An Artificial Intelligence Environment for Information Retrieval Research

Robert K. France
Edward A. Fox

TR 88-10

An Artificial Intelligence Environment for Information Retrieval Research

by

Robert K. France

Committee Chairman: Edward A. Fox
Computer Science

ABSTRACT

The CODER (COmposite Document Expert/Extended/Effective Retrieval) project is a multi-year effort to investigate how best to apply artificial intelligence methods to increase the effectiveness of information retrieval systems. Particular attention is being given to analysis and representation of heterogeneous documents, such as electronic mail digests or messages, which vary widely in style, length, topic, and structure. In order to ensure system adaptability and to allow reconfiguration for controlled experimentation, the project has been designed as a **moderated expert system**. This thesis covers the design problems involved in providing a unified architecture and knowledge representation scheme for such a system, and the solutions chosen for CODER. An overall **object-oriented environment** is constructed using a set of message-passing primitives based on a modified Prolog call paradigm. Within this environment is embedded the skeleton of a flexible expert system, where task decomposition is performed in a knowledge-oriented fashion and where subtask managers are implemented as members of a **community of experts**. A three-level **knowledge representation** formalism of elementary data types, frames, and relations is provided, and can be used to construct knowledge structures such as terms, meaning structures, and document interpretations. 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, defined functionally and implemented under UNIX™ using sockets and the TCP/IP protocol for communication. 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++.

Computing Reviews Categories and Subject Descriptors:

D.2.6 **[Software]** Software Engineering – *programming environments*
H.3.1 **[Information Systems]** Information Storage and Retrieval – *content analysis and indexing*
H.3.3 **[Information Systems]** Information Storage and Retrieval – *information search and retrieval*
H.3.4 **[Information Systems]** Information Storage and Retrieval – *systems and software*
I.2.1 **[Computing Methodologies]** Artificial Intelligence – *applications and expert systems*
I.2.4 **[Computing Methodologies]** Artificial Intelligence – *knowledge representation formalisms and methods*
I.2.7 **[Computing Methodologies]** Artificial Intelligence – *natural language processing*
I.2.8 **[Computing Methodologies]** Artificial Intelligence – *problem solving, control methods and search*
K.6.3 **[Computing Milieux]** Management of Computing and Information – *software management.*

General Terms:

Design, Representation, Architecture.

Additional Keywords and Phrases:

abstract data type, blackboard, composite document, distributed expert system, frame, interpretation, knowledge base, message passing, natural language processing, object-oriented design, Prolog, relational lexicon, network, test collection, user interface.

Acknowledgements

The efforts described in this paper were built on on the work of many people, and would not have been possible only a few years ago. Important tools such as MU-Prolog, implemented at and supplied by Melbourne University, and the TCP/IP protocol, implemented for different machines by DEC, Sun Microsystems and AT&T, have been crucial in the design and implementation of the CODER environment. Also, the knowledge bases on which CODER depends could not be constructed without the aid of other institutions. Oxford University has performed an incalculable service, not only to this project but to the computational linguistics community in general, in establishing and maintaining the Oxford Text Archive. Dictionary tapes from the Archive provide much of the word knowledge from which the CODER Lexicon is constructed. Ken Laws, moderator of the ARPAnet *AIList Digest*, has been extremely cooperative in helping us maintain a complete backfile of the *Digest* for a test collection, and, of course, has furthered the entire enterprise of artificial intelligence research in this country through the excellent guidance he has given the service. Finally, Bruce Smith of the Microelectronics Center of North Carolina, who has been collecting and evaluating implementations of the Prolog language for several years now, provided important assistance in our choice of MU-Prolog as primary implementation language for the project. The project itself was funded in part by grants from the National Science Foundation (IST-8418877) and the Virginia Center for Information Technology (INF-85-016).

Equally crucial to this project has been the infrastructure at Virginia Tech. The Tech Interlibrary Loan Office, particularly Lyn Duncan, has worked long and hard to supplement Tech's collection with records of other research valuable to our design. The administrative workers at the Computer Science Department have every one helped with one aspect or another of keeping me in school and the project moving along on schedule. In particular, Joy Weiss provided both secretarial and -- equally necessary -- moral support for this research as project secretary to CODER.

Numerous other 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. Joshua Mindel prepared a message pre-parser to support the indexing and search possible with SMART, which Sharat Sharan has kept up-to-date. Sachit Apte and Marie-Lise Lasoen worked closely to help design the communication interfaces within the system. Chen Qi-Fang had many useful comments on the design of the strategist; Mahasweta Sen provided essential feedback during the implementation of the blackboard. Most especially, however, I must mention Bob Wohlwend. Besides the 99% perspiration required to convert the OTA dictionaries to a form sensible with the Prolog interpreter, Bob provided much more than one percent of the inspiration for the current structure of the Lexicon and the CODER system in general. Genial, hard-working and steady, Bob has been a much preferred co-worker.

