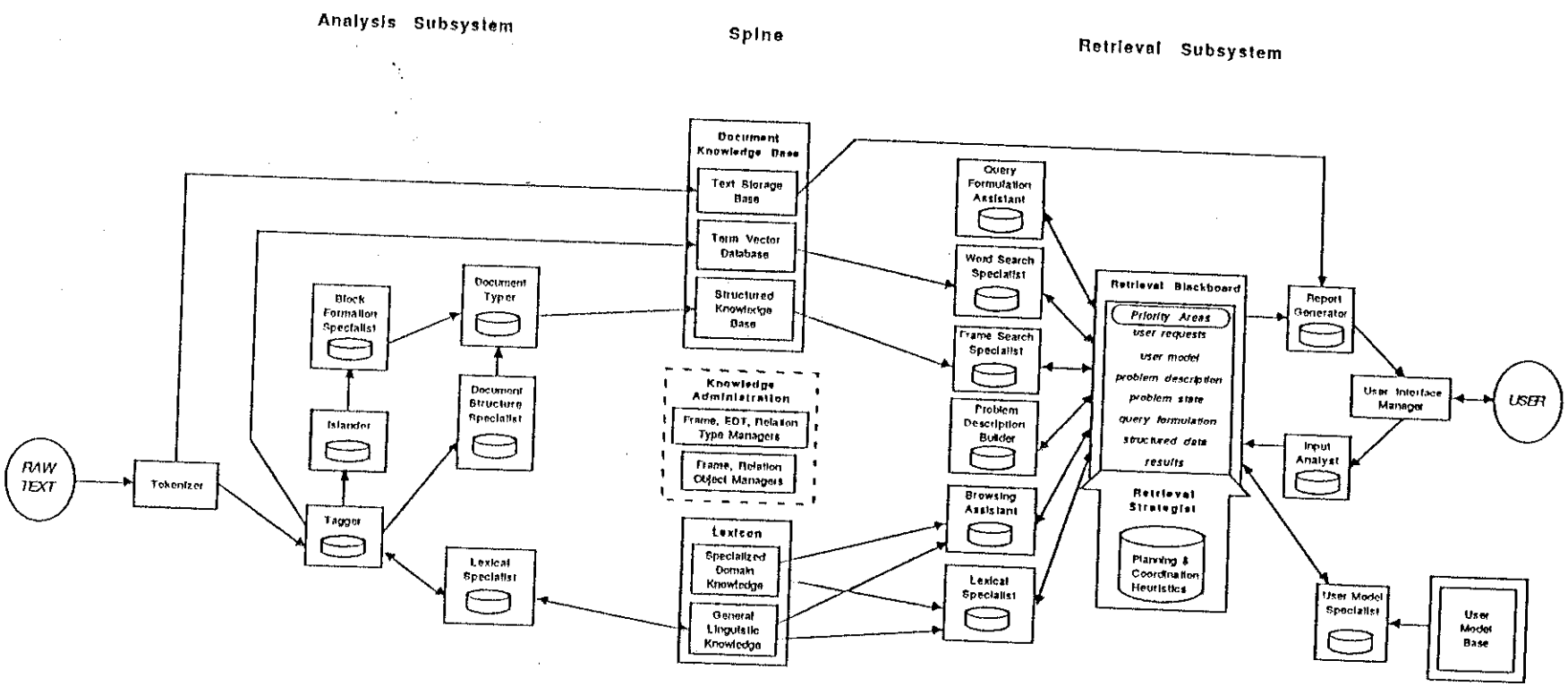


Fig. 1: CODER Version 1.1: system overview.

documents retrieved in response to a given query.

The knowledge engineering paradigm is also employed to store information about users, retrieval sessions and lexical constructs. Examples of user frames and lexical constructs appear in later sections.

1.2. Review of related literature.

In every intelligent ISR system, some formalism and corresponding notation with which to represent knowledge must exist. A variety of different knowledge representation (KR) schemes, for example, logical, semantic network, procedural and frame-based schemes have been studied [11, 41]. Smith and Warner limit their discussion of knowledge representation schemes to those that are used in ISR systems [51]. The CODER system has adopted a frame-based KR scheme to represent document, user and query knowledge. Frame-based systems are used to model entities where each entity is defined by a set of attributes called slots. A complex lattice structure defining the frame hierarchy may be found in such systems. The standard terminology, methods and goals of generic frame-based systems [17, 29, 40] also apply to the CODER frame system.

During the past decade, a variety of frame-based languages have been developed. Such languages have included KANDOR [43, 44], KRL (Knowledge Representation Language) [42], KL-ONE [12], and the languages in the CYC [36], KEE (Knowledge Engineering Environment) [42], and TOPIC [27] systems. Several of the frame language developers have worked on more than one of the aforementioned languages, and commonalities among the languages are apparent.

Many of the concepts in the CODER knowledge administration complex have roots in the Krypton, KRL and KL-ONE representation schemes. For example, Krypton's separation of two representation languages, a terminological one called Tbox and an assertional one termed Abox [9, 45] are paralleled by CODER's segregation of Type Managers and Object Managers for processing frame definitions and instantiations respectively. The taxonomic structure of frames and strict inheritance found in KL-ONE and Krypton also exist in CODER.

Most of the frame representation languages have been developed using some version of LISP, although frame-based languages written in Prolog, like CODER, are beginning to appear [32, 35]. Rowe [49], in his introductory text on artificial intelligence, covers many key issues regarding the utility, implementation, and operation of Prolog frame systems for abstracting collections of facts. One notable difference among the frame-based languages is their treatment

of default values [42]; CODER's treatment of defaults is discussed in §5.2.

One hypothesis that CODER is being used to test, "Users can perform more effective retrieval when structured knowledge is employed," is also a premise of The Information Lens [39], a prototype intelligent information sharing system.

> A rich set of structured message types (or frames) can form the basis for an intelligent information sharing system. For example, meeting announcements can be structured as templates that include fields for "date," "time," "place," "organizer," and "topic," as well as any additional unstructured information.

Based on studies of information sharing in organizations, the Information Lens group has explored how to use cognitive, social and economic information to filter the information that individual users really want to see. As in the CODER design, the researchers have concluded that a frame inheritance lattice simplifies analysis and retrieval of messages. The automatic analysis of document types in the CODER system has much in common with the manual document description in this study.

TOPIC [27] and ARGON [43, 44] are two knowledge-based text retrieval systems that use frame representation schemes. Like CODER, they have incorporated concepts from earlier frame systems; moreover, the representational structures of the frame languages used carry over into the ISR systems and influence their appearance and functioning. The frame system found in CODER is also evident in the capabilities of the system, such as the user modelling functions and the user entry of structured knowledge into frame slots. The TOPIC system, however, focuses on document analysis, using semantic parsing to map text onto frame representation structures. CODER uses its representation language for both analysis and retrieval.

CODER Version 1 is made up of about two dozen different modules, each of which may reside on a different host computer. Modules communicate by passing messages to a blackboard structure. This structure certainly places the CODER system in the category of distributed expert-based information systems (DEBIS). Many of the other DEBIS systems, such as CANSEARCH and I^{3}R are, like CODER, based on a group of cooperating experts centered around a blackboard/scheduler.

The CANSEARCH system [46], also a Prolog-based document retrieval system, retrieves documents related to cancer therapy from the MEDLINE database. The system acts as a front end but performs no actual searches. Instead, it provides a menu driven interface for narrowing and selecting MeSH query terms which are then formatted into a query to be processed by the host computer at the National Library of Medicine. CANSEARCH achieves its objective of

eliminating the user's need to know query formulation specifics; however, its limited scope allows it to ignore some of the more advanced features of a retrieval system, such as user modelling, natural language processing, term expansion or advanced search strategies.

The distributed I³R system developed by Croft and Thompson [14, 53, 54] most closely resembles the retrieval portion of the CODER system. User models, system state transitions, assignment of uncertainty values to rules and on-line help/explanation exist in both systems. Application of some of the research by Belkin, Brooks, Borgman and Daniels [2, 5, 6, 7, 8, 15] regarding the functions of an intelligent information retrieval system, user modelling, and adapting to different users can be found in both systems. A user model builder, an interface manager, a search strategy module and a browsing expert perform parallel functions in the two systems.

Where CODER uses a Prolog frame representation system for representing documents and domain knowledge, I³R, written in Common Lisp, uses a relational database to represent documents. Knowledge intensive processing is restricted to the retrieval end, where concept frames, created primarily from the system's interaction with the user, contain recognition rules to infer domain knowledge [14].

RUBRIC [55, 56], a commercial system based on production rules and written in Common Lisp, uses a manually built rule base to assist query construction and searching. RUBRIC is like CODER in its object-oriented approach to expert systems, its use of relevance values in the range [0,1] rather than in the set {0,1}, and the availability of on-line help at any point in the retrieval process.

The MICROARRAS system [52], a full text retrieval system under development at the University of North Carolina, focuses on databases distributed over different hosts and well-defined use interfaces to support intelligent dialogues. Correspondingly, the CODER system uses TCP/IP protocols to handle databases and modules which exist on different host computers. A more similar system is the communications protocol used by the Utah Text Retrieval Project [30, 31]. The UTRP make use of a high-level protocol to support modules such as user interfaces and search engines residing on (possibly) different hosts. Like CODER, it boasts "tailorability, extensibility, distributability, portability and instrumentability." Unlike CODER, the UTRP focuses on backend processors and networks rather than on AI methods.

2. CODER.

The CODER system is a community of inferential and procedural modules that function concurrently to achieve intelligent information retrieval. These modules include *domain specialists*, which apply domain knowledge to the many tasks involved in information retrieval, and *external knowledge bases*, that maintain large amounts of factual knowledge describing the system's domain, as well as communication, control, and support modules. The system is perhaps best understood as two separate subsystems sharing a central spine of common resources and knowledge. On one hand, the *analysis subsystem* is responsible for cataloging new documents; on the other, the *retrieval subsystem* is responsible for retrieving documents or portions of documents that satisfy a given user's information need.

The CODER *spine* is made up of the central knowledge bases of the system and a set of type managers that support the knowledge representation structures used throughout the system for representation of facts in the problem world. The knowledge bases include the *lexicon*, holding the system's knowledge about individual words, the *user model base*, holding knowledge about users and classes of users, and the *document database*, holding knowledge about individual documents.

Any ongoing session is moderated by an active blackboard. A *blackboard* is a repository for communication between experts, usually divided into a number of subject posting areas. CODER blackboards are considered active because each is managed by a *strategist*, which carries out the main planning and control operations for the session. The strategist initiates the participation of each specialist in the a system task using a knowledge-based model of the specialist's area of competence.

The analysis subsystem includes a specialized user interface that allows for easy document entry, analysis, and storage. During an analysis session, the system accepts input documents of various types, arranges for storage of the text itself, and constructs a set of interpretations describing document structure and content. Each interpretation is represented by a set of facts (ground instances of logical propositions) that can be stored in the document knowledge base.

The retrieval subsystem uses these facts, along with knowledge about words stored in the lexicon and a specialized fact base of knowledge about users, to match documents or portions of documents to a user's information need. The user interface for the retrieval subsystem is designed to be adaptable to different styles of query presentation, including Boolean, vector, or p-norm queries and natural language descriptions of information needs. User behavior is

monitored by a problem description specialist, and the resulting feedback can be applied to sharpen the retrieval. The entire session is coordinated by a strategist whose local knowledge base contains a model of the search process relating user models and search approaches to stages of query refinement.

Communication among modules is restricted to a message primitive called ***ask***, modeled on Prolog call. The UNIX™ *socket* construct provides a means by which even modules that service many clients (on one or many machines) can appear to be serving a single input stream. The entire CODER system, thus, is a set of concurrent modules, executing on one or several machines, and connected together using the socket construct and the TCP/IP protocol. Some of these modules are procedural, some rule-based, and some coupled tightly to large databases, but all use a common interface paradigm, and all use a unified representation of the knowledge that the system applies to the task of information storage and retrieval.

The language used in communicating among these module is called the CODER factual representation language (FRL). Besides providing a *lingua franca* for communication among the specialists, the FRL serves two important functions. First, it provides a language and a set of provably tractable operations on that language that can be used in storing factual knowledge. As such, it is the language used exclusively by the blackboards and external knowledge bases. Second, the FRL provides an underpinning for the more powerful sorts of inference performed by the specialists: classification, rule-based interpretation, and spreading activation in the current version.

Statements in the FRL are finite, grounded *relations* over *frames* and *elementary data items*, where relations correspond roughly to logical predicates taking relations, frames, and data items as arguments, frames are structured descriptions with named attributes filled by frames and data items, and elementary data items are the usual primitives such as integers, atoms and so forth. A statement in the factual representation language, a *fact*, is always a predication that tells something about an object. The three language constructs make possible a wide variety of representation styles within the context of a single language. Of the three, frames have proven most useful in the information retrieval domain. We argue that information retrieval system users think about contexts rather than words, and that frames let us describe contexts. Whereas it is impossible to test general hypotheses regarding the need for "conceptual information retrieval," our theory is falsifiable and so should lead to experimental findings of interest. The remainder of this paper deals with the definition of frames within CODER, the implementation of the principal modules that manipulate them, and their use in information retrieval.

3. Frames.

3.1. Frame systems.

In the years since Minsky's classic definition [40], the term 'frame' has been used to cover a wide variety of artificial intelligence structures. All of these structures have shared with Minsky's initial vision a feeling of concreteness: a frame differs from, for instance, a property list in that one cannot (usually) add arbitrary data to a frame. On the other hand, frames tend to be less rigid than, for instance, relational tuples: not all slots in a frame, to use the familiar terminology, need always be filled. Beyond this, most frame systems produce objects with a great surface similarity. Frame objects tend to look to the static observer like sets of attribute-value pairs. This surface similarity, however, conceals a horde of differences.

