Architecture of an Object-Oriented Expert System for Composite Document Analysis, Representation, and Retrieval

Edward A. Fox
Robert K. France

April 1986

TR–86–10

Architecture of an Object-Oriented Expert System for Composite Document Analysis, Representation, and Retrieval[†]

Edward A. Fox
Robert K. France

Department of Computer Science
Virginia Tech, Blacksburg VA 24061

ABSTRACT

The CODER project is a multi-year effort to investigate how best to apply artificial intelligence methods to increase the effectiveness of information retrieval systems while handling collections of composite documents. In order to ensure system adaptability and to allow reconfiguration for controlled experimentation, the project has been designed as an expert system. The use of individually tailored specialist **experts** coupled with standardized **blackboard** modules for communication and control and **external knowledge bases** for maintenance of factual world knowledge allows for quick prototyping, incremental development, and flexibility under change. The system as a whole is structured as a set of communicating modules, designed under an object-oriented paradigm, and implemented under UNIX™ using pipes and the TCP/IP protocol. Inferential modules are being coded in MU-Prolog; non-inferential modules are being prototyped in MU-Prolog and will be re-implemented as needed in C++.

CR Categories and Subject Descriptors: H.3.1 **[Information Storage and Retrieval]:** Content Analysis and Indexing; H.3.3 **[Information Storage and Retrieval]:** Information Search and Retrieval; H.3.4 **[Information Storage and Retrieval]:** Systems and Software ; I.2.1 **[Artificial Intelligence]:** Applications and Expert Systems; I.2.4 **[Artificial Intelligence]:** Knowledge Representation Formalism and Methods; I.2.7 **[Artificial Intellligence]:** Natural Language Processing ; I.2.8 **[Artificial Intelligence]:** Problem Solving - *control methods and search*; K.6.1 **[Management of Computing and Information]:** Project and People Management; K.6.3 **[Management of Computing and Information]:** Software Management

General Terms: Design

Additional Keywords and Phrases: abstract data type, architecture or distributed expert system, blackboard, composite document, knowledge base, message passing, natural language processing, PROLOG, relational lexicon, user interface

†Project funded in part by grants from the National Science Foundation (IST-8418877) and the Virginia Center for Information Technology (INF-85-016).

1. Purpose

As the world's pool of information, and particularly of machine-readable information, becomes larger, it becomes increasingly necessary to engage the help of computers to control and manipulate it. Initial attempts at computer-aided text storage and retrieval, however, focused principally on performance and have achieved only moderate levels of effectiveness. The CODER (COmposite Document Expert/Extended/Effective Retrieval) expert ISR system is a research system intended to address these problems through the mechanisms of knowledge-based and goal-directed AI techniques. In keeping with its purpose as a research tool, the system is designed with sufficient flexibility to encompass a wide range of experimental techniques. The system is also designed to permit evolutionary development, both of its overall design and of its specific functional modules. The construct of a **moderated expert system** has been instrumental in achieving both of these design goals.

Particular attention is being given in the system to analysis and representation of heterogeneous documents, such as electronic mail digests or messages, which vary widely in style, length, topic, and structure. Understanding the structure of such documents is one ability that allows human catalogers and retrieval specialists to out-perform conventional information retrieval systems. Knowledge of document structure will be combined in the system with knowledge of the entities that make up documents (words, sentences, dates, or electronic addresses, to give a few examples) to accomplish both more refined analysis and more satisfactory retrieval of documents (or sections of documents) that satisfy the information needs of the system users. The initial application is to support queries directed toward retrieving any relevant passage in the messages included in a three year archive of issues of the ARPANET **AIList** digest.

This effort has evolved in part out of prior studies with the SMART information retrieval system, where use of an extended Boolean logic proved beneficial, and where query expansion using a relational lexicon improved search effectiveness [SALT 83]. By representing the uncertainty involved in indexing, and by viewing query processing as a type of inexact reasoning, later versions of the SMART system successfully used the **p-norm** model to support "soft" Boolean evaluation. Relational models of the lexicon were particularly appealing since automatic processing of machine-readable dictionaries could be employed to identify many of the key relations for a relatively large vocabulary. Another origin of this effort was work applying information retrieval techniques to the construction of expert systems [WINE 85]. It became clear that a rule-based approach would allow systems to be tailored to (classes of) users, and that several search algorithms could be combined dynamically to give instantaneous rather than a "batch" type of feedback. This work implies that planning and scheduling operations should be an integral part of an analysis and retrieval system.