Professors John Roach and David Miller have given useful discussion and criticism to both the work described herein and the document itself. They have both helped shape my thoughts on pertinent subjects; Dr. Roach on natural langauge and reasoning and Dr. Miller on planning. Dr. J.J. Martin also helped form the organizing principles underlying this project, not least by introducing me to the works and ideas of Dana Scott.

Several computer scientists from the larger computer science community have been vital to the design of CODER. Besides Dr. Scott, Joseph Goguen was important in informing the abstract perspective from which the system was designed. The knowledge representation and manipulation aspects of the system owe much to Ronald Brachman and Hector Levesque, as well as to such seminal thinkers as Marvin Minsky and Daniel Bobrow. Dr. Bobrow has also helped inform my understanding of the need for multiparadigm environments. Mark Stefik and Leon Sterling provided mutually reinforcing analyses of the task of automated problem solving that became important, curiously enough after most of the system had been designed, in putting its organizing principles on a sound foundation. Bernard Witt developed the discipline of communicating modules used in the pragmatics of system organization. Finally, I owe a special debt of gratitude to Fred Brooks. Besides coining the phrase "gratuitous complexity" and the dictum "Representation is the essence of programming," Dr. Brooks is responsible for my understanding of the essential role of software engineering to computer science. Much of the techniques of problem analysis and system construction used in this project I learned under his auspices.

None of this work, either my own or that of the gentlemen mentioned above, of course, would have been possible without the great thinkers of the past. The debt that artificial intelligence owes to such thinkers as Aristotle and Leibniz would be too great to be mentioned were it not often ignored. Their advances in the understanding of thought has, perhaps, relatively little direct impact on blackboard architectures and ATN parsers, but it is safe to say that the approach taken to the understanding and use of knowledge in CODER would have been very different without the work of these gentlemen on classification and representation. More immediately, I want to mention Gottlob Frege and Ludwig Wittgenstein for their elucidations of the nature of knowledge, C.S. Pierce for his work on the types and functions of inference, and Alfred Tarski for his contributions to formal semantics. It is safe to say that knoweldge-based systems cannot reach maturity without a powerful and consistent theory of knowledge, and any future theory of knowledge will depend, implitly or explicitly, on the work of such thinkers as these.

Returning to the present, I must thank Dr. Edward Fox, my advisor and committee chairman. It is Dr. Fox who originated the idea of CODER, as well as many of the specifics of what capabilities it should display and what knowledge it should manifest. The work described in this thesis can, in fact, be best described as a process of taking Dr. Fox' vision of an intelligent system for information retrieval and changing it into an implementable structure. In this process, Dr. Fox has been both an invaluable guide and a conscientious overseer. He has provided both the motivation and the opportunity for whatever there is of value in this work; the faults I accept as my own.

Finally, I thank my wife Ruth Doan and my son Jaime for their support throughout a process that kept me away from them far more than any of us would desire. It can not be said that they never complained, but their complaints were always such as to make me feel more loved.

Table of Contents

List of Figures

1. Motivation

Computers come and go, but data go on forever.

-- N. Bruce Berra

As the world's pool of information, and particularly of machine-readable information, continues to grow, it becomes increasingly necessary to engage the help of computers to control and manipulate it. Early attempts at computer-aided text storage and retrieval, however, have focused principally on performance and have achieved only moderate levels of effectiveness. The CODER (**CO**mposite **D**ocument **E**xpert/Extended/Effective **R**etrieval) project aims at constructing 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 philosophy and of its specific functional modules. The construct of a *moderated expert system* has been instrumental in achieving both of these design goals.

1.1. Problem Description

There is little need to dramatize the so-called 'information explosion' to any member of the academic or business community. The proliferation of information-control services such as BRS and DIALOG, the focus of both Japanese and American research initiatives on providing hardware and software for the 'upcoming information society,' the current deepening crisis in library techniques and resources; even the existence of such catch phrases as 'information explosion' and 'information society' point clearly to the increasing importance of information as a commodity of the modern world. It is worthwhile, however, to highlight three aspects of the problem. First and most obviously, we are currently witnessing a geometric (or supergeometric) growth in *amount* of information, and particularly on-line information. Second, we are witnessing a change in *kind*, in which more and more on-line information is in the form of undigested text. The advent of the word processor and electronic publishing are in part responsible for this change; the development of CD-ROM as an information-dissemination tool [FOXE 86b] will only add to this effect. In part, however, this change in kind from database to document base arises from the third aspect of the information explosion: the social change in our *attitudes* toward information. It is not our place here to trace the causes behind modern society coming to regard information as an important commodity; we only note that in the business no less than the academic community, possession of information is considered to be as important as the possession of tools.

Particular attention is being given in the design of CODER 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*, a fitting but challenging collection of composite documents.

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 83c]. 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. The model of a relational lexicon is particularly appealing since automatic processing of machine-readable dictionaries can 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] and vice versa (see § 1.5). It became clear that a rule-based approach allows systems to be tailored to (classes of) users, and that several search algorithms can be combined dynamically to give instantaneous rather than "batch" type of feedback with superior performance. 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 in several classical IS&R systems, relatively simple document analysis has been 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, should allow some questions to be more effectively handled, and others to be answered for the first time.