Syntactically, frames are tightly bound clusters of attribute-value pairs, called slots. Each slot has a name and either one or a list of values. Usually, each cluster has some sort of unique identifier, often in the form of a filler to a required slot. This clustering is what distinguishes a frame-based representation system from a network, and what allows frame systems to capture the notion of "context" that we believe is crucial for effective information retrieval. While frame systems can include links, those links are seen as part of the structured object(s) they connect, rather than as first-class entities in their own right. On the other hand, frames are different from the structured objects found in third generation languages such as Pascal and C in that they are typically dynamic: the number and composition of slots in a frame object can change over the object's lifetime in much more radical ways than a Pascal record.

The true power of a frame language, and the main area of divergence among frame systems, comes not in the syntax of the constructs, but in the semantics. At any given moment, an individual frame may appear to be a simple set of slots. Underlying that static object, though, is a frame system that both constrains and enriches what slots, and what slot fillers, are possible. Most frame systems, for instance, provide some facility for *inheritance*, where frames can inherit both slots and slot fillers from other frames designated as their parents. Frame systems also tend to provide both constraints on certain slots (for instance, that the slot can have at most one value, or that the value must be within a certain range) and default values for slot fillers. Finally, many frame systems provide for *attached procedures* – 'demon' processes that can compute, to take a common example, a filler for a given slot when it is accessed if the slot

does not already have a filler.

Inevitably, this semantic power requires a commensurate amount of machine power to implement. Brachman and Levesque, in their ground-breaking study of subsumption in frame systems[2], have shown that a syntactically simple frame system, coupled with inheritance and a perspicuous set of slot constraints, is provably computationally intractable. There are certain questions that can be asked about the objects in that system – reasonable questions such as whether it is possible that two frames describe the same real world object – that can only be answered in polynomial time if P = NP. Brachman and Levesque's result is even more sobering when one realizes that they studied neither defaults nor attached procedures. Both of these features raise semantic and computational difficulties. particularly in the context of inheritance. The interaction of defaults and inheritance has been fruitfully examined – although not resolved – by Fahlman, Touretzky and others [16, 57], and shown to involve both subtle paradoxes and non-trivial processing. Attached procedures also have computational cost. If a reasonably powerful language is used to represent them, establishing equivalence of two frames with attached procedures will be an uncomputable problem.

3.2. Frames in CODER.

In designing the frame component of the CODER FRL, we have chosen to exercise discretion. Starting from the bare syntactic construct of a set of filled slots (Fig. 2), we have added only features that could be implemented cleanly and could run in real time. Procedures for filling slots automatically have been de-tached from the frame system proper and located in those specialists responsible for the construction of new frame objects. A facility is provided for specifying default values for slots, but these values are only inherited by child frame types if there is no conflict among parents (see §5.2). Finally, a strong typing system is defined and enforced throughout the factual representation language.

Strong typing of frames has two major effects. First, it provides consistent and intuitive semantics for inheritance in terms of type subsumption. Second, it makes possible a clean distinction between frame types and frame objects. This distinction between an abstract category of structured objects and the individual objects that manifest it has had significant pragmatic benefits in system engineering and programmer training.

---

2. Brachman and Levesque's study has been published in at least three versions. The original study [10] discusses frames in most depth. Later versions have been expanded to cover the range of knowledge representation formalisms. The most recent version [37] appeared in 1987.

```
frame_type ::=    [ frame_type_name , [ { [ slot_name , slot_filier_type ] }* ] ]

slot_filler_type ::=   frame_type
                     | elementary_data_type
                     | list_of slot_filler_type

frame_object ::=  [ frame_type_name , object_id ,  [ { [ slot_name , slot_filler_type ] }* ] ]

slot_filler_obj ::= frame_type
                     | elementary_data_type
                     | [ { slot_filler_type }* ]
```

Fig. 3: CODER frame syntax. Terminals are shown in outline script; commas between list elements are not mentioned, but should be assumed.

```
subsumes(ancestor_frame, descendant_frame) ⊃
        slot_list(ancestor_frame, anc_list) ∧
        slot_list(descendant_frame, desc_list) ∧
        (x) (x ∈ anc_list ⊃
            (∃y) (y ∈ desc_list ∧ name(x) = name(y) ∧ subsumes(type(x), type(y)) )).
```

Fig. 4: Semantics of frame type subsumption.

```
match(frame1, frame2) ⊃
        slot_list(frame1, list1) ∧
        slot_list(frame2, list2) ∧
        (x) (x ∈ list1 has_value(frame1, x, v) ⊃
            (∃y) (y ∈ list2 ∧ name(x) = name(y) ∧ has_value(frame2, y, w) ∧ match(v, w) )).

match(edt1, edt2) ⊃
        edt1 = edt2.
```

Fig. 5: Semantics of frame matching.

One of the difficulties in engineering a knowledge-based system is the variety of knowledge that must be built into it. CODER, for instance, includes knowledge about documents, about words in the English language, and about the users of the system. But it also includes knowledge about the "real world" that the documents and words reflect: knowledge about bibliographic entities, for example, or about the affiliation of computer network users. It includes knowledge that crosses categories, such as how many times a given word is used in the document collection or has been used in a retrieval request by a particular user. Finally, it includes knowledge about its knowledge: how to tell whether a given mail message is a seminar announcement; where to look for the part of speech of a given word sense. This variety of types of knowledge, much more than the sheer bulk of knowledge required for production-scale operation, can slow implementation. It both increases the complexity of an already complex system and lengthens the learning curve for new programmers, especially those with no previous experience in knowledge engineering.

The frame construct itself is an important tool for cutting through this complexity. Even a programmer new to the fine distinction between representing knowledge about the world and representing the world itself can quickly become comfortable with the notion of a frame as a *description* of an entity or situation in the world. This metaphor can help the new knowledge engineer understand how a single world object can be represented by several (even inconsistent) statements in the representation language. A given document may simultaneously satisfy two descriptions; may be at once an ***email_message*** and a ***seminar***. A given lexeme may be described as both a noun and a verb without inconsistency: each description is only relevant in certain situations, but both are descriptions of the same object. Adding to this basic concept the distinction between a frame type and a frame object plays on the metaphor. A frame type, the argument runs, is a description of a *category* of objects in the world; a frame object, a description of an *individual* object.

Actually, a frame type is a category of descriptions, and a frame object is an individual description. More formally, a frame type can be seen as the ideal generated by a set of consistent frame objects.[3] Thus, the slot list of the ***email_message*** type is the union of all features appropriate to a description of an email message, and the default values are those which

---

3. The CODER frame semantics, as has been mentioned elsewhere [26], is based on the domain theory originated by Scott [50]. The conception of types as ideals within this model is due to MacQueen, Plotkin and Sethi [38]. As it happens, we do not need the full power of the ideal construct to describe frames without procedural attachments, but the construct gives an exceptionally clean semantics to the notion of a least upper bound over a domain of recursively defined objects.

are appropriate across the category. Frame types are established at the level of the entire system, as an administrative decision of the system implementers, and cannot be changed dynamically during system operation. They provide the framework around which the systems knowledge takes form: if you will, the categories of the system's perception.

Frame objects, by contrast, are constantly created, manipulated, and either stored or freed during system operation. A *user* frame is created at the login of each new user, and its slots filled as information about the user becomes available to the system. Document description frames are produced by the analysis subsystem and stored in the document knowledge base, where they are later matched against frames that describe the objects of a user's search. As each frame object is created, it is given a type. The type determines what slots the object may have, what default values may exist, and what type of object may fill each slot. As more information is accumulated about the entity being modelled, more slots can be filled and the frame object can become a more complete description. Slot values can also be changed, and slots can even be removed when it is desired to represent the situation where the system has no knowledge (not even, perhaps, the default assumption) about that feature of the description.

3.3. Frame operations.

Frame objects are generally *partial* descriptions of entities. It is possible, in particular, for nothing to be known about an entity except that it belongs to a certain type – in which case it is represented by the frame with no slots. Frame types, on the other hand, are *exhaustive* catalogs of particular categories. Different relations, therefore, hold among frame types than among frame objects.

Frame types are related by the classical relation of type subsumption (see [26] and Fig. 3). One frame type is said to *subsume* another just in case all of its slots are either included in the stronger frame type's slot list, or are generalizations of slots in the stronger frame's slot list. Specifically, frame type a is said to subsume frame type b if every slot of type a corresponds to a slot of type b with the same name and either the same or a stronger type. The frame type with no slots subsumes all frame types, and two frame types are equivalent if and only if each subsumes the other. New frame types are added to the subsumption hierarchy in one (or a combination) of two ways: either by constraining a slot in an existing frame to a stronger type, or by adding new slots. Special cases of this process include adding a frame identical to its parent and adding a frame with two or more parents. This last case succeeds only if the slot lists of the two parents are consistent: i.e., if any slot that appears on both lists appears with either the same type or two types one of which is subsumed by the other.

All frame types thus fall into a single inheritance hierarchy defined by the relation of frame subsumption and having the frame with no slots as bottom. Frame types inherit the slots of their parents, although they may be restricted as described above. There is no provision made for deleting a parental slot from a child type, as this leads to well-known problems of classification. Nor is there provision within the context of frame type definition for expressing relations among slots or relations other than subsumption among frame types. Instead, this knowledge is maintained in the specialists responsible for frame creation, where it can be used as appropriate in the classification of entities. This allows the system to provide a fast and computationally sound method of managing the frame types themselves, while maintaining metaknowledge about use of the types in an inference environment suitable to its application. In particular, this allows the system to maintain knowledge relating frame types (or entities represented as frame instances) on the basis of their *similarity* to each other, something that is not possible in classical frame-based systems.

The corresponding relation to subsumption for frame objects is *matching* (see Fig. 4). A frame object X is said to match a frame object Y if every filled slot of X matches a filled slot of Y, where elementary objects are said to match if and only if they are equal. (Matching is an asymmetric relation: a less specific description matches a more specific description, but not vice versa.) Note, however, that matching frame objects do not need to be instances of subsuming frame types. In particular, frame objects with no slots filled always match, no matter from where in the subsumption hierarchy their types were drawn. What can be said about the corresponding frame types is that they have a greatest lower bound (GLB: this is assured by the method of construction of the hierarchy) and that the frame objects *match at* the GLB: in other words, that a description based on the GLB type is guaranteed compatible with the descriptions represented by the two frame objects.

In a later section, we will show how the two relations of frame subsumption and frame matching work together to help retrieve knowledge structures that, while semantically related, are not overtly similar. First, though, we will discuss some of the structures that occur naturally within the domain of information retrieval, and how CODER frames are used to model them.

3.4. Summary.

Despite its cautious design, the CODER frame system has certain significant advantages. First, it was designed as an integral part of a broader representation language including a well defined set of elementary data types and a system of relational predicates. On the one hand, this

allows the semantics of frames to be kept simple. There is a clean distinction, for instance, between an *embedded* frame, representing a structured part of a larger description, and a *related* frame, representing a description that is related as a matter of fact to a separate object. On the other hand, frames can be specialized. In CODER, frames represent a certain type of knowledge: structured, descriptive knowledge about a single entity. Attributes of entities are represented by elementary data; relationships among entities by relations.

Having thus limited the definition and purpose of frames, we have been able to deal with them at a high level of abstraction. This is reflected in the treatment of inheritance and defaults, where we have been able to give what we feel is a particularly clear and problem free semantics. It is also evident in the operational specification of the frame system. Uniformity across the various languages, modules, and hardware of CODER is enforced by requiring all frames operations to make use of a single set of functions, detailed in §5 of this paper.

A CODER frame is thus an abstract data type, the operations of which are logically specified in terms of the type hierarchy, the filled slots of frame objects, and the corresponding abstract operations on elementary data types. The most interesting of these operations are subsumption and matching.

Frame subsumption is not greatly used in CODER proper. The subsumption hierarchy represents the systems knowledge about types of (descriptions of) entities, rather than the entities themselves. That knowledge is useful for classification of entities; that is, for determining the best category under which to describe a given entity in a given context. Thus, the problem description builder navigates the ***address*** hierarchy in determining the best description for an address supplied by a user as part of a search request. CODER Version 1, however, does relatively little classification, and the frame type system is most used by the frame object system.

Frame objects, on the other hand, are used throughout the system, as is the operation of frame object matching. Matching has proved useful in retrieval, as it provides a precise, implementable definition of the vague concept of "matching descriptions." We discuss how it is used in frame retrieval in §6, and provide some validation of its usefulness in §7. First, though, we turn to a discussion of the usefulness of frames as a representation tool for the domain.

4. Frames for the Retrieval Domain.

Frames are used in describing objects from all aspects of the system's knowledge environment. Users of the system are modelled by frames that describe their status, background, and level of experience; their preferred type of query and document display, and a history of their previous retrieval sessions. The internal structure of documents is captured in frames with slots for the information natural to their type. For instance, a ***seminar*** frame has slots automatically filled by the analysis portion of CODER for the title of the seminar, the person giving the talk, and where and when the seminar will take place. Much of this information, including descriptions of individuals and times, is also structured as frames. Again, frames are used in the CODER lexicon to describe individual senses of words, while relations describe conceptual links between words. The definition and manipulation of such frames will be discussed in this section.

4.1. User modelling frames.

The information retrieval user base is gradually shifting from a group of trained intermediaries to a mass audience of users having diverse aptitudes, computer skills and needs. Intelligent information retrieval systems should support and adapt to a broad range of users, from novice to sophisticated. If this is to be accomplished without losing effectiveness, the retrieval system must perform many of the functions of a human search intermediary. One of the primary functions performed is that of modelling the user.