Furthermore, it was clear that whereas complex probabilistic models of queries had been investigated, relatively simple document analysis was involved in prior work. Now that powerful AI engines are becoming available, it seems timely to examine the benefits of carrying out a partial natural language analysis of the incoming documents. Such a semi-controlled knowledge acquisition process, where more precise and comprehensive knowledge representation is possible, allows some questions to be more effectively handled, and others to be answered for the first time.

2. Functional Description

Conceptually, CODER can be thought of as being composed of two overlapping subsystems: the Analysis Subsystem and the Retrieval Subsystem. Actually, the system is so constructed that any number of instances of these subsystems can be running at any given time, but it is useful nonetheless to envision it as being composed of two parts, one for each of the system's principle tasks of information cataloging and information retrieval. These two parts are in many ways mirror images of each other: each is modeled as a community of experts and each makes use of the central

"Spine" of knowledge banks and knowledge administrators (see Fig. 1). Many of the experts perform similar functions in the two subsystems and use the same bodies of specialized knowledge: what they do with this knowledge, however, and above all the strategies for combining the experts' opinions, can differ radically from subsystem to subsystem.

CODER is a knowledge-intensive system. Each of the experts includes its own base of specialized knowledge about its field. Beyond these, however, the system must also manage large amounts of three basic types of knowledge:

- knowledge about documents stored in the system,
- knowledge about terms in the language, and
- knowledge about users and their information needs.

Facilities must therefore be provided in the system for knowledge representation and knowledge maintenance. These facilities, separate from the knowledge manipulation abilities required by the experts, are provided by non-inferential modules which can be separately implemented and optimized. Knowledge representations are moderated by a set of type managers which provide the abstract structures necessary for modelling attributes, objects, and logical relations among objects. Knowledge maintenance functions are provided by specialized external knowledge bases, which provide storage and quick recall for the "gratuitously complex" facts of the problem universe. Isolating these collections of specific facts allows expert design to focus on general knowledge of principles and on the interaction of these principles with the complexities of the problem universe.

2.1. The Analysis Subsystem

The purpose of the Analysis Subsystem is to construct consistent sets of propositions describing input documents. These propositions can then be stored in a Document Knowledge Base and eventually used by the Retrieval Subsystem to judge the relevance of a document to a given query. To this end the Analysis Subsystem must decide both *what a document is* and *what it is about*. Processing a document, in other words, involves classifying it (as, for instance, a journal article or a bibliography) and cataloging it under appropriate concepts. It is also necessary to establish **structured data** about the document: it's author, date of origin, and so forth. Experts in structured data recognition and document classification interact with experts in natural language parsing, indexing and concept identification to form hypotheses of the most important features of each document. When hypotheses of sufficient levels of confidence can be combined into consistent interpretations, they are passed to the Document Knowledge Base for storage. (Note that while each interpretation is required to be internally consistent, there is no reason to expect them to be consistent among themselves. Different interpretations may emphasize different aspects of a given document, or may represent different parses of the same text.) A document may produce one or several interpretations.

2.2. The Retrieval Subsystem

The Retrieval Subsystem performs the dual of this process. Based on a query from a user, the Subsystem develops an abstract representation of her information need and searches the cataloging information in the Document Knowledge Base to establish a set of documents that may satisfy it. To this end, it must marshall expertise in understanding the user's query, in modelling the user herself, and in navigating the Document Knowledge Base. It may also call on knowledge of words and concepts to expand the query, or it may rely on characteristics of document types or structured data (it may, for instance, look for *journal articles* with a specific *author)*, or use more subtle means, such as co-citation clustering. Reports of its progress are constantly submitted to the user, either in the form of documents that may fill her information need or in the form of terms (or complex structures built of terms) that may better express that need. The relevance of an entity (document, term, or structure) to the user's information need is the focus of a dialog between the user and the experts in the Retrieval Subsystem that continues until the information need is satisfied.

2.3. The Spine

Central to the system is a set of knowledge bases and type managers that provide control and bulk storage for the information that is used by both the Analysis and Retrieval Subsystems. This set of relatively "dumb" modules is called the Spine as a metaphor for its relationship to the "smarter" experts in the two Subsystem communities. There are several component modules in the Spine, falling into two broad functional areas: external knowledge bases such as the Document Database or the Lexicon, and the system resources of the Knowledge Administration Module. All these components are equally usable by both Subsystems, although the amount and type of usage may vary. All the components as well have a privileged interface through which they are accessible to a System Administrator. Comparable to a Database Administrator interface, this interface provides a deep (and dangerous) level of inspecting and changing functions for knowledge base verification and maintenance. None of the system modules have access to this level.