1.2. Prior Work - Information Storage and Retrieval

Though visionaries have described their hopes for intelligent information retrieval systems since at least the mid-1940's [BUSH 45], that goal has still not been reached. Many preliminary steps have been taken, though, and by integrating the results of some of the most productive efforts, it is hoped that significant progress can be made during the 1980's and beyond.

Traditional information retrieval systems can be divided into *data* retrieval systems, which capitalize heavily on the constraints and internal structures of such entities as employee records and account transactions, and *document* retrieval systems, which work with natural language documents such as mail messages or literature abstracts using algorithms informed by the syntactic and occasionally the semantic structure of words in text. The most successful approach to the former problem has come through the theory of relational databases (see, e.g., [DATE 86]). Thus far, an equally successful theory has yet to be provided for the latter. The most widely used representaiton for text documents remains the vector of terms (or stemmed terms); the two most widely used means of matching documents to user needs are by Boolean combination or vector distance.

Several extensions to this model have proven fruitful. Approximate matching algorithms [HALL 80] can help avoid spelling errors or other errors caused by lapses in memory or lack of awareness of an author's means of expression. Sophisticated access methods aim at employing special data structures to save space and/or processing time

[FALO 85]. The use of feedback information to help in the construction of improved queries was demonstrated by [ROCC 71], developed further in terms of probabilistic estimation by [ROBE 76], and explained more completely in [VANR 79] and [SALT 83a]. It was applied to large collections through an intelligent front-end system as well [MORR 83]. Clustering has been considered numerous times; the most recent thorough study is [VOOR 85].

Fuzzy set theory has been used to build a number of IS&R models. Bookstein suggested including the ability to consider user-supplied term weights [BOOK 80]. Paice carried out some small-scale experiments to demonstrate the value of "soft" Boolean evaluation [PAIC 84]. The p-norm model, generalizing both of these approaches, was explained and validated [SALT 83c], applied to automatic query construction [SALT 83b], and adapted for (extended) Boolean feedback [SALT 85]. A recent study explored the p-norm model further and found it more effective than that proposed by Paice [FOXE 86a].

Further improvements have been sought by utilizing other information besides terms. The value of bibliographic information was established at an early date [SALT 63]. Bibliographic coupling [KESS 63], citations [GARF 78], and cocitations [SMAL 73] were all shown to help determine how closely pairs of documents relate. Relevance feedback techniques were developed to incorporate some of the bibliographic data [MICH 71] in searches. Since the results of different searches carried out for a single query tend to have small overlap [KATZ 82], it seems wise to combine a variety of types of information. Bichteler and Eaton found this to be of value with bibliographic coupling and cocitations [BICH 80]; a mix of those and other information does give demonstrable improvements [FOXE 83a]. Indeed, part of the appeal of using knowledge-based and plan-based retrieval in CODER is to better handle combining the wide variety of available information that describes most documents.

1.3. Prior Work - Computational Linguistics

For more than a dozen years, work has progressed on the application of linguistic insights to the development of information retrieval systems [SPAR 73]. Wendy Lehnert has studied the variety of possible question types that people construct and how they should be answered [LEHN 78]. Oddy [ODDY 77] viewed the retrieval problem as a dialog, stressing the human-computer interaction. To better understand users and their behavior, a small amount of psychological research has been conducted; much further study is required [BORG 84]. Most of the linguistic effort, however, relates to document analysis. In the retrieval community, attention has been given to the use of discourse analysis methods to aid in identification of "answer-passages" for passage retrieval [OCON 80]. Discourse analysis has also been of value in the TOPIC system, which uses word expert parsing [RIEG 81] to summarize text into a hierarchical condensation. Research relating to the Linguistic String Project has focused on applying a powerful parser to a given sublanguage, such as that found in medical reports [SAGE 75].

Parsing unrestrained text, however, presents a number of problems to conventional parsers. Charniak points out the importance of understanding the context of a given sentence [CHAR 82], and advances the idea that context is crucial for integrating syntax and semantics [CHAR 83]. Indeed, the FRUMP system [DEJO 82] could not function without having stored *scripts* to match against in order to establish the proper context(s). Schank et al. have used scripts for a variety of natural language analysis tasks [SCHA 77], including a conceptual approach to retrieval [SCHA 81]. Simmons [SIMM 84] uses a similar construct, the *schema*, to aid in analysis, building of a representation, and possible later translation. Wilensky et al. [WILE 84] use a phrasal approach to parsing, and have developed support for dialog, translation, and accessing a knowledge base on UNIX™ use.

Despite, or perhaps because of, this wide range of approaches and prototype systems, practical natural language parsing has made slow progress over the last twenty years. Practical parsing requires a vast lexicon, parsers capable of handling many different syntactic patterns, and the ability to fail gracefully, all characteristics rarely found in research systems. Progress has been appreciable, however [SLOC 85], and success rates of 80-90% in machine translation contexts, for instance, are not unusual. Encouraging results, moreover, have recently been published both in the theoretical analysis of natural language [PERR 84, GAZD 85b] and in the application of expert, knowledge-based systems technology to natural language parsing. The theoretical results provide evidence of computationally tractable classes of formal languages (e.g., those based on head grammars and indexed grammars) that may be sufficient to embed natural languages; the expert systems approach provides a "phenomenologically plausible" means for combining the different types of knowledge required in natural language understanding.