The aim of the *user modelling* module of CODER is to identify relevant aspects of the user's short and long term goals, background, experience and knowledge. To accomplish its aim, the module uses the knowledge administration complex to build frames and relations for individual users of the system. From information collected during previous sessions as well as from user-supplied responses to menus and prompts, the user modelling expert builds frames for each user. Slot values in the frames determine the user stereotype to be posted to the blackboard; that type, as well as other information about the user, may assist CODER modules in determining action sequences and modes of interaction with the user.

Following research by Daniels [15], we have defined frames for describing users of the system. Frame types for storage of information about each different session in which a user is engaged have also been defined. Each user is described in seven nested frames. All frames are either slots in the primary *user* frame, that is they are part of the user frame, or they are slots in

other frames which are part of the user frame. The user frame contains slots for user identification, user type, and statistics on frequency of use. Other slots are filled by the following embedded frames:

environment: The user's preferred session environment includes the usual type of query, as well as a preferred document ordering and document quantity. These preferences are inferred by reviewing up to ten of the last user sessions, and weighting the preferred environment selected during the current and previous sessions more heavily.

info: General information includes the user's status, background, and native language.

knowledge: The user's level of experience with computers and information retrieval systems.

loginfo: This frame has slots for cumulative statistics as well as historical data. The history is represented by a list of frames containing averages and totals for each time the user has used the system.

The user modelling frames and their slots are displayed in Figure 5. Figure 6 shows a portion of the frame constructed for one of the authors. The frame was captured by a screen dump during a CODER session while the author was browsing his own description. Being able to examine the system's knowledge about oneself is a feature of the CODER interface.

Each time a user logs on to CODER, the user modelling module accesses the local Prolog database, via the knowledge administration functions, to determine whether the user is known or unknown. If the user is known, existing information in the user and session frames is used to classify the user as novice, average or expert. Information is explicitly sought from unknown users so that new frames can be created. At the end of each retrieval session, user frames are updated and a new history frame for the session is created.

4.2. Lexical frames.

One key application of intelligence to ISR is in so-called *conceptual* retrieval. A conceptual retrieval system would be capable of treating words as linguistic or conceptual entities, rather than as atomic items in a list or vector. CODER has been developed under the hypothesis that such lexical understanding will improve retrieval, perhaps as much as or more than the grammatical understanding of document text. In order to test this hypothesis, it has been necessary to assemble a large base of syntactic and semantic knowledge about words. This

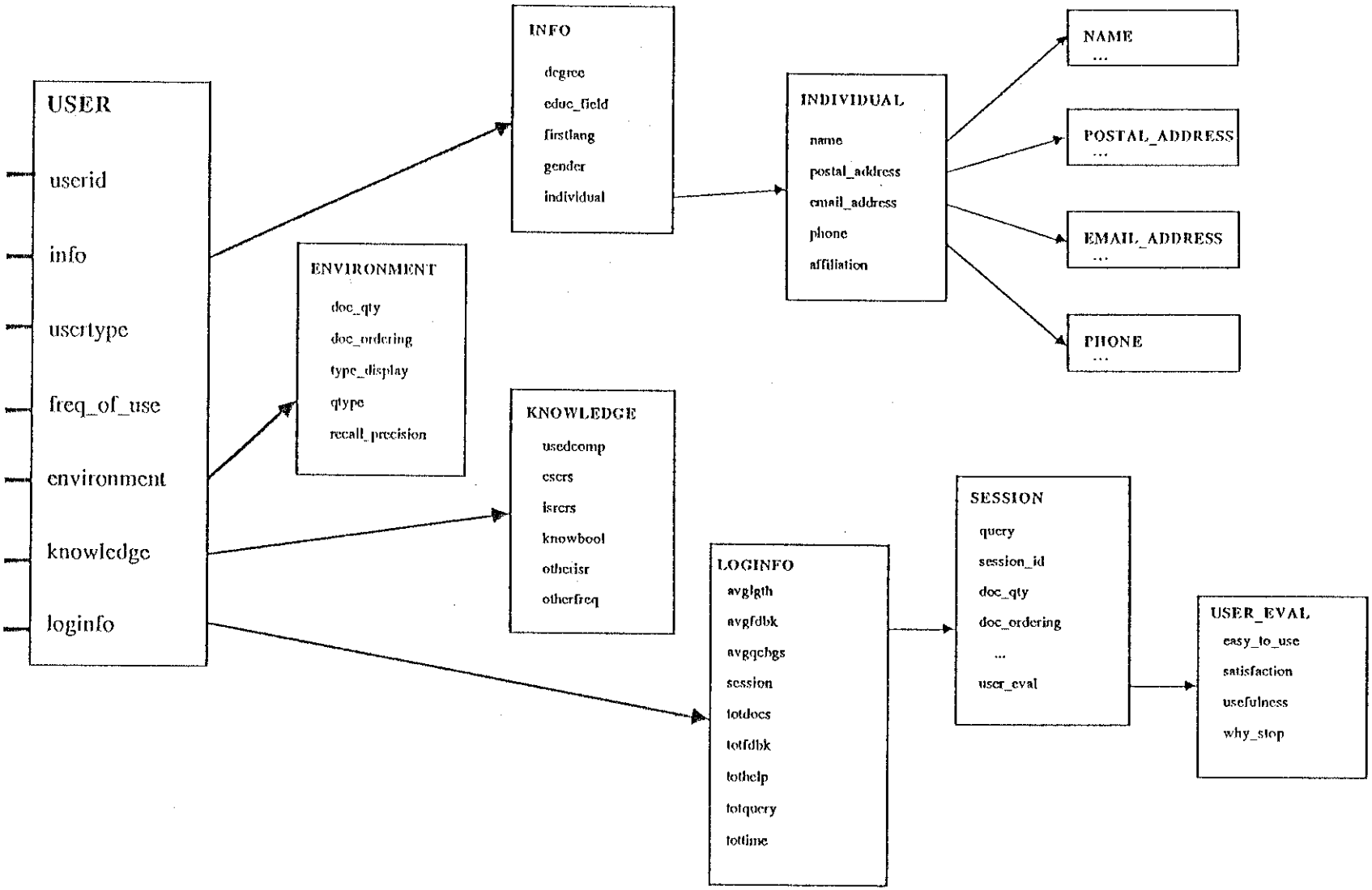


Fig 5: The *user* frame, showing slots and various embedded frames.

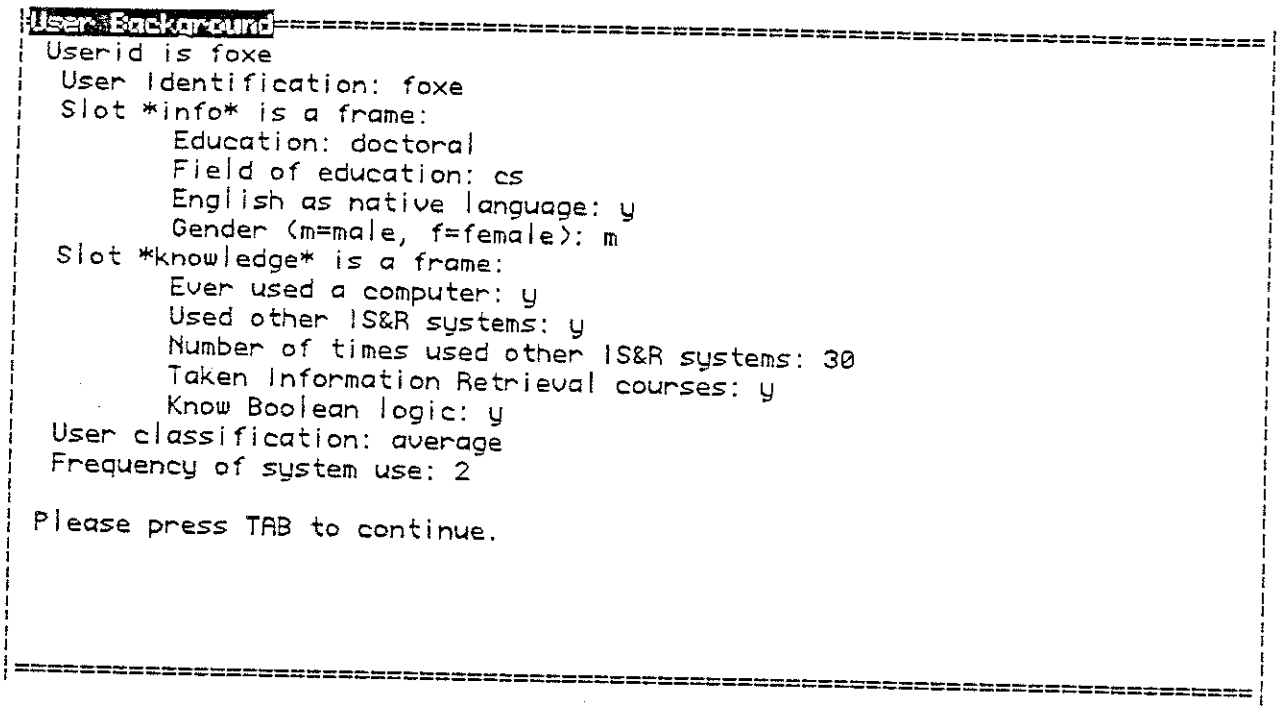


Fig. 6: Browsing one's own decription. A *user* frame is pretty-printed during a CODER retrieval session.

lexical knowledge base, or 'lexicon,' is used by both the analysis and retrieval subsystems. The analysis subsystem currently uses it when recognizing proper nouns; the retrieval subsystem to support browsing during query construction. Both ends of CODER use it for morphological analysis. A related term specialist has been written that draws on the semantic knowledge in the lexicon to expand queries, and a natural language user interface is under development. Despite all this, the possibilities of the lexicon can be said to have hardly been touched.

The lexicon development effort began concurrently with the implementation of CODER. Extensive processing of a typesetters' tape of the *Collins English Dictionary* [28] obtained from the Oxford Text Archive[4], produced several sets of Prolog relations [59]. These relations are currently being refined in an attempt to elaborate the microstructure of the dictionary. In addition, a joint project has been undertaken by VPI&SU and the Illinois Institute of Technology to supplement this information with semantic structures parsed from the definition text of the *CED* and other machine-readable dictionaries [24]. The resulting semantic network will be used both for retrieval and natural language understanding.

A necessary precursor to all these project was a representation scheme that would capture the structure of a dictionary. Dictionaries, like many reference works, are composed of structured entries arranged in what is for all intents a random-access file. A typical entry from the *CED* is shown in Figure 7. Clearly, the structure of the entry is hierarchical, with sub-senses occurring within senses, which are grouped by part of speech, which are themselves grouped into homonyms. Information attaches at each level of this hierarchy. Definitions attach at the sense or sub-sense level; derivational variants attach at the part of speech level; variant spellings, at the homonym level. Semantic registers, style markings, and *compare* references may attach at any level. Information attaching at a high level in the hierarchy, however, is inherited by the levels beneath it.

This structure is captured in CODER by a particular sort of frame called a *term_descriptor*. The frame includes slots for the *lemma* (the distinguished form of the word

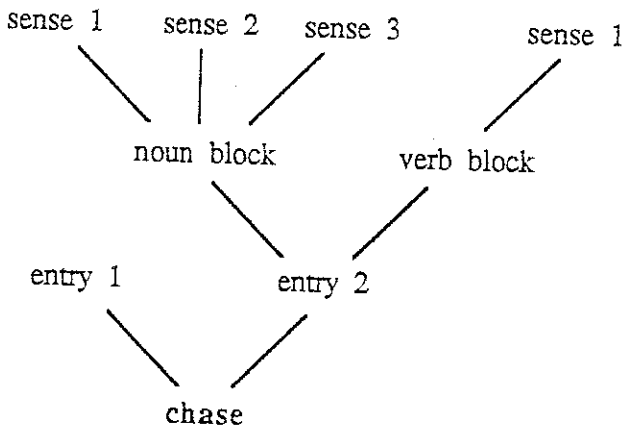
---

4. The Oxford Text Archive, a division of Oxford University Computing Services, is an archive of machine readable works available on tape. The Archive includes works of literature and fiction, as well of reference works of all kinds. Most are in the form of typesetters' tapes, but some have been translated into more perspicuous formats. Both the original typesetters' tape of the *CED* and the Prolog facts produced at Virginia Tech can be purchased from the Archive if the needed permission is obtained. The tapes are distributed for modest fees to the research community. A catalog is available from the Oxford Text Archive, Oxford University Computing Service, 13 Banbury Rd., Oxford U.K. OX2 6NN.

(a)

chase[2] *n.* **1.** *Letterpress printing.* a rectangular steel or cast-iron frame into which metal type and blocks making up a page are locked for printing or plate-making. **2.** the part of a cannon or mounted gun enclosing the bore. **3.** a groove or channel, esp. one that is cut in a wall to take a pipe, cable, etc. ~*vb.* *(tr.)* Also: **chamfer.** to cut a groove, furrow, or flute in (a surface, column, etc.).

(b)



(c)

```
[ term_descriptor [
      [lemma      string],
      [entry      integer],
      [p_o_s      integer],
      [sense      integer],
      [sub_sense  integer] ]]
```

Fig. 7: A dictionary entry. (a) The entry as in appears in the *CED*. (b) The internal structure of the entry. (c) The frame type used to describe that structure.

that heads the entry), the homograph or entry number, the part of speech block number, and so forth (again, see Figure 7). This frame is unusual in that the slots are interdependent: the *sense* slot, for instance, can only be filled if both the *entry* slot and the *p_o_s* slot are also filled. It is also somewhat unusual in its use: it describes a point in a conceptual structure, rather than an entity in the world. While a *user* frame fully describes a user of the system, a *term_descriptor* frame does not fully describe an entry in the dictionary. Rather, it describes a place in the entry where certain attributes may attach.