The Knowledge Administration Module catalogs and classifies the types of entities about which the system has knowledge, ensuring consistent representation and use of knowledge throughout the system. The set of types chosen, after consideration of several different AI formalisms, allows for three types of knowledge objects: **elementary** objects such as characters, integers, lists of lists of characters and so forth; semantically structured **frame** objects such as documents, dates, sentences and so forth; and logical **relations** over objects. Canonically, these types of entities can be used to represent knowledge of attributes, individuals, and facts about individuals, respectively; the formalism is, however, sufficiently flexible to represent AI structures such as conceptual dependencies and semantic networks as well.

Common storage is provided in the Spine for detailed knowledge of each of the primary types of entities in the system's universe. The Lexicon, for instance, maintains knowledge about terms in the language. It can be conceptually divided into two parts, one of general linguistic knowledge and the other of specialized world knowledge, particular to the collection of documents employed. Although knowledge from both conceptual halves may be recalled by a given request, tagging the knowledge in this way promotes portability by allowing knowledge of general use to be decoupled from the pragmatics of a given document collection and re-used in other applications. Similarly, the Document Knowledge Base maintains facts about the documents. Attached to the Knowledge Base is a simple resource manager providing storage and retrieval for raw document text. These two modules together are referred to as the Document Database. Finally, there is a User Model Base of facts about individual users. These include reports of occurrences during a single session and general statements about the user, such as the type of information that has proven relevant to the user in the past, semantic knowledge particular to or supplied by the user, and common characteristics of relevant documents. These bodies of knowledge inform the system's response in intelligently analysing and retrieving documents.

3. Structural Description

CODER is an ambitious system in several ways. Its very size is unusual in the domain of AI programming, as is the diversity of methods that can be expected to be applied to the system tasks over its lifetime. The system must manage knowledge of diverse entities, utilizing a variety of knowledge representations. The size and complexity of the system provokes problems of coordination, both among the parts of the system and among the people implementing it.

We have attacked this complexity through the use of two organizing constructs. From object-oriented programming, we have borrowed the paradigm of **classes of objects**, and from expert systems, that of the **community of experts**. Using the principles of object-oriented programming, we have been able to define a few general classes of objects out of which the system can be constructed. The construct of the community of experts then provides a model for knowledge-based problem decomposition: a problem task is broken into a set of subtasks each of which can be addressed by an expert in some specialized domain of knowledge.

Breaking the system into specialized experts alleviates the engineering problems of structuring the system and distributing responsiblity for its construction among several implementors. It creates new problems, however, in the operation of the system on any individual task. Since each expert is a specialist in a narrow subtask, none is equipped to solve the task as a whole. In order to address the overall task, the experts must communicate among each other, and in order to solve it effectively, their efforts must be coordinated according to some global strategy. The concept of a blackboard [ERMA 80, HAYE 85] has been used in several expert systems to provide communication among experts: in CODER we have specialized the blackboard construct to include an active planning principle for expert coordination.

Regarding the objects in the system as being instances of classes has several advantages. First, many modules can be implemented once on a generic level and the generic module specialized to create individual system objects. Second, even for modules (such as the individual experts) that must be built individually, the object-oriented approach provides an effective way of specifying the external behavior of all the modules in the class. There are also payoffs in the design phase: thinking of modules in generic terms promotes a cleaner design, and makes the overall picture easier to understand.

Designing the system as a collection of communicating objects also makes easier two system goals that could otherwise be quite sticky. In the real world of limited machines and programming environments, CODER presents implementation difficulties both in the number of modules that it comprises and in the amount of knowledge necessary for its function. Not having a large mainframe to dedicate to the sole purpose of running the system, we very early made the design decision to distribute the system among a group of networked machines in the local environment, all of which run UNIX™ and thus support intra-machine communication via the pipe construct and inter-machine communication via the TCP/IP protocol [LEFF 84]. This decision allows us to place the large external knowledge bases where there is sufficient storage, the user interfaces where there are bit-mapped screens, and the inferential modules where there is computational power. Designing the system as a collection of communicating objects makes it possible to implement the distribution of the system separately from the system architecture itself. Each module can be built as a single process (or, conceivably, set of processes) running on a single machine. Communication with other modules is then handled through abstract message-passing primitives. All details of the location of the other modules in the system and the protocols needed to reach them are hidden within the communication managers that implement these primitives. At the level of system objects, all the implementor needs to know are abstract names.