One method for applying expert systems technology to parsing involves the use of *word experts* [RIEG 81, HAHN 84a], where each word in a section of text triggers a separate knowledge source with expertise on the relations between possible contexts and possible meanings for the word. This approach has actually been used in the TOPIC system, with real success, for parsing documents in the domain of information technology, but it requires a substantial amount of effort in constructing the word experts. A related approach [METZ 85], makes use of individual experts for different syntactic patterns, encoding words by the patterns in which they can participate. Other approaches (e.g. [CULL 84] and [WALT 84]) divide the problem of parsing conceptually, assigning an expert to each type of knowledge involved in the task. Common knowledge sources involved in such a deconstruction include experts in syntax, semantics, context, and world knowledge.

Approaching the problems of parsing from a knowledge-based perspective requires a source for knowledge, particularly knowledge about words. Evens and Smith described a comprehensive lexicon of word knowledge for the support of natural language processing and question answering in [EVEN 79]. Large lexicons have proven crucial in the Linguistic String Project natural language system [WHIT 83] and in PHRAN [WILE 80]. A relational lexicon has also been found useful for expansion of natural language queries to retrieval systems such as SMART [FOXE 80, 83b]. Consequently, there has been a great deal of interest in the computational linguistics community in lexicon construction. Two basic approaches are discernible in this work: production of lexicons from machine-readable dictionaries and from bodies of target text.

The basic principle in the analysis of target text involves imposing some sort of patterns on the text and reading off relations among the words in the text based on those patterns. For instance, Burghard Rieger [RIEG 84] analysed the statistical distribution of words in text based on their co-occurrence frequency, then clustered them into dependency trees based on a pre-defined metric space. Jerry Hobbs [HOBB 84] went farther, and used syntactic and semantic patterns to derive word relationships from selected medical text passages before again applying a clustering algorithm to derive semantic relationships. Finally, a set of extremely sophisticated patterns were applied in the RINA system [ZERN 85] to locate unknown figurative phrases in text and attempt to fit them, through context, into an existing lexicon. The results produced in this sort of conceptual bootstrapping are encouraging, but it must be noted that they are more encouraging the more information is already available in the lexicon to start with.

Work on deriving computational lexicons from text dictionaries has only been possible through the generosity of dictionary publishers. Merriam-Webster, in previous years, allowed researchers to format the *Seventh Collegiate* [SHER 74]. Amsler studied their *Pocket Dictionary* and discovered a "tangled hierarchy" of word relationships [AMSL 80]. Since that time there have been other dictionary studies as well [PETE 82, AMSL 84]. A great deal of analysis, however, is needed to build a lexicon from a dictionary. Peterson and Amsler were able to automatically extract syntactic information from the Merriam-Webster tapes; collecting semantic information, on the other hand, was only possible with the aid of quantities of human labor. Nor is it clear just how much implicit semantic knowledge is available in a dictionary to be extracted. Some indications of possible semantic relations have been described by the Sedelows [SEDE 85a, SEDE 85b] and by Martha Evens [EVEN 82]. Thomas Ahlswede has studied adjective definitions [AHLS 83] and has been developing a tool kit for manual and/or automatic construction of a relational lexicon [AHLS 85]. Chodorow, Byrd, and Heidorn [CHOD 85] have automatically extracted semantic hierarchies from large dictionaries, relating some 40,000 nouns and 8,000 verbs in short, bushy trees. There is reason to believe, however [MILL 85], that this work only scratches the surface.

1.4. Prior Work - Artifical Intelligence

The hallmark AI systems of the past have typically been the work of a single mind, if not always a single implementor. Attempts to extend such systems to cover realistically large domains, however, have generally run afoul of the AI maxim 'the solution to the small problem is generally not the solution to the large problem' (see, for example [LENA 84]).† This effect has resulted in much disappointment, both within the field and from sources outside it. Consequently, considerable attention has recently been placed on

†Part of this scale-up problem, of course, is algorithmic: procedures which work well in sub-problems rarely work well in the more general case. The intention in CODER is to minimize this effect through a knowledge-intensive approach to system development: algorithms and inference mechanisms are kept purposefully simple and general, the complexity of the problem domain being expressed instead in large amounts of factual knowledge. Thus the natural language recognizers under construction are in themselves very simple engines, but are allied to a huge lexicon of knowledge about individual words and phrases in the English language. The more important aspect of the problem, however, remains the engineering of AI systems (again, see [LENA 84] and its motivator, [RITC 84]).

defining models that will allow 'production-scale' AI systems to be built.