The actual information contained in a dictionary entry is represented in the lexicon by a set of FRL relations, captured in the sets of Prolog facts mentioned above. Each fact begins with a term descriptor identifying its context and continues with the information that attaches there. For instance, the part of speech of the first block of the second homograph of "chase" is captured by a *c_POS* fact. In order to minimize storage, we make use of the slot dependencies noted above and represent *term_descriptor*s in the Prolog fact bases by an ordered list, where the first component is always the value of the *lemma* slot, the second (if it exists) the value of the *entry* slot, and so forth. Thus the part of speech fact mentioned is stored as:

c_POS(["chase", 2, 1], n).

For the moment, this representation best suits our storage capabilities (see §6.2.3). If it proves desirable in the future to include further information, such as the explicit part of speech, into the descriptor itself, it will only be necessary to define a daughter frame with an additional slots to hold the information desired.

4.3. Document frames.

If documents are not to be treated as flat lists of words, some account must be given of their internal structure. The structure varies among different types of documents: the components and organization of a journal article, for instance, are quite different than those of a reference work. Document structure has been the concern of a number of researchers over the last decade. We have chosen to take as our model the evolving SGML standard [1], supplemented with the work of the Electronic Manuscript Project [34]. Each type of document known to the system is represented by unique frame type, all of which occur in a subsumption hierarchy with the slotless *doc_type* at the bottom. More than one type, however, may be used to describe a document, as a document may answer to more than one description.

As an example, consider the document description frames created by the Version 1 analysis

subsystem. All of the documents in the current collection, drawn from the *AIList Digest*, are examples of a subclass of electronic mail messages characterized by being collected into a digest and re-distributed. As such they are each represented by a frame of type ***digest_message***. The ***digest_message*** type is a subtype of the ***email_message*** type, which is itself a subtype of the base ***doc_type*** (see Figure 8). It contains all the slots needed to describe an electronic mail message, including the sender, the date sent, the subject header, and so forth, but it is a stronger type in also having slots for forwarding information and an additional header, added by the digest moderator.

The analysis subsystem is also capable of recognizing certain "soft types" of document as well, most notably the seminar announcement type. For each of these, it builds a ***seminar*** frame that includes, in knowledge structures comprehensible to the system, all of the information that can be derived from the announcement without fully parsing the text. The ***seminar*** type and the ***digest_message*** type are common descendants of ***doc_type***. But they are incommensurable, as neither type subsumes the other. This is a reflection of their status as conceptual objects. Each frame type represents a description from a different perspective. In the one, a document is viewed in the context of how it was created and distributed; in the other, the context of what it was intended to communicate. By representing these two sorts of descriptions as different frames, the contexts are kept separate. And by representing them *as* frames – that is, as tightly bound, structured objects – attributes that belong to each context are kept together.

CODER frames are thus an apt tool for document description in that they capture well the relationships of subsumption among document types and alternate contexts among document descriptions. They also allow for many of the other relationships that can exist among documents, such as embedding. A cardinal feature of document structure is that one document may be embedded within another, as when a bibliography is included in a journal article. This feature is easily represented using a frame-valued slot. Full recursion, where a slot may have the same type as its parent frame, is also possible, and serves to model the case where an entire document is embedded within another. Still other relationships may be represented by FRL relations.

4.4. World knowledge frames.

As in any sort of text understanding, a certain amount of world knowledge is required in analyzing documents and queries. This is true in CODER even in Version 1, where in depth

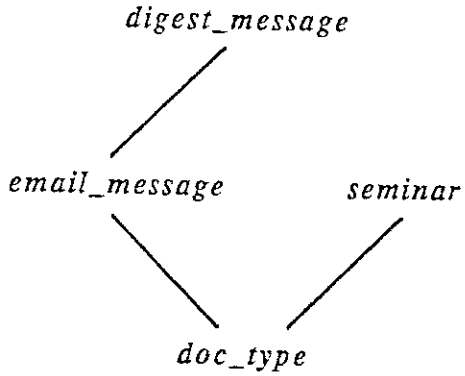


Fig. 8: Subsumption hierarchy of implemented document types.

linguistic parsing is not undertaken. Knowledge of what constitutes a name or address, of what text conventions are used to delineate a title, or of how to decompose an electronic mail address all come into play in the recognition and analysis of "soft" document types. Conversely, analysis of documents results in much factual knowledge about the world, as well as knowledge about the documents themselves. Authors are connected with works and their descriptions, resulting in *bibliographic* knowledge. Individuals are connected with institutions, resulting in a knowledge base of *affiliations*. And explicit dates and times establish *events* in the world, such as the presenting of a paper or the creation of a document. All this information is potentially of use to information retrieval, but only if it can be stored in a formalism that the system can manipulate.

The use of frames in representing structured text items is straightforward. A person's name is represented by a ***name*** frame with slots for first name, last name, title, and so forth. Dates and times are similarly represented. The ***place*** frame type shows by its composition its genesis in describing the location of lectures and seminars: its slots include ***room***, ***floor***, ***building***, and ***hall***. Proper treatment of postal addresses has required an entire subsumption hierarchy of address frame types, including educational and non-educational institution addresses and U.S. and non-U.S. addresses. When these structured text items occur in blocks within the text, though, it is possible to build more complex knowledge frames.

An individual person, for example, has more qualities than his or her name. In recognition of this, the CODER ***individual*** frame has slots for name, physical and electronic mail addresses, phone number and affiliation. A document written by a person allows the analysis subsystem to fill some of these slots; a document about the person allows it to fill others. We hope in Version 2 to build a separate knowledge base of these frames, and relations linking them to publications and institutions. Such a knowledge base would contribute to resolving ambiguous references to individuals. It would also provide an underpinning for merging partial descriptions of individuals into more detailed descriptions. In Version 1, no attempt is made to identify new frames of type ***individual*** with consistent descriptions, and all such frames are stored in the document knowledge base as part of document descriptions. Nonetheless, the information is available in a form that the system can manipulate. Thus it is possible, as described in §6.2.2, to match document frames with a given author's name, even when the name has been given incompletely. In addition, it is possible to display information about all such matching individuals to the user, so that the user can choose among them.

The treatment of time in the FRL is somewhat less satisfactory. On the one hand, with dates and times represented as frames it is possible to use frame matching to find all dates with a

given year, or times with a given hour and minute. On the other hand, the CODER frame system provides no easy way to match a *range* of values, making it awkward to retrieve any date *before* a given date, or all times *between* two points. Section 6.2.2 mentions how we are attempting to overcome this difficulty, which has thus far arisen only in the context of time.

5. The Representation System.

As an AI system, CODER requires an appropriate method for representing knowledge. As an ISR system, it has a specific domain of entities that it must be able to model. These entities include words, names, and other lexical items, documents and their components, and users of the system. The mechanisms for representing these entities and their attributes must include facilities for naming and describing entities, for relating these names and descriptions to each other, and for interpreting them in context. A representation system must also provide inference within the knowledge itself and reasoning about the knowledge. In CODER, all these facilities are drawn together in the *knowledge administration complex.*

The knowledge administration (KA) complex requires a system to support the creation and manipulation of the three levels of knowledge representation in the FRL: simple *elementary data types*, structured *frames*, and propositional *relations*.

Each level of knowledge representation has a type manager; the frame and relation levels also have object managers. The type managers provide the ability to identify, test, and manipulate EDT, frame and relation definitions. The object managers support the creation and manipulation of objects, that is, the instantiation and use of data. The type and object managers are used by the system administrators and by the CODER community of experts respectively. The system administrators are designated knowledge engineers who set out the broad outlines of the domain model. All four of the authors have worked at this task during the implementation of Version 1. We feel that this is about the maximum practical number of administrators. The CODER community of experts uses the knowledge administration predicates to define and represent factual knowledge, and to determine relations, such as subsumption or matching, among both objects and types.

5.1. Organization of the knowledge administration complex.

Type and object files for each of the three types of knowledge, EDTs, frames and relations, are maintained by the knowledge administration complex. Programs for the creation and manipulation of types and objects are also part of the KA complex.

The knowledge administration type manager, **knowadm**, acts as a Prolog *resource manager*. That is, other inferential modules may *ask* the knowadm module for information about EDT, frame and relation types. In addition, the user interface may query knowadm for the natural language information. For instance, when the user constructs frames as part of a query, the interface *ask*s knowadm for names with which to prompt the user for slot data and for a tutorial display if the user requests explanation of the slot. This data is stored in an ancillary file of facts containing a natural language description of each slot defined for a frame type and the prompt and tutorial file associated with the slot. These descriptions must be input by the system administrator when new frame types are defined. Programs also exist to print or display frame definitions and objects.

Although not all of the functions provided by the knowledge administration complex are used in Version 1, representations of entities, classes of entities and the relationships between them can be defined and stored using the KA type and object managers. The type and object programs are the largest of the CODER retrieval subsystem inferential modules. The files containing the type definitions and object data are relatively small and are consulted as local fact bases by this version of CODER. However, incorporation of the NU-Prolog version of MU-Prolog will support the minor modifications required to establish the KA files as external databases supported by special Prolog extensions.

A facility for modifying type definitions does not exist. Due to the complex taxonomies which may be created by EDT and frame type structures and the inherent difficulty associated with such modifications, maintenance to type definitions can be applied by the system administrator only by means of editor software such as **vi**. It is hoped that the structure of frames to represent documents and document fields will be stable enough that lack of a maintenance facility will not stifle the system's capabilities.

5.2. The type managers.

All three KA type managers are contained in a single Prolog program. To simplify entry of type definitions, the program supplies the system administrator with a menu-driven interface. The main menu is shown in Figure 9. For each of the first three options, the system

administrator is prompted to supply the data required to create the Prolog facts which represent type definitions. The information stored for each type definition is described in Appendix A. Option 4, ***Save Updates***, provides a checkpoint facility as a safety measure and is recommended when large numbers of type definitions are entered during one session. It creates a Prolog save state and explains how to re-enter the session at the point where updates were saved. The final option, ***Terminate Processing***, appends the new type definitions to their respective type files.

Please enter function desired:
1. New Elementary Data Type
2. New Frame
3. New Relation
4. Save Updates
99. Terminate Processing

Fig. 9: The knowledge administration complex main menu.

5.2.1. Frame type manager.

The CODER frame type manager supports all functions required for the construction of frame definitions. New frame types are specified either by describing their constituent slots, by specifying parent types from which they inherit features, or by a combination of the two. By specifying each new frame type as a *subclass* of more general frame types, frames may be ordered into taxonomies. Frame types may have more than one parent as well as more than one child, and thus a complex lattice framework may result. Predicates for navigation of frame taxonomies and for identifying subclasses of frame types are included in the frame type manager.

Default values are passed from parent frame slot lists when slots are inherited. However, inherited default values for child frame types may be modified; therefore, an inherited slot may have all of the parent characteristics except the default value. Default values are optional for slots which are EDTs. For a slot which has a frame or relation type, the *identifier* of a frame or

relation object is stored as the slot value; therefore, default values are not permitted for such slots.

Since a single frame type may have more than one parent frame, slots from parent frames are merged according to the ***class*** and ***type*** for the slot. The following heuristics are applied.

- If the slot name, class and type are unique, the slot is inherited.
- Duplicate slot names with the same class and same type are inherited once.
- Duplicate slot names having different classes (EDT, frame, or relation) are not allowed. An error message is provided to the system administrator if this occurs.
- Duplicate slot names with the same class but a different type are inherited only if a subsumption relationship can be established, in which case the stronger frame type is used. If a subsumption relationship cannot be established, then an error message is generated.

5.3. The object managers.

The CODER object managers support the creation and maintenance of frame or relation objects; that is, data is assigned to slots or arguments of instantiations of previously defined frames or relations. To create an object, the frame or relation *type* must be specified so that appropriate slots may be filled or proper relation arguments may be specified. In addition, a unique identifier is assigned to the object being created. Either the object manager or the module requesting creation of a new object may assign the object identifier.

The object manager program has been written so that any module which needs to build or maintain frames or relations may do so in its local Prolog fact base. Then, objects may be stored in appropriate EKBs such as the user model base, or objects such as structured names and addresses in queries may be created locally for use during one retrieval session only. The object manager program issues ***asks*** to the type manager program to validate objects entered as slot values or relation arguments. Unlike the type manager, the object manager does not have its own socket and is not included in the CODER configuration. Rather, modules requiring frame or relation objects *consult* the object manager code. A full description of the frame object manager is included in Appendix A.

6. Knowledge Base Implementation.

Storage and retrieval of frames in CODER occurs in the general context of storage and retrieval of facts. A statement in the CODER FRL is not complete unless it is a relation over either frames and/or elementary data: unless, that is, it is predicating some fact of some description, name, or data value. At the current time, however, no general fact base construct has been implemented. For the purposes of testing and demonstration, temporary knowledge bases have been built for each large pool of frames. These knowledge bases can only handle small numbers of frames (in the thousands rather than the hundreds of thousands) and are generally dependent on the structure of the particular frames stored in them. We will briefly examine each of them in §6.2. Section 6.1 first describes the general-purpose construct, currently in the testing phase.

6.1. The External Knowledge Base.

The external knowledge base is a specialized inferential module. External knowledge bases provide mechanisms for transparent storage, indexing, and retrieval of large numbers of facts about individual entities in the CODER problem universe (whence the nickname "fact bases"). By providing these mechanisms, they shield the remainder of the system from problems of indexing and database support. Individual specialists in the CODER community are freed to maintain only general knowledge in their particular area of expertise in their local rule bases, relying on the various fact bases of the system to hide the "gratuitous complexity" of the problem universe. By the same token, the EKBs are freed from the more complex forms of inference required for if-then rules, quantified statements, or knowledge about knowledge. This distinction is akin to that between necessary and accidental knowledge: the experts maintain derived knowledge about the general nature of the world; the fact bases store knowledge about the particular composition of the world at the moment.