In addition, the message-object paradigm helps hide language-dependent details. CODER involves both modules such as the experts and blackboard strategists that are by their nature inferential and modules such as the type managers and external knowledge bases that have no such commitment. In fact, some portions of the user interfaces are best written procedurally, in a language with sufficient low-level primitives for bit manipulation. These conflicting demands have made it desirable to build the system in a multi-language environment. Inferential modules are being coded in MU-Prolog, a UNIX-based dialect of Edinburgh Prolog supporting extended communication and database operations. Procedural modules are being coded in C++, a dialect of C that supports the class-object paradigm [STRO 85]. Modules with no paradigm requirement will be prototyped in MU-Prolog and may then be re-implemented in C++ as required for efficiency. Consistency between languages is maintained at the level of module-to-module communications.

Standardizing the message protocol between objects and treating the knowledge representations as abstract objects themselves makes it possible to conceal the language in which a given module is implemented from all the other modules in the system.

4. Object Types

The basic building blocks of the CODER system are detailed below. Objects of these four classes, together with the environment provided by the knowledge representation administrators and communication managers, make up the entire system.

4.1. Blackboards

A blackboard is an area for communication between experts. This communication takes place through the medium of posting and reading hypotheses in specialized subject areas. In CODER blackboards, a specialization of this process provides a means for asking and answering questions, which are contained in their own special area of the blackboard. The importance of this type of communication was noted convincingly in [BELK 84]. In addition, the CODER blackboards provide a special area, maintained by a blackboard management expert called the **Blackboard Strategist**, containing a small set of consistent hypotheses of high certainty. This pending hypothesis area is available for read access by all experts and thus, indirectly, by the outside world. It provides an instantaneous picture of the "consensus" of the blackboard; i.e., what the system as a whole hypothesizes about the problem under consideration at any moment.

An hypothesis is represented by a four-tuple: the **fact** hypothesized, the **expert** hypothesizing it, the **confidence** that the expert has in it, and its **dependencies** on other hypotheses. This additional information, besides providing motivating information for the distillation of the set of pending hypotheses, allows truth-maintenance functions to be performed within the blackboard subject areas. If an expert withdraws an hypothesis, for instance, or radically changes the confidence level with which it proposes it, this information makes it possible to schedule reconsideration of the hypotheses depending on it.

Monitoring the blackboard for this sort of event is one function of the blackboard strategist. Since the rules governing truth maintenance are independent of the particular predicates involved in the facts hypothesized, this function is independent of the application task of the blackboard community. The strategist also monitors the blackboard for task-specific events and conditions that trigger new processing. These two categories of function are kept separate in the strategist, so that the truth maintenance function can be transported to other tasks. Nonetheless, they have both been designed as rule interpreters: neither the strategies involved in truth maintenance nor those involved in analysis or retrieval are yet well-understood. Consigning these strategies to a rule base allows them to be changed easily without the entire blackboard needing to be re-implemented.

The final component of the strategist is a scheduler for the tasks identified by the other two components. Acting on rules of its own that relate static concerns of how many experts of what type should be active at the same time on which machines to the mix of tasks scheduled by the truth-maintenance and task-oriented components, the scheduler issues *wake* and *sleep* commands to the experts in the blackboard community. This allows different groups of experts to be active at different phases in the community task, but allows experts outside the currently active group to be called up to answer a question or to reconsider an hypothesis.

4.2. Experts

An expert is, conceptually, a specialist in a certain restricted domain pertinent to the task at hand. Experts are designed to be implemented in relative isolation from one another: no expert has knowledge of the other experts in the community, and all experts communicate with the community strictly through the construct of the blackboard. 'Isolation,' of course, is a relative term here. Part of the specification of an individual expert is the set of predicates that it may view in a blackboard area and the (possibly overlapping) set of predicates that it may post back. Obviously, there must be agreement among the expert implementors on the structure and bounds of those predicates if the experts are to work together. What each expert does with those predicates, however, and what internal knowledge and processes it uses to produce new hypotheses, are left to the implementor of the individual expert. Each expert can therefore be built in the way that best takes advantage of the characteristics of its particular domain of expertise.

An expert has only two requirements for its operation: it should be knowledge-driven, and it must recognize the appropriate commands from the strategist scheduler. The first is philosophical in nature: it is part of the CODER design that the complexities of the system tasks be realized in the knowledge required for their execution, rather than the process of execution itself. In the case of experts, this implies that expertise be represented as explicit knowledge, separate from whatever engine manipulates it. The knowledge in the expert, moreover, is constrained by system design to be general knowledge: either rules for finding and manipulating factual knowledge in one of the external knowledge bases, or facts that can be applied to classes of objects in the problem universe. The second requirement is pragmatic: for the strategist to schedule their activity properly, experts must go through a canonical cycle of operations.