One promising suggestion is the 'society of mind' model [MINS 77], that pictures intelligent activity as the result of a consensus among competing functional units, each of which expresses a different facet of intelligent behavior. This concept can be dated to Oliver Selfridge's PANDEMONIUM proposal [SELF 59] and is still advocated by cognitive scientists and computer scientists alike. J.F. Sowa, for instance, uses a related model to describe conceptual processing in [SOWA 84]. The metaphor is attractive both as an engineering methodology, where it provides a criterion for system decomposition, and introspectively, where it corresponds well to many of our own experiences of how we think. The main problem in expressing this model in programs has been providing the elements in the society with some sort of organizing control, so that literal pandemonium does not ensue.

A key breakthough in resolving this problem came in the late 1970's with the HEARSAY-II speech understanding system [ERMA 80]. HEARSAY approached its task through a community of *knowledge sources* communicating through a shared data area called a *blackboard*. Knowledge sources could post both hypotheses about the problem under consideration at each run and new tasks motivated during their activation. Processing and control were thus equably distributed across the community. The blackboard itself was so structured, however, as to ensure that processing always proceded in a hierarchical manner, with knowledge sources at each logical level triggering processing at the next higher level until the sentences recognized from the input speech were posted to the highest level of the blackboard. This hierarchical structure was a key factor in the system's admirable performance, as subsequent attempts to generalize the blackboard structure revealed [BALZ 80, ERMA 81]. Thus, while the blackboard continues to be a common device for communication in AI systems, its use in control has declined.†

A more effective method of control involves designating a particular module to control the *focus of attention* of the community (see, e.g., [LESS 80] or [RYCH 84]). This module can then combine tasks suggested by the knowledge sources with some sort of control knowledge or planning algorithm [ERMA 84, NECH 84]. This model fits

†Exceptions to this rule, such as Barbara Hayes-Roth's work [HAYE 84, HAYE 85a], are admitted to be excessively slow and cumbersome for production-quality systems. [CORK 82] reported some success in maintaining focus of attention through classifying knowledge sources as either goal-directed or data-directed, but little follow-up work appears to have been done on this approach.

neatly into the theory of hierarchical planning proposed by Mark Stefik [STEF 81], with the focus of attention corresponding to the strategy space and the tactical methods used by the individual knowledge sources corresponding to the design space. Leon Sterling [STER 84] has elucidated a related model of logical programming where the knowledge involved in problem solving is divided into three levels: a *domain level* of factual knowledge, a *methods level* of tactical knowledge and a *planning level* of strategic knowledge.

The 'society of mind' model is attractive to those involved in building large systems because it provides a method for distributing an artificial intelligence project among many developers. An orthogonal approach to coordinating development teams and to reducing the overhead of building AI systems has been to provide environments for AI 'programming in the large'. The expert systems production tools such as S1 [RYCH 84] and KEE [KEHL 83] fall into this category, as do more general AI tools such as CK-LOG [SRIN 84], LASER [REDD 85] and LOOKS [MIZO 84]. Key questions in design or choice of such an environment include the programming paradigm on which it is built and which others it supports. A logic programming paradigm has been advocated [HELM 85], as has an object-oriented approach [TOKA 84]. Carlo Zaniolo [ZANI 84] advocates combining the two; Daniel Bobrow goes farther, suggesting that AI development requires a wide variety of paradigms [BOBR 85a]. Bobrow's ideas are reflected in the LOOPS environment [BOBR 82], which provides facilities for mixing several different programming styles under an overall object-oriented paradigm.

Several of the environments mentioned above, including CK-LOG and LOOKS, provide some sort of knowledge representation facility. Knowledge representation is an integral part to the construction of knowledge-based systems, and is arguably one of the foundations of AI. The Fifth Generation project has designed a knowledge representation language, Mandala [FURU 84], as one of the key provisions of their artificial intelligence programming environment. Other integrated knowledge representation / programming environments include Prolog/KR [NAKA 84] and KRINE [OGAW 84]. Of course, several stand-alone knowledge representation / knowledge management systems have been constructed, notably KRL [BOBR 77], Krypton [BRAC 83b] and KL-ONE [BRAC 85b]. Greiner [GREI 80] has even proposed a knowledge representation language language to be used in designing such environments; unfortunately, the state of the art in knowedge engineering is not sufficiently stable to make this a viable possibility.

1.5. Comparable Systems

CODER is not alone in the attempt to apply artificial intelligence techniques to the problem of text information systems. The problem is, in fact, an obvious candidate for an expert systems approach: it is a problem where conventional programming techniques have consistently fallen short of satisfactory solutions, yet where humans produce acceptable performance without appearing to use creative or intuitive problem-solving techniques. Thus, several different projects over the last few years have begun to attempt to define the categories of knowledge and the "rules of thumb" that human experts in document analysis and document retrieval use to serve clients with needs for text-based information.