The facts that the external knowledge bases manage are ground instances of the CODER logical relation data type. This data type has been designed to parallel the syntax of propositions in the Prolog language, so CODER facts can be mapped directly to Prolog facts. Specifically, each fact can expressed as a Prolog proposition that includes no variables. The proposition may have other propositions nested within it arbitrarily deeply, but eventually the tree formed by such propositions will terminate in objects of the other two CODER data types: frames and elementary data objects.

Facts are added to an External Knowledge Base as single statements, but may be retrieved

in either of two ways. The Knowledge Base may be queried with a skeletal fact, that is, a fact containing one or more variables, and will return the set of all facts in the Knowledge Base that match the skeleton. Alternatively, the Knowledge Base may be queried with an object (either a term, a frame, or a relation) and will return the set of all facts involving that object. In addition, a Knowledge Base may be queried about the *number* of facts that match an object or a skeletal fact. This information enables the calling module, with minimal communication overhead, to adapt its query to produce a retrieval set of the desired size.

As facts are entered into an EKB, they are broken down according to FRL syntax and stored in three sets of files, one for each level of the language. A ***relations*** file keeps track of which relations (functor/arity pairs) are used in the fact. A ***frames*** file keeps track of any frame types and slot names used. And a set of ***EDT*** files record the ground values that occur within the facts.[5] Structural connections within facts are recorded in a set of ***intermediate*** files.

Retrieval from an EKB uses a combination of three modes, one for each level of the language syntax. Retrieval in EDT mode uses classical B-tree techniques to determine the set of facts that include (at some level of nesting) a particular elementary data value. Frame mode uses the frame type (if specified) and slot names of a skeletal frame to establish a set of facts that may include matching frames. This set is then constrained by recursive calls to the frame mode retrieval module for embedded frames and by EDT mode retrieval for elementary slot values. The candidate matching frames in any facts that remain are then tested for structural equivalence to the incoming frame, using the rules for matching described in §3.3. Relation mode works similarly, except that all three modes are used in the constraining phrase, as both embedded relations and frames may occur as relation arguments, and that the rules for structural equivalence are simpler.

Written entirely in C, the general-purpose EKB is being optimized for quick retrieval on very large collections of facts. The only limitation on the number of facts, frames, or elementary data value that can be stored is the length of a C long integer (2^{31} under VAX Ultrix™). The major limitation on speed is the retrieval time of the B-trees used to store string data, including frame types, relation names, and slot names. Members of the CODER team are currently examining whether perfect hashing [33] can be applied effectively to this problem. A full specification of EKB functionality is given in Appendix B.

---

5. In the current implementation, there is only a single *EDT* file, and all elementary data are treated as strings. This simplification is sound, since the intermodule communication routines guarantee that atoms and integers are always transcribed uniquely before the EKB receives them. Efficiency would clearly be increased by treating integers as integers, however, and we expect to do this in the next version.

6.2. Current implementations.

In order that system testing and development could proceed while the general EKB module was developed, specialized modules were constructed for each reservoir of factual knowledge in the system. This section reviews the three modules constructed and the inference methods used in each. Each module reveals certain features of the utility of frames as descriptions and as contexts. Further, since these temporary modules were written largely in Prolog, each module reveals details of knowledge base implementation in that language.

6.2.1 The user model base.

The user model base (UMB) is unique in the CODER system in that only a single specialist makes use of it. The facts in the UMB are created by the user modelling specialist and used (often in later sessions) by that same specialist. Thus the simplest method for approximating the UMB was to maintain knowledge about users locally in the user modelling specialist.

The knowledge used by the user modelling specialist is fairly sparse. A single *user* frame is created for each user of the system. This frame has several levels of embedded frames (see §4.1), including history for an unlimited number of sessions, but the amount of knowledge per user is still relatively small. More importantly, the pool of users currently using the system is small enough that all such frames can be represented by Prolog clauses in a single local database. As descriptions of system users evolve and change, clauses are asserted and deleted directly into the user specialist's database by the knowledge administration routines. At the end of each retrieval session, this database is saved as a file of Prolog clauses, which is then consulted at the next retrieval session.

For a small set of facts, this method of knowledge storage is optimal. The knowledge administration code is required in the user specialist in any case, as the module must match frames to determine user identity and preferences. Thus the only additional overhead is the actual storage of the frame clauses themselves. Matching for retrieval is provided through the *frames_matching* and *equal_frames* predicates for frames, and through Prolog clause unification for relations. Thus, as long as the internal structure of the facts being sought is known in advance, the more sophisticated matching abilities of the EKBs are not needed. Fortunately, given the structure of the *user* frame, this is true. The decomposition of the *user* frame was therefore coded directly into the user modelling specialist using unification and the knowledge administration predicates as primitives. We estimate that this implementation will prove adequate until the user count exceeds three figures, although consult time becomes noticeable with more than 20 users.

6.2.2. The document knowledge base.

Lacking a single storage module adequate for the quantity of document knowledge produced by the analysis subsystem, CODER Version 1.1 makes use of two specialized components, one inferential and one procedural. The procedural *term vector manager* is a classical document/term ISR system. Using components adapted from the SMART system [13], it matches <document number, term number, weight> tuples produced by the analysis subsystem against either vector, Boolean, or p-norm structures of <term number, weight> tuples produced by the retrieval system. All of these structures are represented by nested lists in CODER proper, although both Boolean and p-norm structures could be better represented by either frame or relational objects. The *document structure base*, on the other hand, collects the document descriptions produced by the analysis subsystem, and as such is based on frame representations.

In contrast to the user model base, the document structure base contains several different sorts of frames. Moreover, these frames can be nested in relatively arbitrary ways, making it difficult to hard-code the structural navigation routines. Finally, it was desired to provide a method of retrieving frames with a slot value in a range of possibilities, for instance in order to retrieve documents within a range of dates. Therefore, rather than relying directly on the knowledge administration code, a different matching algorithm was used.

The frames produced by the analysis subsystem were written out in a file of Prolog clauses using the same knowledge administration routines as in the user model base. They were then translated procedurally into a semantically equivalent, but syntactically different form deemed more effectively retrieved by the Prolog database. Around this database, two nested shells were built. The inner shell retrieves frames whose slots satisfy certain conditions. The outer shell takes a frame description from the retrieval subsystem, breaks it into calls to the inner shell, and translates any frames satisfying the appropriate constraints back into standard FRL for return to the retrieval subsystem.

The inner shell searches for frames satisfying conditions in disjunctive or conjunctive form. Each search function accepts a list of unit constraints, but the former interprets the list as [F_1 or F_2 or F_3 ... or F_n] while the latter interprets it as [F_1 and F_2 and F_3 ... and F_n]. Each unit constraint F_i specifies a set of constraints for a single frame object: either a frame type or a (range of) slot value(s). For example, the constraint

[date, [year, eq, 1987], [month, before, 6]]

restricts the frames retrieved to be date objects with 'year' set as 1987 and 'month' set in the range [1, 5]. The constraints supported by the module are set inclusion (used for finding words or phrases in a list of terms) and the normal mathematical relations (equal to, greater than, less than, and so forth) for integer valued slots. As coded in MU-Prolog, the module uses a simple recursive predicate to break down the query into unit constraints as well as a set of search predicates, one for of each type of unit constraint, to find candidate frames. Since the constraints apply only to elementary data, no nesting of frames is possible.

The outer shell takes frame objects representing the user's description of a desired document, and breaks it into smaller frames whose slots contain only elementary values. It then translates these into calls to the inner shell, combines the results, and chains backward through slot clauses in the local Prolog database to establish which documents include the frames found. Ranges of acceptable values are received as a frame with slots for the two endpoints.

Much improvement can be made on the current version of the module. Most importantly, it should directly support queries about nested frame structures, and make use of the type subsumption when matching frame types. Knowledge should also be added to allow it to better handle temporal queries. The current version can only handle ranges of days within a month or months within a year, for instance. A relatively small number of rules in the outer shell would remove this restriction. All such improvements, however, can be made in the outer shell, allowing the inner shell to remain optimized for fast search.

6.2.3. The lexicon.

There are several difficulties inherent in supporting large knowledge bases directly in Prolog. The principle problem is, of course, that of simply maintaining a database that may consist of hundreds of thousands of clauses. MU-Prolog and its successor, the compilable NU-Prolog, provide an external database facility that uses dynamic hashing and a superimposed codeword scheme to minimize search and update times [47, 48]. Using this facility, fact bases with over 10,000 facts have been successfully stored and consulted by CODER, with each fact represented by a single Prolog clause. Storing and consulting facts, however, is not the only bottleneck in Prolog.

The CODER lexicon is the central storage facility for knowledge about English words and phrases. Its basis is a set of syntactic and semantic relations derived from the *Collins English Dictionary*. As described above, each of these relations is indexed by a frame object describing

the binding point of the relation in the dictionary entry. Attempting to load these facts directly into a Prolog database, however, creates two problems. First, the frames are structures, represented in Prolog as specially formed lists. Hashing schemes tuned to optimal performance when the hash keys are atoms and integers no longer function as well when the key is a structure. MU-Prolog attempts to solve this through its superimposed codeword scheme: the hash keys for structures are created by composing the structure elements. In the case of the *term_descriptor* frames, however, most of the structure elements are small integers, which serve mainly to obscure the significant element, the lemma itself.

Then again, the representation of the lemma causes problems. If lemmata are represented by atoms – certainly the first alternative which presents itself – the Prolog atom table is quickly overloaded. If they are represented by strings, the strings are structures composed of letters drawn from the small set of the alphabet – so the problem of assigning unique hash keys quickly reasserts itself. Instead of taking either of these alternatives, we have chosen to represent each lemma by a unique integer, which is then separated from the rest of the term descriptor for optimum recall (see Figure 10). This makes possible optimal hashing and a minimal database size. To translate from character strings back and forth to these *lexeme numbers*, the lexicon depends on a secondary facility written in C.

The lexicon thus consists of two functional modules, each with a set of data files (Figure 11). The primary module, written in Prolog and using MU-Prolog external databases, keeps track of lexical properties and relationships. It depends on the C module to translate text items into lexeme numbers. Morphological analysis is distributed between the two modules. Regular inflectional transformations and some high-frequency derivations are handled by the C program, which can recognize them efficiently. A large number of irregular inflections and most derivational transformations have been derived from the *CED* and are currently stored in one of the MU-Prolog databases. In the near future, we hope to incorporate this information into the C module.

Once morphological analysis and lexeme-to-number translation takes place, frame matching can be accomplished directly through Prolog unification. This is possible because the lexicon contains only one frame type, and that has all its slots filled with elementary data. Further, the *term_descriptor* frames used in the lexicon have the special property, noted above, that their slots are filled in order. The biggest problems in frame matching, subsumption and recursive matching, thus fail to arise: two term descriptors match if and only if their lexeme numbers are the same and the hierarchy list of the first is a head of the second. No code beyond basic list operations is required to discover frames matching an incoming term.

(a)

```
c_COMPARE( ['chase', 1, 2, 1], ['steeplechase', 1, 1, 2] ).
```

(b)

```
cComp( 43207, [1, 2, 1], 76492, [1, 1, 2] ).
```

(c)

```
c_COMPARE( [term_desc, [[lemma, [chase]], [entry, [1]], [pos_num, [2]],
[def_num, [1]]], [term_desc, [[lemma, [steeplechase]], [entry, [1]],
[pos_num, [1]], [def_num, [2]]] ).
```

Fig. 10: An example of the COMPARE ("see also") relation from the *Collins English Dictionary* (a) as it is produced by the dictionary parser, (b) as it is stored in the Prolog external database, and (c) as it is returned to the rest of the CODER community, as a statement in the FRL.

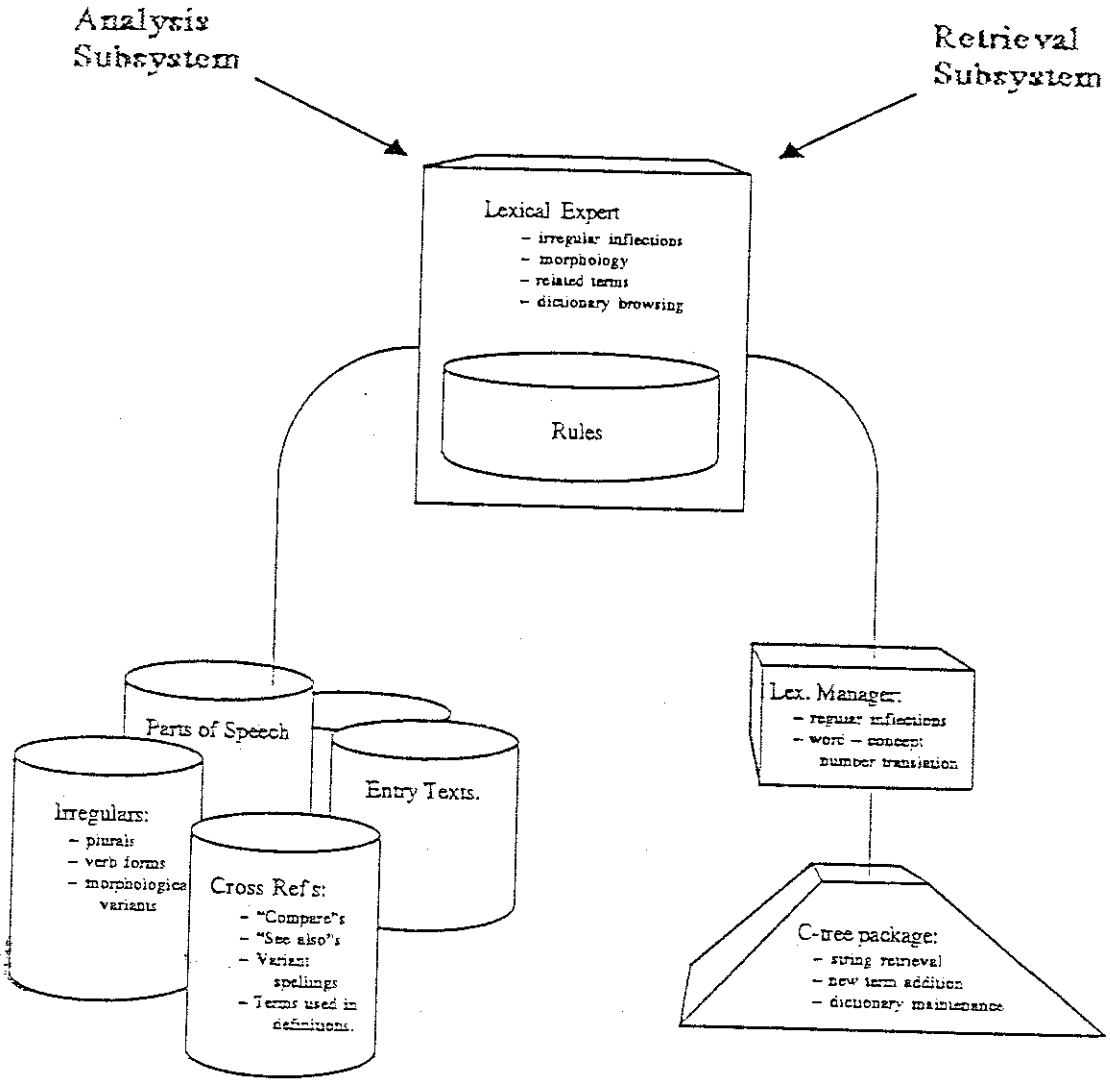


Fig. 11: Internal structure of the lexicon in CODER Version 1.1.