Assigning an expert to a small area of specialization and decoupling it from the remainder of the system has several advantages. First, the development of the expert is separated from that of the surrounding system. Interaction problems, normally a plague of Artificial Intelligence systems, are thereby kept to a minimum. In addition, the experts are kept small, so problems of rule interaction within the expert are minimized. Tasks which are found to require too much complexity can be further subdivided along the lines of the areas of expertise required to solve them, until they are reduced to manageable size.

As a further benefit, experts can be specialized to deal with different types of knowledge, so that an expert that manipulates knowledge of *how* to do things can use different inference mechanisms than an expert in *what* to do. This will enable the reasoning portions of these experts to run with more efficiency than, for instance, general rule-based inference engines (see [LEVE 84] for a formal analysis of this effect). In practice, we expect a few generic knowledge-handling engines can be used to produce a plethora of different experts. Possible engines include both forward-chaining and backward-chaining rule interpreters, frame-based classification engines, and pattern-matching engines. These generics would then be associated with different rule bases, classification trees, and similarity measures, respectively, to produce specialized experts in a variety of disciplines. The work of Chandrasekaran (e.g., [CHAN 85]) holds great promise that a few such powerful generics can be isolated and put to good use. And as such methods are better understood, they can be used in CODER to build new experts, again without affecting the existing modules of the system.

4.3. External Knowledge Bases

An external knowledge base (or "fact base") is an object for storage and retrieval of factual world knowledge. The Document Knowledge Base, the Lexicon, and the User Model Base are all specializations of this class: others can be created as needed during the development of the system. Each maintains knowledge about a particular class of objects in the form of specific statements of fact. This is in direct contrast to the internal knowledge bases of the specialist experts, which are primarily composed of general rules.

Formally, propositions entered into a Fact Base are required to be *ground instances* of logical relations known to the system; that is, to involve neither unbound variables nor meta-terms. These propositions 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. Alternately, the Knowledge Base may be queried with an object (either an elementary term, a frame, or a relation) and will return the set of all facts involving that object. (Note here that, since facts are simply instances of logical relations, querying with a skeletal fact is a special case of querying with a skeletal relation. Restricting the search for matching relations to those that occur as the head of the fact is, however, sometimes advantageous.) In addition, a Knowledge Base may be queried about the *number* of facts that match an object or a skeletal fact: this information can be used by the querying entity for statistical purposes, or simply to avoid receiving excessively large sets of facts.

All knowledge in an External Knowledge Base is associated (explicitly or implicitly) with the source of the fact and with the date, time, and session of its entry into the Knowledge Base. Knowledge can be entered by individual modules during system operation, in which case each fact is individually and explicitly stamped, or it may be entered by the System Administrator, in which case the information is implicit in the names of the fact predicates entered. This information, available to the System Administrator only, allows suspect facts to be traced and updated. A function is provided to the System Administrator to delete facts from the Knowledge Base and to add facts *en masse* from sources external to the system; in addition, the System Administrator has access to all of the functions described above.

4.4. User Interfaces

There are three points at which users interact with the CODER system. The System Administrator interacts directly with the Spine to define new types of knowledge to the system and to fine-tune the system knowledge bases. This is a classical, command-driven interface: the Administrator is assumed to be sufficiently sophisticated to interact directly with the modules being managing. The other two interfaces (one for document entry sessions and one for document retrieval sessions) are more complex, since the entities communicating are more complex. On one side, the users are expected to be less sophisticated and have less precise formulations of their tasks. On the other, the interfaces relate the users not to functionally discrete modules, but to the blackboard-moderated communities accomplishing these tasks. The interactions moderated by these interfaces are less sequences of commands and command executions than dialogues seeking concensus on the nature of the task to be executed.

The interfaces for the Analysis and Retrieval Subsystems thus require a large amount of built-in expertise. They also, of course, require a large amount of good old-fashioned device management and presentation ability, this last somewhat complicated by the design decision to distribute CODER over several computer systems, and the unfortunate reality that these systems make use of different interface devices. These two requirements inform the architecture adopted.

Basically, the user interface to a CODER blackboard community consists of an **Interface Manager** and a set of **translation experts**. The Interface Manager is a hardware-specific module that maps logical "display areas" to and from physical representations such as windows, screens or voices. Translation experts moderate between these logical areas and the local blackboard. The Retrieval Subsystem user interface, for instance, consists of an Interface Manager together with three translation experts, one translating the internal representations of the Pending Hypothesis Area into representations comprehensible to humans and the other two monitoring the user's actions and translating them to hypotheses posted to the blackboard.