Such systems generally fall into three different categories, depending on the focus of their application of expertise. First, there are systems that attempt to analyze documents for the purpose of determining their content. These systems are generally not concerned with the problems of document retrieval and presentation so much as with automatically producing abstracts or summaries of input documents. Pioneeering work by Wendy Lehnert and Michael Dyer at Yale [LEHN 81, DYER 83] used an analysis of narrative structure to produce summaries of stories in a highly restricted universe. Successful summaries have also been produced from text drawn from the syntactically restricted world of newspaper articles [DEJO 82, DECK 85] and semantically limited domains such as medical writing [SAGE 75, OBER 85] and scientific texts [GOME 85]. Larger domains are addressed by the EPISTLE system [MILL 80, MCCO 84], which attempts to summarize electronic mail messages in an office environment, and TOPIC [HAHN 84b, HAHN 85] which produces abstracts of arbitrary documents in the field of information processing.

Some of these abstracting systems map documents into sets of key words or key phrases (see [PAIC 81] for a discussion of candidate techniques for this process). Others, including EPISTLE and TOPIC, use specialized languages or logics for capturing document content. This representational ability is the second area in which the knowledge-based approach has an impact on the process of document retrieval. Current expert retrieval systems run the gamut from systems like I^3R, where a sophisticated analysis of user needs is coupled to a flat document representation as vectors of stemmed terms, to systems like RESEDA [ZARR 81, ZARR 84], where highly structured knowledge permits efficient temporal and linguistic inference in response to user queries. Unfortunately, RESEDA's knowledge base is so sophisticated that it cannot be constructed automatically, making it unsuitable in CODER's intended domains of application. More suitable are the systems employed in RUBRIC [TONG 85, MCCU 85] and ARGON [PATE 84a]. ARGON, like RESEDA, uses a hand-constructed database in its current implementation, but provides a frame-based knowledge representation structure using the KANDOR system [PATE 84b] that comes to grips with many of the problems involved in representing natural language. RUBRIC, which works in a large document-collection environment similar to the CODER domain, deals with text through a flexible concept hierarchy built above words and word phrases. In this way, it is able both to identify word (strings) in text as exemplifying concepts and navigate among concepts in an attempt to satisfy users' information needs.

The most sophisticated part of RUBRIC, however, falls in the third area for the applying of expert-systems technology to information retrieval: the process of understanding and fulfilling a user's information needs. Query construction in RUBRIC is viewed as building a rule base that captures domain knowledge relating to a given question, so that the job of a user stating an information need is much akin to that of a knowledge engineer formulating rules in a sub-domain of special interest [TONG 83]. This approach, which has the advantage that domain knowledge can be gathered as needed at retrieval time, instead of having to be engineered into the system before it is completed, is also followed by Gautam Biswas' group in Columbia [BISW 85, BISW 86]. Several other expert systems have concentrated on this end of the problem, including the early CONIT system, where expertise was applied to help users connect to suitable commercial databases and carry out searches using a common command language [YIPM 79, MARC 83]. Karen Sparck Jones has performed much valuable research on using linguistic knowledge to expand the terms of a user's query in order to match semantially related document terms [SPAR 84a, SPAR 84b]. Other researchers (e.g. [COYL 85], [DESA 85] and [FIDE 86]) attempt to capture the behavior of trained search intermediaries in expert systems. Several of these techniques are combined in the I^3R system, where expertise is encoded in a set of cooperating knowledge sources on search methods (vector, probabilistic, clustered or other), domain concepts, and user characteristics [THOM 85, CROF 86].

While CODER has similarities to several of these systems, it is unique in that it attempts a unified attack on all phases of the problem, from document analysis to user interface management. By adopting a coherent overall picture, where analysis, representation and retrieval are all driven by a single theory of document structure and content, CODER is expected to provide high levels of recall and precision and a subtle matching of user needs to passages retrieved. Further, CODER differs from the majority of the systems described in this section in that it is constructed as a *research testbed,* where many theories of content and form can be tested experimentally. Therefore, the thrust of the explorations described in this document has been to design a unified environment within which a broad range of different techniques and theories may be applied to assess the impact of expert systems / knowledge-based techniques on the entire process of information storage and retrieval.

2. Design

If we are doing something that we understand weakly, we cannot hope for good results. And language ... is still rather weakly understood.

-- Kimmo Kettunen, 1986

The purpose of the CODER project is to provide an experimental testbed for investigating the use of artificial intelligence techniques in the storage and retrieval of composite documents. The system is designed to allow a variety of techniques from different branches of AI to be applied to various aspects of the task of analysis, indexing, and retrieval of documents. It is hoped that the system will be of use for a wide range of experiments, as the SMART system has been over the last decades, and that it will have the flexibility and ruggedness to endure, like SMART, over a considerable lifespan.

In this chapter, we discuss design issues raised by the project mission and some of the decisions that were made to resolve them. In making these decisions, our basic aim has been to keep the environment both powerful and flexible enough to satisfy the evolving demands of an experimental system. The fields of artificial intelligence and information retrieval are both currently undergoing a rapid process of change. If CODER is to serve its purpose as a comparative system, it is important that it be able to adapt to such changes as they occur.