6.2.4. Summary.

All three current knowledge bases depend heavily on the specific structure of the frames in their domain. In the case of the document knowledge base, where least is known about the internal structure of the frames used, a broader matching algorithm is used than that intended in the language design. In effect, frames can be retrieved that include a given piece of knowledge at any level of embedding, but not at a specific level, or in a specific role. The general EKB module has the ability to do either. The other two current implementations are constrained to work with only the particular frame types currently in use to describe their domains. As the picture of their domains is refined, and more types are added, this approach will quickly become unwieldy and prone to errors. Fortunately, we expect the general module to be ready before this happens.

7. Feasibility Testing.

As mentioned earlier, CODER has always been intended to serve as a testbed for the use of AI methods in information retrieval. While ultimately this will include formal comparative testing against current versions of other systems such as SMART and RUBRIC [56], there is a great deal of system extension, tuning, and interface refinement needed before such comparisons will be feasible. Further, there are thorny problems regarding experimental design that must be resolved for such testing to have a chance of yielding insightful results. However, CODER *has* been tested at various stages of its development, and the results of those tests give evidence that we are progressing on the right track. Indeed, "micro" level testing may ultimately prove to be the best way to explore various hypotheses, where slightly different versions of CODER are compared to determine effects of varying approaches to document analysis, retrieval, human-computer interaction, etc.

The first test of CODER concerned the ability of our distributed Prolog system to function in real time on a collection of processors. In early 1987 modules were brought up on a VAX and two SUN computers, and a skeletal version of CODER, involving the blackboard/strategist complex and several simple experts, exhibited reasonable response time while performing a partial search. This successful test of our communications approach led us to proceed with implementing and integrating the various components needed in a distributed expert-based information system [4].

The second test of CODER was in early 1988, when all of the system modules had been at least partially implemented, that is of Version 1.0 as described in [58]. An interface designed for VT100 class terminals allowed users to reply to questions needed for user modelling and to later examine the frames constructed, to read short tutorials about system operation, to browse in a subset of the files from the CED or in the complete machine readable version of the *Handbook of Artificial Intelligence*, to be led through a step-by-step process of Boolean query formulation, to enter p-norm or vector queries constructed from single words, and to apply Prolog search routines to find identifiers of potentially relevant documents. Though only a small number of rules were included to demonstrate "intelligent" information retrieval, much of the needed information for such rules was being collected, and there were "stubs" to facilitate substantial extensions in each local expert.

The third test of CODER, in the Spring of 1988, was of Version 1.1. Term vectors were built for an entire year of AIList Digest issues (1987: this year makes up about a quarter of the total store of some 8000 messages). Frames were constructed for a subset of those messages (January 1987), that being the largest number that could be stored in the current document structure base. C modules adapted from the SMART system provided access to the term vectors, the lexicon B-tree routines gave access to substantial portions of the CED (with morphological analysis support), and improvements were made in the interface and other components. The following subsections give more specifics regarding the Version 1.1 environment.

7.1. Document analysis examples and results.

In Version 1.1, the main change from Version 1.0 was to have a realistic collection of data, and to integrate the lexical manager, the analysis subsystem, and the retrieval subsystem. As such these extensions relate directly to our desire to test, as soon as possible, the utility of using frames for information retrieval.

Figure 12 illustrates, in steps, the analysis of part of a digest from the raw text stage to frames. Part A shows the issue header for one of the digests sent early in 1987. After applying LEX and C processing, tokens are identified and the data is put in the form of Prolog facts, as shown in Part B. Thereafter, Prolog routines manipulate the document representations. A clue dictionary is consulted to help identify the issue date and other fields, and sections of the issue table of contents are broken out, as can be seen in Part C. Ultimately, frames are produced as given in Part D; in particular, issue and topic frames are constructed. Note that the topic slot in the digest_issue frame is filled with a list of slots of type topic, each of which has slots for the

(a) Raw Text Input

AIList Digest Monday, 19 Jan 1987 Volume 5 : Issue 8

Today's Topics:
Seminars - General Logic (SRI) &
Using Fast and Slow Weights (UCB) &
An Implementation of Adaptive Search (SRI) &
The Semantics of Clocks (CSLI) &
Intelligent Database Systems (SRI) &
Formal Theories of Action (SU) &
Mid-Atlantic Math Logic Seminar (UPenn),
Conference - Directions & Implications of Advanced Computing

(b) Raw Text as Prolog Line Facts

```
line(1, [ $parabegin, 1, $hhcap, ailist, 3, $hcap, digest, $space, 12, $isdab, 'Monday', 19, 'Jan', 1987,
        $isdae , $space, 8, $issuebegin, $volissbeg, '5', '8', $volissend, $return]).
line(3, [ $empty_line, 1, $parabegin, 2, $topicsbeg, $return]).
line(4, [ $h1beg, $hcap, seminars, $h1end, $h2beg, $hcap, general, $hcap, logic, '(', $cap, sri,')',
        '&', $hcap, using, $hcap, fast, and, $hcap, slow, $hcap, weights, '(', $cap, ucb,')', '&',
        $hcap, an, $hcap, implementation, of, $hcap, adaptive, $hcap, search, '(', $cap, sri,')', '&',
        $hcap, the, $hcap, semantics, of, $hcap, clocks, '(', $cap, csli,')', '&', $hcap, intelligent,
        $hcap, database, $hcap, systems, '(', $cap, sri,')', '&', $hcap, formal, $hcap, theories, of,
        $hcap, action, '(', $cap, su,')', '&', $hcap, mid, '-', $hcap, atlantic, $hcap, math, $hcap,
        logic, $hcap, seminar, '(', $hhcap, upenn, 2, ')', $h2end, $return]).
...
```

(c) Indexed Facts

```
para(1, 1.1.none.[]).
para(2, 3.4.none.[]).
para(3, 11.11.none.[]).

index($paragraph(1.1.[], 1)).
index($space(1.5.[], 12)).
index($issuedate(1.6.[], 'Monday'.19.'Jan'.1987.[])).
...
index($header1(4.1.[], ($hcap).seminars.[])).
index($header2(4.2.[],
        ($hcap).general.($hcap).logic.'('.($cap).sri.')'.(&).($hcap).using.($hcap).fast.and.($hcap).slow.($
        hcap).weights.'('.($cap).ucb.')'.(&).($hcap).an.($hcap).implementation.of.($hcap).adaptive.($hca
        p).search.'('.($cap).sri.')'.(&).($hcap).the.($hcap).semantics.of.($hcap).clocks.'('.($cap).csli.')'.(
        &).($hcap).intelligent.($hcap).database.($hcap).systems.'('.($cap).sri.')'.(&).($hcap).formal.($hc
        ap).theories.of.($hcap).action.'('.($cap).su.')'.(&).($hcap).mid.(-).($hcap).atlantic.($hcap).math.(
        $hcap).logic.($hcap).seminar.'('.($hhcap).upenn.2.')'.[])).
...
```

(d) Frames Produced

Note: Following are slot names and contents for a digest_issue frame.
Slots with no contents are not shown, in the interest of brevity.
Explanatory comments are included in curly brackets.

```
issue:     {Frame type is 'issue'; contents are shown indented below. }
  isu_date:     {frame type is date }
    year:       1987
    month:       1
    day_of_month:       19
    day_of_week:       Monday
  isu_num:     8
  isu_vol:     5
topic:     {This slot is filled with a list of frames of type 'topic', each
              containing 'header1' and 'header2' slots. A few are shown here.}
  header1:     [$hcap, seminars]
  header2:     [$hcap, general, $hcap, logic, (, $cap, sri, )]

  header1:     [$hcap, seminars]
  header2:     [$hcap, using, $hcap, fast, and, $hcap, slow, $hcap, weights, (, $cap, ucb, )]
  ...
```

Fig. 12: Example of issue header analysis.

two portions of a topic line: header1 and header2.

Figure 13 illustrates the same type of processing, albeit in less detail, for a digest message. In Part A the raw text is shown for a message that is a seminar announcement. After document analysis according to the stages discussed above, frames are again produced, as can be seen in Part B. Here there are a variety of low level frames, for dates, times, names, addresses, etc. that identify the context of word occurrence. There are also some higher level frames describing the document content that allow us to tell, for example, that "Vladimir Lifschitz" is the name of the speaker for the seminar.

Figure 14 gives overall statistics on the first moderate scale test of document analysis routines. As can be seen in Part A, 12 issues were analyzed, and contained almost four thousand lines of text making up 76 messages. Part B illustrates that 112 islands were explored to identify 64 blocks, and that over two thousand frames were automatically identified. In Part C all frames that had more than 15 instances are listed, giving the number identified in this text collection.

Clearly, the CODER analysis subsystem is capable of automatically extracting many of the important low level and some of the higher level frames that most document readers would agree describe useful objects in the domain of this collection. Further, this processing is surprisingly fast, given that a pipeline of interpreted Prolog routines are involved and that the full word list from the CED is accessed when vectors are constructed from the messages.

7.2. Retrieval examples.

Given that frames at various levels are produced, it is important to illustrate how the contextual information provided thereby can be useful to improve retrieval. This is shown in Figure 15 by way of examples taken from the test collection. Three different types of context are given. In Part A, the name "Mark Richer" is shown to appear as a name with an associated electronic address in the first example line. However, the lexeme "mark" occurs as a regular noun as well, as can be seen in the second line. Further, the lexeme "richer" occurs as an adjective, as can be seen in the next line.

In Part B the word "university" is considered. It appears as part of an address, in the subject line of a message, and in its normal usage as part of the body of a message. In Part C the word "January" appears, first as part of the date line of a message, second as part of the subject line, and finally in a line of message body text.

Clearly, then, the CODER analysis subsystem can identify lexemes in varying contexts, and we expect this will allow users to state their information needs with a high degree of

(a) Raw text input.

Date: 17 Jan 87 2119 PST
From: Vladimir Lifschitz <VAL@SAIL.STANFORD.EDU>
Subject: Seminar - Formal Theories of Action (SU)

Commonsense and Nonmonotonic Reasoning Seminar

FORMAL THEORIES OF ACTION

Vladimir Lifschitz

Thursday, January 22, 4pm
Bldg. 160, Room 161K

We apply circumscription to formalizing reasoning about the effects of actions in the framework of situation calculus. An axiomatic description of causal connections between actions and changes allows us to solve the qualification problem and the frame problem using only simple forms of circumscription.

In this talk the method is illustrated by constructing a circumscriptive theory of the blocks world in which blocks can be moved and painted. We show that the theory allows us to compute the result of the execution of any sequential plan.

(b) Frames produced.

Note: Following are slot names and contents for a digest_issue frame.
Slots with no contents are not shown, in the interest of brevity.
Explanatory comments are included in curly brackets.

msg_id: 40931515
date_sent: {frame type is date_time }
date: {frame type is date }
year: 1987
month: 1
day_of_month: 17
time: {frame type is time }
hour: 21
minute: 19
time_zone: PST
from: {frame type is individual }
name: {frame type is name }
first: vladimir
last: lifschitz
email_address: {frame type is email_address }
user_id: [VAL]
source_node: {frame type is node }
local: [SAIL, ., STANFORD]
domain: [EDU]
subject: [$ncap, seminar, -, $ncap, formal, $ncap, theories, of, $ncap, action, (, $cap, su,)]
start_line: 1
end_line: 24
doc_type: seminar
content_frame: {frame type is seminar }
seminar_date: {frame type is date }
month: 1
day_of_month: 22
day_of_week: Thursday
seminar_title: [$ncap, commonsense, and, $ncap, nonmonotonic, $ncap, reasoning, $ncap, seminar, $remn]
speaker: {frame type is individual }
name: {frame type is name }
first: vladimir
last: lifschitz

Fig. 13: Example of digest message analysis.

(a) Description of input

Entity	Number in the input collection
Issues	12
Messages	76
Lines	3910

(b) Results of analysis.

Entity	Number produced
Low-level Prolog facts	3566
Indexed entries from tagging	6109
Islands	112
Blocks	64
Frame related facts	2120

(c) Slots filled in (most frequent slots).