This architecture leaves a great deal of room for experimentation. Different styles of query input can be investigated, for instance, ranging from Boolean to extended Boolean to term-vector to natural language. In each case, only the query understanding expert need be changed; the other experts and the Interface Manager are unaffected. Similarly, the range and subtlety of user monitoring can be changed without effecting any module except the responsible expert. Finally, the system can be adapted to an electronic mail front-end by a local adaption of the Interface

Manager, in which case the dialog becomes very slow, but the operation of the remainder of the system does not change. The Analysis Subsystem Interface is structured in a similar way, with display areas for document text, unknown or suspect words discovered, and interpretations formed. Wherever possible, the interfaces are structured for the ease of the user. CODER is, after all, designed to be an experimental system. Sophisticated user interfaces are thus required to provide as much information as possible during incremental development, and to keep data collection on system versions, if not a pleasant, at least not an excessively painful process.

5. Progress to Date

Early work on the CODER system has concentrated on design and on preparation of the knowledge to be loaded into the external knowledge bases. Design of the overall system structure, the individual object types, and the knowledge representation formalisms has occupied the second half of 1985 and first few months of 1986. Overall design was completed and implementation begun in early 1986.

Concurrently with this phase, a large experimental test base of documents has been assembled from the AIList moderated digest. A collection of the first three full years, comprising over 3000 messages and about 1,000,000 words of text, has been prepared. The collection has been brought up under the SMART retrieval system and comparative runs made. The SMART implementation of the collection has also been made available at Virginia Tech as a public service, and logs kept of the queries submitted. It is hoped that these logs will provide sample queries that can be used to test versions of CODER.

Any language understanding system requires a large base of knowledge about words in the language upon which to draw, and the text parsers required in the analysis phase of CODER are no exception. To prepare such a knowledge base for the CODER lexicon, two large English-language dictionaries are being analyzed, and the knowledge implicit in the dictionary text translated into Prolog-syntax predicates. The first of these, the (roughly) 85,000-word **Collins English Dictionary**, has been converted; work on the second, the **Oxford Advanced Learner's Dictionary of the English Language**, is currently underway.

Implementation of the CODER system is beginning with the system resources: the communications and knowledge representation managers. Concurrently, work has begun on the generic modules: the generic blackboard and blackboard strategist and the external knowledge base class. The user interfaces have been prototyped and are currently being extended to full functionality. Implementation of experts will be saved until last.

Acknowledgements

Numerous students in the last two years have played a role in the development of CODER. Muriel Kranowski studied hundreds of messages to identify message types and rules for recognition. Lee Hite and Mark Tischler have developed user interfaces using SUN Windows and CURSES software packages, respectively. Sachit Apte has explored problems with distributed processing. Joshua Mindel prepared a message pre-parser to support the indexing and search possible with SMART, which Sharat Sharan has kept up-to-date. Robert Wohlwend has effected the construction of the Prolog lexicon from the typesetting tapes of the major dictionaries. Pat Cooper and Joy Weiss have provided secretarial support.

The Oxford Text Archive and Melbourne University Dept. of Computer Science have supplied tapes with dictionaries and Prolog interpreters, respectively.