2.1 Knowledge Engineering

In designing a knowledge-based system such as CODER we must deal with at least four issues regarding the knowledge on which it is based. First, we must decide how the knowledge is to be encoded: that is to say, the *represention* of system knowledge. Second, we must deal with how it is to be managed within the system: with issues involving the *maintenance* of knowledge. Then again, we must plan where system knowledge is to come from: we must consider the *acquisition* of knowledge. Finally, we must consider the use to which the knowledge is to be put: the *inference methods* that the knowledge must support. In all these areas, we must allow as broad an area of experimentation as possible to the CODER user within the constraints of system consistency.

While knowledge representation techniques in AI have rightly been described as "as yet non-convergent" [BRAC 86, p. xiii], there are several schemes that are recognized as showing promise of broad applicability. These include schemes based on *frames*, on *semantic* (or *associative) networks,* on first-order *logic* (or some restriction of it such as Horn clauses), and on *production rules.* In addition, there is a widely recognized need for a system of representation for *procedural* knowledge beyond that provided by underhanded use of one of the declarative formalisms or by arbitrary Lisp PROG statements. It has been argued (e.g., by [CLAN 83]), that production rules play several different roles, allowing both declarative and procedural interpretations at different times. Indeed, none of the existing representation systems is free from semantic ambiguity† and all have been used in various systems to model different things. This lack of clear definition has generally been recognized by the system designers, and has often been used as a justification for adding "escape clauses" whereby arbitrary functionality could be added to the system by its users: a refreshing contrast to this trend is provided by the recent KANDOR system [PATE 84b, PIGM 84].

Such clarity would be admirable in CODER as well if it could be achieved without compromising the generality of the knowledge representation provided. However, there are many types of knowledge for which representation issues cannot be separated from the inference methods used. As design of inference mechanisms is a specialized process that will take place, probably many times, during the experimental use of the system, there is no way to predict exactly what the demands of the inference engines used will be. This

†See, for instance, [WOOD 75] and [BRAC 83a] for semantic networks, [HAYE 79] and [ISRA 81] for frames, and [MOOR 82] for logic. Israel and Brachman [ISRA 81], in fact, attempt a criticism of all three schemes from a model theoretic perspective: their observations on semantics are cogent and clarifying. Further clarification has been provided by Alan Newell [NEWE 81] and Hector Levesque [LEVE 84b], who argue that the functional semantics of knowledge manipulation can and should be kept independent from the particular scheme used in knowledge maintenance. Many issues, however, must still be resolved before either a unified lanaguge for knowledge representation or a consistent semantics for that language can be constructed.

motivates the overall approach to knowledge engineering in CODER. Knowledge is divided into two conceptual types: *factual* or world knowledge, and *expert* knowledge. Factual knowledge is knowledge of entities in the problem world, their attributes and the relations among them. This type of knowledge is controlled tightly in CODER, using a three-level system of elementary data types, frames, and logical relations (see Chapter 3). Expert knowledge is knowledge at a logical level above world knowledge: knowledge of classes of entities, metaknowledge of how to create and manipulate facts, and so forth. It can involve hypothetical, procedural, or rule-based knowledge, it may draw upon the same formalisms provided for factual knowledge, or it may use other forms entirely. No restrictions are placed on the syntax or semantics of expert knowledge, so that individual experimenters can be free to develop their own formalisms as they are needed. Expert knowledge is, however, limited in locality: its use must remain local to the expert(s) developed by the experimenter. For communication among the modules of the system, only the factual knowledge representation is used.

The factual knowledge representation language is also used in defining the knowledge maintenance facilities of the system. These facilities, in concert with systems such as KANDOR and Krypton [BRAC 83b], are defined functionally, in terms of what operations can be performed on objects constructed in accordance with the representation formulas (see § 4.3.3). Following their intended use as repositories for factual knowledge about the problem world, no inference capabilities are included in the operations defining these knowledge bases. Instead, approximate matching operations are provided, so that sets of matching facts or descriptions of entities can be retrieved by incomplete facts or partial descriptions. Inference, either over factual knowledge or over their own local knowledge structures, is properly the domain of the experts, and will be discussed below (§ 2.6 and § 4.3.2).

The distinction between factual knowledge and expertise is also useful in analyzing the strategies used for knowledge acquisition. Expertise, the representation of the broad knowledge and 'rules of thumb' used in performing some particular task, will generally be constructed manually by the experimenter attempting to analyze and automate the task. Factual knowledge, the thousands of tiny facts that flesh out the world, can generally be collected automatically, either by the system itself or by ancillary programs. For instance, knowledge of words for CODER is being redacted automatically from machine-readable dictionaries, while knowledge of the documents in a collection and the users in a community will be accumulated automatically by the system during analysis and retrieval sessions respectively. Expertise on how to recognize documents or how to parse natural language sentences, however, will be encoded by the experimenters responsible for the construction of the responsible experts.