Slot name	Number	Slot name	Number
first	107	day_of_month	92
month	92	year	87
hour	82	minute	82
second	82	time_zone	82
date	76	date_sent	76
email_address	76	end_line	76
from	76	msg_id	76
source_node	76	span	76
start_line	76	subject	76
time	76	user_id	70
header1	69	header2	69
day_of_week	62	domain	6
last	60	route	25
net_name	19	middle	16

Fig. 14: Statistics on document test collection.

(a) Names — First and last names that are common words:

From: Mark Richer <RICHER@SUMEX-AIM.STANFORD.EDU>

... Intentionality is the feature that Brentano cited as the mark of the ...

... only appropriate for algorithm-level theories, provide a richer data ...

(b) Addresses — Noun used in address, subject line and message body:

... University of Illinois ...

Subject: Conference - University Demos at AAAI, ...

... University and research institutes are invited to participate in the ...

(c) Dates — Month name in heading, subject line and message body:

Date: 20 January 87 13:28-EDT

Subject: CSLI Calendar, January 15, No.12

... January 14 meeting only: $15.00 ...

Fig. 15: Examples of disambiguation by context.

precision. However, for that to be useful, the retrieval subsystem must facilitate entry of appropriate specifications, followed by search, retrieval, and presentation. Before implementing all of these capabilities it seemed worthwhile to make a further feasibility test. This is illustrated in Figure 16.

In Part A, six different English statements are given regarding queries supported by the initial frame search implementation (described in §6.2.2). The right column gives a more formal description of each query, referring to frame and slot names in a pseudo-Boolean query notation adapted from the actual Prolog code employed. Conjunctions and disjunctions of clauses are allowed, and slot values can be examined to see if words are included, if values are exactly matched, or if arithmetic comparisons hold true.

In Part B search times are given for these queries. All but the first query gave reasonable response times on a fairly heavily loaded VAX-11/785, and this test was with our first simple implementation. Clearly the initial results are promising. Further enhancement and testing is planned as is discussed in the next section.

8. Future Work.

A wide range of activities are planned regarding further development, testing, and application of the CODER system in general and the fact representation language in particular. First, consider further work on CODER. A variety of enhancement activities should be completed later in 1988. The user interface for VT100 class terminals will be completed, and will be supplemented by a second interface where natural language query processing helps make the dialog structure more flexible. A third interface will also be developed using windowing on a Macintosh II system.

The NU-Prolog compiler from Melbourne University will be extended so that CODER-style communications can be included, and will be ported to other computers so that further development can indeed be done in a distributed environment. Work on NU-Prolog handling of large fact bases will be continued. Significant speed-up in all aspects of the system operation is expected as a result.

Another thread of research involves testing the utility of query expansion based on the use of a large lexicon. Preliminary work, as discussed by Fox and others [20, 21] suggest that retrieval performance will improve and that machine readable dictionaries will produce a rich

(a) Description of each sample query

Number	English Statement	Specification
1	Subject has one of 'expert', 'system' or 'blackboard'	digest_message.subject has ('expert' OR 'system' OR 'blackboard')
2	Date is during 1987 and is before 15th of month	date.year=1987 AND date.day_of_month<15
3	Digest message subject has 'Minsky'	digest_message.subject has 'Minsky"
4	Digest message subject has 'AI'	digest_message.subject has 'AI"
5	Message longer than 40 lins	span.end_line > 40
6	Last name 'Shaffer'	nname.last has 'Shaffer'

(b) Search times (on loaded VAX 11/785)

Query	User CPU secs	System CPU secs	Clock time (secs)
1	17.6	1.2	34
2	4.7	0.5	15
3	6.4	0.3	15
4	6.4	0.3	14
5	1.0	0.1	0
6	1.0	0.0	0

Fig. 16: Queries and their timings.

harvest in terms of word relationships to make this possible. A variety of dictionaries will be integrated to aid in this process. Eventually, a CD-ROM version of the ERIC education database and the ERIC thesaurus will be used for query expansion experimentation.

Regarding the utility of frames, the current preliminary testing will lead to further investigations, after some tuning of the document analysis processing and a complete integration of that with external knowledge bases and the retrieval subsystem. Ultimately, large scale testing is expected with students searching the same collection of AIList messages on CODER, SMART, and a version of RUBRIC. If these tests are successful, further study may involve the use of techniques to "learn" about document types and structures so that the initial knowledge engineering effort now required when applying CODER to a new domain can be significantly reduced.

Bibliography

[1] ANSI Standard: Information Processing — Text and Office Systems — Standard Generalized Markup Language. ISO 8879-1986, ANSI.

[2] Belkin, N.J., T. Seeger, and G. Wersig. "Distributed expert problem treatment as a model for information system analysis and design." *Journal of Information Science* **5** (1983), pp. 153-167.

[3] Belkin, N. J. and W. B. Croft. "Retrieval Techniques." In Martha E. Williams (ed.) *Annual Review of Information Science and Technology v. 22*. 1987, pp. 109-145.

[4] Belkin, N.J., C. Borgman, H. Brooks, T. Bylander, W. Croft, P. Daniels, S. Deerwester, E. Fox, P. Ingwersen, R. Rada, K. Sparck Jones, R. Thompson, and D. Walker. "Distributed Expert-Based Information Systems: an interdisciplinary approach." *Information Processing and Management* **23**:5 (1987), pp. 395-409.

[5] Borgman, Christine L. "Designing an information retrieval interface based on user characteristics." *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (Montréal, Québec: 5-7 June 1985)*. ACM, 1985, pp. 139-146.

[6] Borgman, C.L. "Individual differences in the use of information retrieval systems: a pilot study." *ASIS-86: Proceedings of the Annual Meeting*. Knowledge Industry Publications, 1986, pp. 30-31.

[7] Borgman, C.L. "Individual differences in the use of information retrieval systems: some issues and some data." *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New Orleans, LA: 3-5 June 1987)*. ACM, 1987, pp. 61-71.

[8] Borgman, Christine L. "Information systems functionality: a user-driven perspective." Paper presented at the *Workshop on Distributed Expert-Based Information Systems*. School of Communications, Information and Library Studies, Rutgers University, March 1987.

[9] Brachman, Ronald J., Richard E. Fikes and Hector J. Levesque. "Krypton: a functional approach to knowledge representation." *IEEE Computer* **16**:10 (Oct. 1983), pp. 67-73.

[10] Brachman, Ronald J. and Hector J. Levesque. "The tractability of subsumption in frame-based descriptive languages." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984)*. AAAI, 1984, pp. 34-37.

[11] Brachman, Ronald J. and Hector J. Levesque. *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufman, 1985.

[12] Brachman, Ronald J. and James G. Schmolze: "An overview of the KL-ONE knowledge representation system." *Cognitive Science* **9** (1985), pp. 171-216.

[13] Buckley, C. Implementation of the SMART Information Retrieval System. TR 85-686, Cornell Univ., Dept. of Comp. Sci., May 1985.

[14] Croft, W. Bruce and Roger H. Thompson. "I^{3}R: a new approach to the design of document retrieval systems." *Journal of the American Society for Information Science* 38:6 (1987), pp. 389-404.

[15] Daniels, Penny J. "The user modelling function of an intelligent interface for document retrieval systems." In *IRFIS 6. Intelligent Information Systems for the Information Society (Frascati, Sept. 1985).* Amsterdam: North-Holland, 1986.

[16] Fahlman, Scott E., David S. Touretzky and Walter van Roggen. "Cancellation in a parallel semantic network." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* Los Altos, CA: Morgan Kaufman, 1981, pp. 257-263.

[17] Fikes, Richard and Tom Kehler. "The role of frame-based representation in reasoning." *Communications of the ACM* 28:9 (Sept. 1985), pp. 904-920.

[18] Fox, E.A. "Information retrieval: research into new capabilities." In Steve Lambert and Suzanne Ropiequet (eds.) *CD-ROM: The New Papyrus.* Redmond, WA: Microsoft Press, 1986, pp. 143-174.

[19] Fox, E.A. "Development of the CODER system: a testbed for artificial intelligence methods in information retrieval.' *Information Processing and Management* 23:4 (1987), pp. 341-366.

[20] Fox, E.A. "Improved retrieval using a relational thesaurus expansion of Boolean logic queries." In Martha W. Evens (ed.) *Relational Models of the Lexicon: Representing Knowledge in Semantic Networks.* Cambridge: Cambridge University Press, 1988 (to appear).

[21] Fox, E.A. "Optical discs and CD-ROM: publishing and access." In Martha E. Williams (ed.) *Annual Review of Information Science and Technology Vol. 23.* 1988 (to appear).

[22] Fox, E.A. and Q-F. Chen. "Text analysis in the CODER system." *Proceedings Fourth Annual USC Computer Science Symposium: Language and Data in Systems (Columbia, SC: 8 April 1987).* pp. 7-14.

[23] Fox, E.A. and R.K. France. "Architecture of an expert system for composite document analysis, representation and retrieval." *International Journal of Approximate Reasoning* 1:2 (1987), pp. 151-175.

[24] Fox, E.A., J.T. Nutter, T. Ahlswede, M. Evens, and J. Markowitz. "Building a large thesaurus for information retrieval." *Proceedings Second Conference on Applied Natural Language Processing (Austin, Texas: 9-12 Feb. 1988).* ACL, 1988, pp. 101-108.

[25] Fox, E.A., M.T. Weaver, Q-F. Chen, and R.K. France. "Implementing a distributed expert-based information retrieval system." *Proceedings RIAO (Recherche d'Informations Assistee par Ordinateur) 88: User-Oriented Text and Image Handling March 21-24, 1988, (Cambridge, MA: 21-24 March 1988).* pp. 708-726.

[26] France, R.K. and E.A. Fox. "Knowledge structures for information retrieval: representation in the CODER project." *Proceedings IEEE Expert Systems in Government Symposium (McLean, VA: 20-24 Oct. 1986).* IEEE, 1986, pp. 135-141.

[27] Hahn, Udo and Ulrich Reimer. TOPIC Essentials. Postfach 5560, D-7750 Konstanz 1, Universtat Konstanz, April 1986.

[28] Hanks, P. ed. *Collins Dictionary of the English Language.* London: William Collins Sons & Co., 1979.

[29] Hayes-Roth, Frederick. "Rule-Based Systems." *Communications of the ACM,* **28**:9 (Sept. 1985), pp. 921-932.

[30] Hollaar, L.A. "The Utah Text Retrieval project." *Information Technology: Research and Development* **2**:4 (Oct. 1983), pp. 155-168.

[31] Hollaar, L.A. "A testbed for information retrieval research: the Utah Retrieval System architecture." *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (Montréal, Québec: 5-7 June 1985).* ACM, 1985, pp. 227-232.

[32] Huu, C.T. and U. Kekeritz. "Eine frame implementation in Prolog." *Rundbrief des Fachausschusses* 1.2 der GI (April 1986), pp. 19-25.

[33] Jaeschke, G. "Reciprocal hashing: a method for generating minimal perfect hashing functions." *Communications of the ACM* **24**:12 (Dec. 1981).

[34] Jennings, M. "The Electronic Manuscript Project." *Bulletin of the American Society for Information Science* **10**:3 (Feb. 1984), pp. 11-13.

[35] Lee, Newton S. "Programming with P-Shell." *IEEE Expert* (1986), pp. 50-63.

[36] Lenat, Doug, Mayank Prakash and Mary Shepherd. "CYC: using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks." *AI Magazine* (Winter 1986), pp. 65-84.

[37] Levesque, Hector J. and Ronald J. Brachman. "Expressiveness and tractability in knowledge representation and reasoning." *Computational Intelligence* **3**:2 (May 1987), pp. 78-93. An earlier version appeared in [11], pp. 42-70.

[38] MacQueen, David, Gordon Plotkin and Ravi Sethi. "An ideal model for recursive polymorphic types." *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages (Salt Lake City, UT: Jan. 15-18, 1984).* ACM, 1984, pp. 165-174.

[39] Malone, Thomas W., Kenneth R. Grant, Franklyn A. Turbank, Stephen A. Brobst, and Michael D. Cohen.. "Intelligent information-sharing systems." *Communications of the ACM* **30**:5 (May 1987), pp. 390-402.

[40] Minsky, Marvin. "A framework for representing knowledge." In Winston, P. (Ed.). *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975. Reprinted in [11].

[41] Mylopoulos, John and Hector J. Levesque. "An overview of knowledge representation." In Michael L. Brodie, Joachim W. Schmidt and John Mylopoulos (Ed's.) *On Conceptual Modelling*. Berlin: Springer-Verlag, 1984, pp. 3-16.

[42] Nado, Robert and Richard Fikes. "Semantically sound inheritance for a formally defined frame language with defaults." *AAAI-87: Proceedings of the National Conference on Artificial Intelligence (Seattle, WA: Aug. 1987)*. Los Altos, CA: Morgan Kaufman, 1987, pp. 443-448.

[43] Patel-Schneider, P.F., R.J. Brachman, and H.J. Levesque. "ARGON: knowledge representation meets information retrieval." *The First Conference on Artificial Intelligence Applications (Denver, CO: Dec. 5-7, 1984)*: IEEE, 1984, pp. 280-286. (Also: Fairchild Technical Report No. 654; FLAIR Technical Report No. 29, Sept. 1984).

[44] Patel–Schneider, P.F. "Small can be beautiful in knowledge representation." *Workshop on Principles of Knowledge-Based Systems (Denver, CO: Dec. 3-4, 1984)*: IEEE, 1984, pp. 11-16. (Also: FLAIR Technical Report No. 37, October 1984).

[45] Pigman, Victoria. "The Interaction between assertional and terminological knowledge in Krypton." *Workshop on Principles of Knowledge-Based Systems (Denver, CO: Dec 3-4, 1984)*. IEEE, 1984, pp. 3-10.

[46] Pollitt, Steven. "CANSEARCH: an expert system approach to document retrieval." *Information Processing and Management* 23:2 (1987), pp. 119-138.