CODER

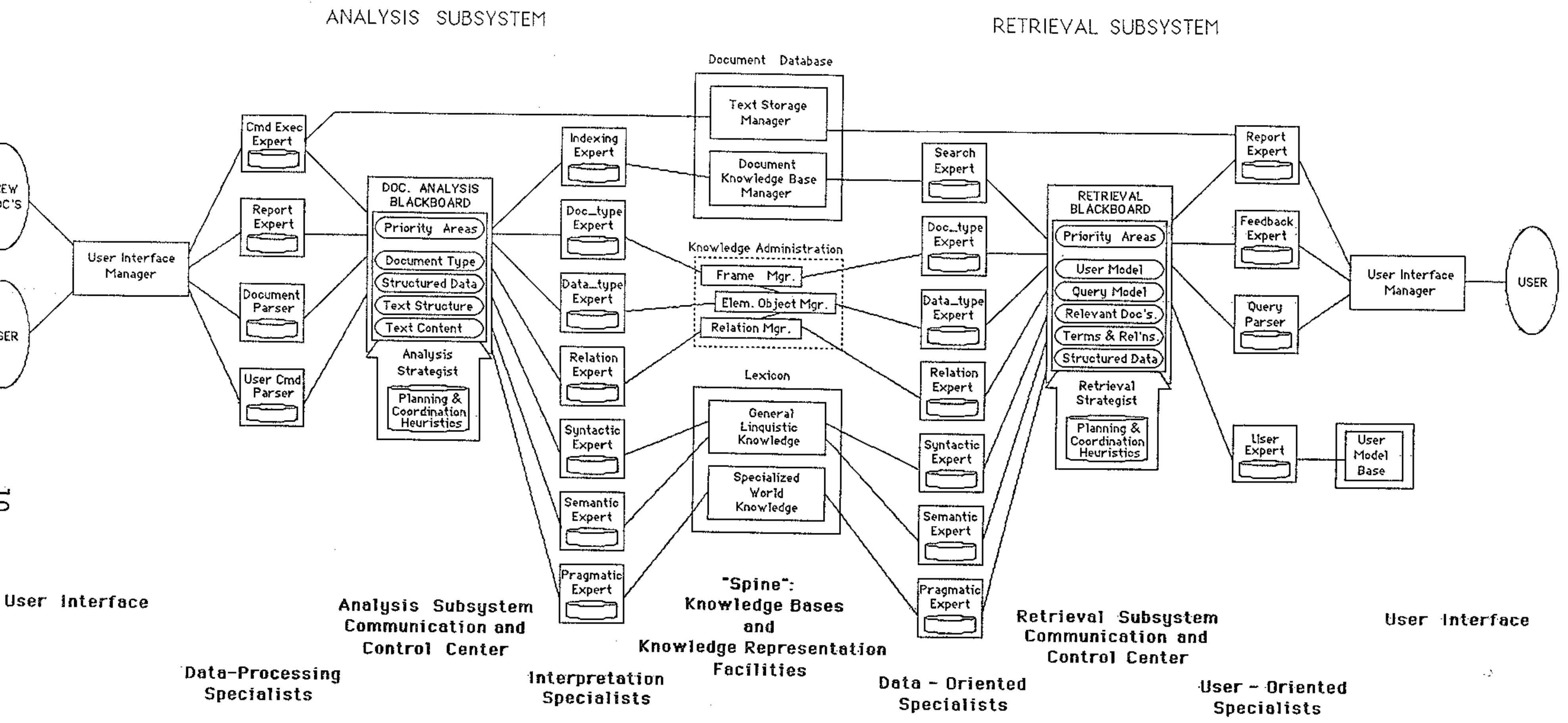


Figure 1

Figure 2: CODER Object Types

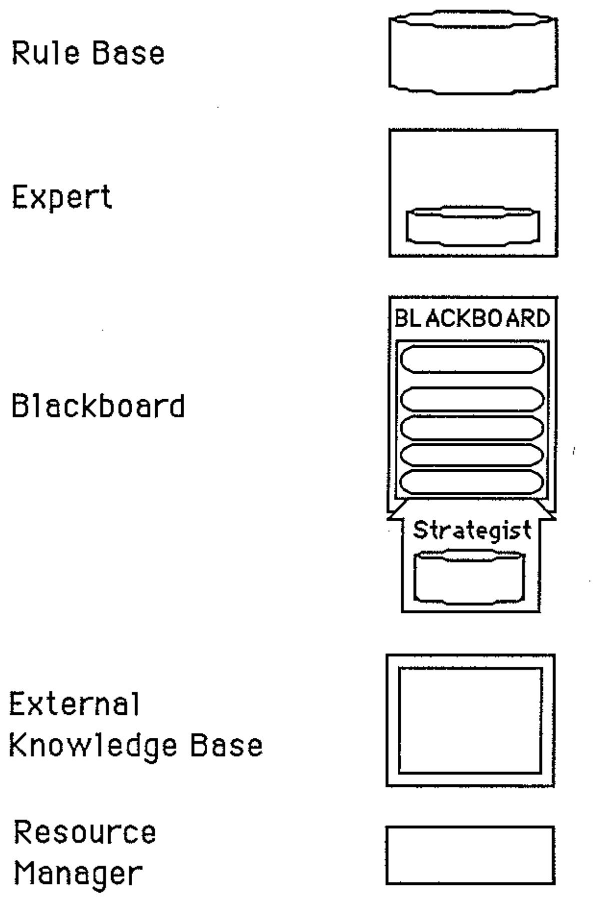
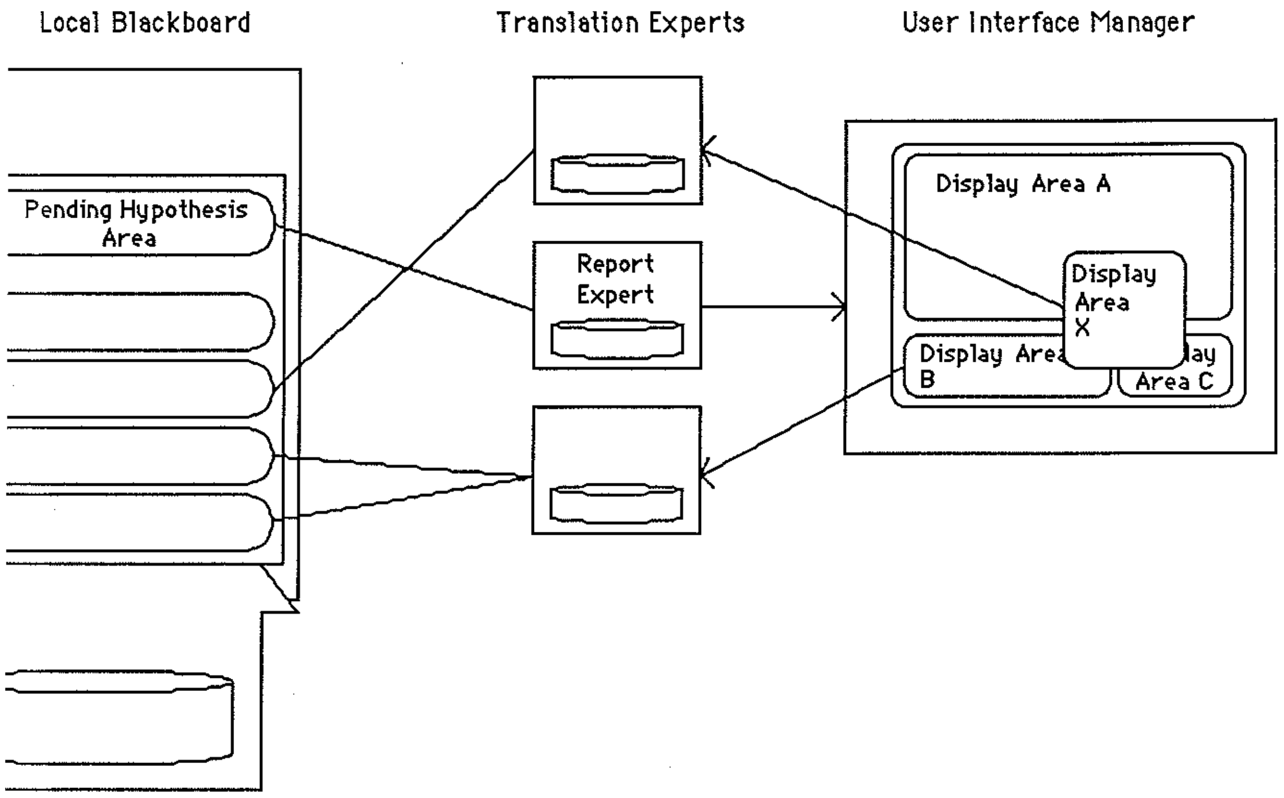