[47] Ramamohanarao, K., John W. Lloyd and James A. Thom. "Partial-match retrieval using hashing and descriptors." *ACM Transactions on Database Systems* 8:4 (Dec. 1983), pp. 552-576.

[48] Ramamohanarao, K. and J. A. Shepherd. "An indexing scheme for very large databases of Prolog clauses." *Proceedings of the Third International Conference on Logic Programming (Imperial College, London: July 1986)* 1986, pp. 231-278.

[49] Rowe, Neil C. *Artificial Intelligence through Prolog*. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[50] Scott, Dana S. "Domains for denotational semantics." In Mogens Neilsen and Erik Meinecke Schmidt: *Automata, Languages and Programming: Ninth Colloquium (Aarhus, Denmark: July 12-16, 1982)*. Berlin: Springer-Verlag, 1982, pp. 577-613.

[51] Smith, Linda C. and Amy J. Warner. "A taxonomy of representations in information retrieval system design." In Hans J. Dietschmann (Ed.) *Representations and Exchange of Knowledge as a Basis of Information Processes*. New York: North-Holland, 1984, pp. 31-49.

[52] Smith, J.B., S.F. Weiss and G.J. Feruson. "MICROARRAS: an advanced full-text retrieval and analysis system." *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New Orleans, LA: 3-5 June 1987).* ACM, 1987, pp. 187-195.

[53] Thompson, Roger H. and W. Bruce Croft. "An expert system for document retrieval." *Expert Systems in Government Symposium (MacLean, VA: Oct. 24-25, 1985).* IEEE, 1985, pp. 448-456.

[54] Thompson, Roger H. "An implementation overview of I^{3}R." Paper presented at the *Workshop on Distributed Expert-Based Information Systems.* School of Communications, Information and Library Studies, Rutgers University, March 1987.

[55] Tong, Richard M., Lee A. Appelbaum, Victor N. Askman and James F. Cunningham. "RUBRIC III – an object-oriented expert system for information retrieval." *Proceedings IEEE Expert Systems in Government Symposium (McLean, VA: 22-24 Oct. 1986).* IEEE, 1986, pp. 106-115.

[56] Tong, Richard M., Lee A. Appelbaum, Victor N. Askman and James F. Cunningham. "Conceptual information retrieval using RUBRIC." *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research & Development in Information Retrieval (New Orleans, LA: June 3-5, 1987).* ACM, 1987, pp. 247-253.

[57] Touretzky, David. *The Mathematics of Inheritance Systems.* Los Altos, CA: Morgan Kaufman, 1986.

[58] Weaver, Marybeth T. Implementing an Intelligent Retrieval System: The CODER System, Version 1.0. MS Thesis, VPI&SU Computer Science Dept., February 1988, Blacksburg, VA (available from VPI&SU as TR-88-6).

[59] Wohlwend, Robert C. Creation of a Prolog Fact Base from the Collins English Dictionary. MS Report, VPI&SU Computer Science Dept., Blacksburg, VA, March 1986.

Appendix A: Frame Administration Modules

A.1. The frame type manager.

A.1.1. New frame types.

For each frame type, a Prolog fact contains the frame characteristics.

ka_frame(Frame_name, Parents, Slot_list).

where

- *Frame_name* is any name not already used as a frame, EDT or relation type name.
- *Parents* is a list of frame type names representing classes of which this frame is a member. For example, a ***journal_article*** frame type could have parent frame type ***journal*** which could have parent frame type ***bibliographic_reference***.
- *Slot_list* is a list of attribute names and characteristics. All parent slots are inherited, that is they are added to the list of slots defined for a new frame type.

A list of characteristics per slot is contained in the slot_list. Slot characteristics are represented by the list

[Slot_name, Class, Type, Cardinality_min, Cardinality_max, Default]

where

- *Slot_name* is a slot identifier. Although slot names must be unique within a given frame type, they need not be unique among all frame types. However, it is recommended that all slots be given unique names to avoid confusion.
- *Class* identifies the kind of object required to fill the slot. It must be e (EDT), f (frame) or r (relation).
- *Type* is the type name of the EDT, frame or relation to fill the slot.
- *Cardinality_min* is the minimum number of values allowed for the slot when the frame object is instantiated. All slot values for frame objects are stored as *lists* of values. This value may be null if there is no lower bound on the number of slot values.
- *Cardinality_max* is the maximum number of values allowed for the slot when the frame object is instantiated. This value may be null if there is no limit.

- *Default* is the value assigned to an EDT slot when a frame object is created.

A.1.2. Frame lattice manipulations.

When a new frame type is created, Prolog facts per parent-child relationship for the frame are also asserted so that the frame hierarchy may be efficiently navigated. Facts contain the parent frame type and its direct descendant frame type: *ka_fparent(Parent_type, Child_type)*. Frame taxonomy relationships are reported by *subframe_list(Frame_type, Subframes)* and *superframe_list(Frame_type, Superframes)* which return a list of immediate subframe types and a list of all ancestors respectively. These predicates may also either succeed or fail if both arguments are bound, thus indicating whether the frame types provided have a parent-child relationship.

The *subsumes(Ancestor, Descendant)* predicate indicates whether one frame type is a generalization of another. Any frame stored as a parent of another frame should subsume its child frame. However, a parent-child link is not required for one frame type to subsume another. Finally, the frame type manager of the KA complex includes the *Slot_list(Frame_type, Slot_list)* predicate to return a frame type's list of slots or to succeed if a given list of slots is a proper subset of the frame's slot list.

A.2. The frame object manager.

The *new_frame* predicate allows establishment of a new frame object. This predicate is of the form: *new_frame(Frame_type, Frame_object)*. Any CODER expert may issue the *new_frame* predicate to create a new frame object as long as the object manager code has been consulted. When issued, the frame_type must be bound to a defined type; frame_object may be unbound, in which case it will be returned as the object identifier which has been assigned. If the module creating the object wishes to assign its own identifiers, it may do so. However, the identifier assigned must be unique within the module's local fact base of frame objects. Two types of assertions occur to create a new frame object: *fobjid(Object_identifier, Frame_type)*. and *fobj(Object_identifier, Slot_name, Slot_value_list)*. Sample facts representing two frames are listed in Figure A.1. When new frames are created, the fobjid fact is asserted. Next, any slots having non-null default values are asserted as fobj facts. Therefore, a fobj fact will not necessarily exist for every slot defined for a

given frame type. Indeed, no fobj facts are required when a frame object is created. Once a frame object has been created, values such as defaults may be removed and/or new values may be assigned.

```
fobjid(163610917, user_eval).
fobjid(163610918, session).

fobj(163610917, satisfaction, [8]).
fobj(163610917, usefulness, ['67-90']).
fobj(163610917, why_stop, ['out of time']).
fobj(163610917, easy_to_use, [y]).
fobj(163610918, nodoc_queries, [0]).
fobj(163610918, doc_quantity, [10]).
fobj(163610918, user_eval, [163610917]).
fobj(163610918, sessionlgth, [154]).
fobj(163610918, session_id, [26968]).
```

Fig. A.1. Sample frame objects.

The method used to store frame objects does make less efficient use of storage space than other methods considered. However, it simplifies processing of slot value manipulations and reduces execution time processing by eliminating the list traversal required by other methods. Two other possibilities were examined in conjunction with the one adopted.

- For each new frame object, a single fact could be asserted as: *fobj(Object_id, Frame_type, [Slotname.[Values], Slotname.[Values], ...])*. All slot names and a list of values for each would be included in a single fact. Although this method would considerably reduce the size of the knowledge bases created and eliminates the redundant storage of the object identifier necessary in the method chosen, it would require list processing and manipulation every time a slot value were added or removed. In addition, the frame object manipulation predicates would require excessive list processing. The frame objects in Figure A.1 would be replaced by two facts:

```
fobj(163610917, user_eval, [satisfaction.[8], usefulness.['67-90'], why_stop.['out of time'], ... ]).
fobj(163610918, session, [nodoc_queries.[0], doc_quantity.[10], user_eval.[163610917], ... ]).
```

- To reduce storage requirements even more, the Slotname could be eliminated from the list of slot names and values. Instead, each list of values would positionally be matched to the slot names defined in the frame type definition.

So, the single fact for each object would contain: *fobj(Object_id, Frame_type, [[Values], [Values], ...])*. This method would require even more extensive list manipulation as well as matching of the frame type slots to the object slot value list for all frame object manipulation. In cases where no values were assigned to slots, each slot would still have to be included in the list so that the positional values could be properly matched to slots in the frame type definition.

The implementation strategy for frame objects could be rewritten according to one of the above methods or using some other strategy. Such modification, however, would require rewriting of all frame object predicates as well as rewriting of modules which use the current frame object structure.

A.2.1. Object manipulation.

Predicates to support the manipulation of frame objects allow updates to slot values, and support reasoning about the relationships between or among frame objects. The *is_frame* and *has_slot_value* predicates return information about the existence of frame objects and the values assigned to frame slots, respectively. Slot values are asserted using the *set_slot_value* predicate, and may be removed with the *remove_slot_value* predicate. The *equal_frames* and *matching_frames* predicates allow comparison between frame objects. A frame object A matches a frame object B if every filled slot of A matches a filled slot of B. Slot values *match* when slot types, classes and values match or when values for a slot which subsumes another slot match. *Matching* is an antisymmetric relation, whereas *equal* is a symmetric relation: every slot in A must match a slot in B and every slot in B must match a slot in A.

Appendix B: External Knowledge Base.

External knowledge bases provide mechanisms for transparent storage, indexing, and retrieval of large numbers of facts about individual entities in the CODER problem universe 'Facts' in this context is a technical term, refering to a ground instance of the CODER logical relation data type. This data type has been designed to parallel the syntax of propositions in the Prolog language, so CODER facts can be mapped directly to Prolog facts. Specifically, each fact can expressed as a Prolog proposition that includes no variables. The proposition may have other propositions nested within it arbitrarily deeply, but eventually the tree formed by such propositions will terminate in objects of the other two CODER data types: frames and elementary data objects.

A fact base supports a single function for storing new facts

enter (fact, source_id).

and three functions to retrieve facts, one for each data type:

facts_with_rel (skeletal_relation, [fact | _]).
facts_with_frame (frame_type, frame_object, [fact | _]).
facts_with_value (data_type, data_object, [fact | _]).

Specialized functions provide for the case where the relation being matched is the head of the fact:

facts_matching (skeletal_fact, [fact | _]).

and the case where it is only required to know what objects in the fact base match a given frame, rather than what facts are known about the objects:

frames_matching (frame_type, frame_object, [frame_object | _]).

In addition, three parallel functions are provided which return the number of facts that any of the primary retrieval functions would retrieve:

num_with_rel(skeletal_relation, Num).
num_with_frame(frame_type, frame_object, Num).
num_with_value(data_type, data_object, Num).

These allow experts calling the fact manager to guard against unused large retrieval sets. None of the retrieval operations succeed if called with their first arguments unbound or their last argument bound.

B.1 Detailed Description of Functions

enter (fact, source_id). Where **fact** is a ground instance of a CODER relation, every argument of the fact is a syntactically correct use of a recognized relation, frame, or elementary data type, and **source_id** is an atom specifying the source of the fact, succeeds while updating the local fact base to include **fact**. This updating includes time-stamping the fact to facilitate later knowledge maintenance and analyzing the fact so that later retrieval can occur on any of its component relations, frames, or elementary data objects. Fails if and only if **fact** uses unknown data types, includes type violations, or is not syntactically correct.

facts_with_rel (skeleton, [fact | _]). Where **skeleton** is an instance of a recognized relation and all the arguments of **skeleton** are either frame objects, elementary data objects, skeletal relations, or CODER variables, succeeds while binding the second argument to the list of all facts in the local fact base containing ground instances of relations that can be unified with **skeleton**. The order of elements in the list is indeterminate, and may (or may not) vary from call to call. If no facts in the local base contain relations can be unified with **skeleton**, succeeds while binding the second argument to the empty list. Fails if and only if **skeleton** uses unknown data types, includes type violations, or is not syntactically correct.

facts_with_frame (frame_type, frame_object, [fact | _]). Where **frame_object** is a valid frame object of type **frame_type**, possibly with some or all slots unfilled, succeeds while binding the second argument to a list of all facts in the local fact base

that include references to frames of type **frame_type** with at least those slots filled with matching values. In other words, all facts that reference frames that are exactly like **frame_object** and all facts that reference frames that are like **frame_object** except that they have additional slots filled, but no facts that have the same slots filled by non-matching values, and no facts that have unfilled slots where **frame_object** has filled slots. The order of elements in the list is again indeterminate, and the list may again be empty, but the function fails if and only if **frame_object** uses unknown data types, includes type violations, or is not syntactically correct.

facts_with_value (data_type, data_object, [fact | _]). Where **data_object** is a valid elementary object of type **data_type**, succeeds while binding the second argument to a list (in indeterminate order) of all facts in the fact base that reference **data_object** at some degree of recursion. May succeed while binding the second argument to the empty list if no such facts are available, but will fail only if **data_object** is not an object of type **data_type**.

facts_matching (skeletal_fact, [fact | _]). Performs exactly as **facts_with_relapse**, except that the retrieved facts are constrained to be only those whose head relations match the head relation of **skeletal_fact**.

frames_matching (frame_type, frame_object, [frame_object | _]). Where **frame_object** is a valid frame object of type **frame_type**, possibly with some or all slots unfilled, succeeds while binding the second argument to a list of all frames of type **frame_type** referenced by facts in the local fact base that match **frame_object**.

num_with_rel (skeleton, Num).

num_with_frame (frame_type, frame_object, Num).

num_with_value (data_type, data_object, Num). All function exactly as their corresponding functions above, except that on succeeding they bind their final arguments to the number of facts that the corresponding function would provide in its list.