Figure 3: User Interface Subcommunity



References

[BELK 84] Belkin, N.J., R.D. Hennings, and T. Seeger. Simulation of a Distributed Expert-Based Information Provision Mechanism. In *Inf. Tech.: Res. Dev. Applications.* 3(3): 122-141, 1984.

[CHAN 85] Chandrasekaran, B. Generic Tasks in Knowledge-Based Reasoning: Characterizing and Designing Expert Systems at the "Right" Level of Abstraction. *The Second Conference on Artificial Intelligence Applications: the Engineering of Knowledge-Based Systems (Miami Beach, FL, Dec. 11-13, 1985):* IEEE, 1985, pp. 294-300.

[ERMA 80] Erman, L.D., F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Comp. Surveys*, 12:213-253, 1980.

[FOXE 83] Fox, E.A. Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types. Dissertation, Cornell University, University Microfilms Int., Ann Arbor MI, Aug. 1983.

[HAYE 85] Hayes-Roth, Barbara. A Blackboard Architecture for Control. *Artificial Intelligence* **26** (1985), pp. 251-321.

[LEFF 84] Leffler, Samuel J., Robert S. Fabry, and William N. Joy. A 4.2BSD Interprocess Communication Primer. In *ULTRIX-32 Supplementary Documents,* Vol. III, 3-5 - 3-28, 1984.

[LEVE 84] Levesque, Hector J. A Fundamental Tradeoff in Knowledge Representation and Reasoning. *Proceedings CSCSI-84 (London, ON, May 1984):* pp. 141-152.

[SALT 83] Salton, G., E.A. Fox, and H. Wu. Extended Boolean Information Retrieval. *Commun. ACM,* 26(11):1022-1036, Nov. 1983.

[STRO 85] Stroustrup, Bjarne. (1985). *The C++ Programming Language.* Reading, MA: Addison-Wesley.

[WINE 85] Winett, Sheila and E.A. Fox. Using Information Retrieval Techniques in an Expert System. In *Proc. Second Int. Conf. on Artificial Intelligence Applications,* Dec. 11-13, 1985.