2.2 Document Architecture

One of the representational advances of CODER over classical IS&R systems is its ability to easily handle *composite documents*. CODER represents the information with which it is designed to work neither as relational tuples nor as flat strings of text, but as structured entities composed of fields, each of which can be filled by only certain types of data. These fields may themselves be composed of other fields with more specific restrictions on the types and semantic content of data that may fill them, and so forth. This approach allows different aspects of a document to be represented in content-appropriate ways, rather as is currently done by a human cataloger. Moreover, by being able to recognize the semantic restrictions on a given field of a document, the system is given the opportunity to use specialized parsing techniques or inference methods in analyzing the data in that field, and to use specialized disambiguating and clustering techniques during retrieval.

To accomplish all this, however, the system must provide facilities for defining and manipulating both structures of fields and the different data types that fill the fields. These structures may themselves contain structures (as when a **date** occurs as part of a **bibliographic reference** within the **bibliography** field of a **journal article**) or sets or lists of structures. It must be possible both to navigate within such structures and to specify methods for recognizing and making inferences from the data types that make up their fields. Finally, it must be possible to create and store abstract representations of structures, for instance in indexing an analyzed document, even when they are not fully instantiated. For instance, the system must be able to segment a document into a list of bibliographic references even if not all the references are complete, or to identify it as a report of an event even if the document does not exactly match the template for a **report of event** structure.

Of the various knowledge representation structures provided in the CODER system, composite documents form a subtype of frames. CODER frames are structured descriptions that model objects with typed attributes. In composite documents, the attributes in question are the contents of the document fields. For example, one attribute of an **electronic-mail message** (one slot of the **electronic-mail message** frame) is its date of origin, which is of type **date** (itself a frame with slots for **day, month** and **year**). Thus, a document type is defined by listing the possible attributes that a document of that type can have and the types of its fields.

Of course, knowledge about a type of document is not limited to its prototypical definition. Associated with a given document type may be semantic knowledge (such as the expected content of a field), inter-type knowledge (which document types are also permitted to be -- or are also likely to be -- which other types), and relations among fields (this field is *required* in a document of this type; these two are *mutually exclusive).* This knowledge, however, is perhaps best considered expertise, and can be well modeled in the system through rules managed by a Document Type Expert.

Using frames to represent documents has certain obvious advantages. Since frames may have other frames as slot fillers, it is relatively simple to mirror schemes for hierarchical decomposition of documents, such as ISO-WG3 [HORA 85] or the COBATEF model [PEEL 85]. Text fields, for instance, may be defined as lists of paragraphs; paragraphs as lists of sentences; sentences as lists of text items. Markup structures may be included as separate fields of the document frame and/or the component frames (lists or tables, for example, often require specific layout information to clarify their semantic form). In addition, since a frame instance need not have all its slots filled, it is easy to create representations of a document based on imperfect matches. If a document is interpreted as being similar to an ideal type, a partial instantiation of that type can be formed to represent the document. Those aspects of the document that correspond to the ideal can be used to fill slots in the instantiation and those aspects of the ideal that have no match in the document can be ignored. Any aspects of the document that do not fit the ideal can be either ignored as well or fit to another ideal, thereby creating a separate interpretation of the document.

When a document is modeled by several interpretations, of course, there is a possiblility that the interpretations are inconsistent among themselves. This is not always the case. We can say that a document is a **journal article** and at the same time a **book review** without being inconsistent. Neither is it inconsistent to describe the same document as a **bibliographic reference**, although the bibliographic portion of a book review is usually only a small part of it. Inconsistencies may nevertherless arise, however, when a key text item is interpreted in different ways, or simply when different aspects of the document invoke different ideals. When this occurs, we say that the interpretations formed from the document are *not mutually satisfiable.*† We can, however, find maximal satisfiable subsets of the set of all interpretations. Inclusion in one or more such subsets can then serve as a criterion for a good indexing interpretation; i.e., for an interpretation that can be stored as knowledge describing the document.

2.3 Natural Language Understanding

Probably the most difficult parts of a composite document for any IS&R system to handle are the text fields that make up the bodies of most documents in an information retrieval environment. For CODER, this problem is compounded by the experimental nature of the project. It must be possible during the system's lifespan to configure the system with any of a number of different natural language analyzers, with different theoretical bases and of different levels of sophistication, in order to assess their impact on the information handling process. Of course, it is impossible to create a system that will support equally well all of the multitude of natural language parsers proposed by the computational linguistics community. The CODER system, however, has enough flexibility to work well with any of a wide range of parsers, including but not limited to those based on Augmented Transition Networks [BATE 78, WINO 83], Definite Clause Grammars [PERE 83], Linguistic String Grammars [SAGE 81], and Phrase Structure Grammars [CREA 85]. Each of these paradigms can lead to high quality parsing of natural language text, and in the current state of computational linguistics research, it would be foolhardy in the extreme to commit to only one.

Flexibility in choice of parsers is ensured in two ways: first, through providing the flexible knowledge representation structures detailed above, and second, through divorcing the process of parsing from the remainder of the system. In accordance with principles of

†It should now become apparent that the term *interpretation* was not lightly chosen. What we are doing in describing a document through frame interpretations is building a (set of) theory(s) about the document content. This is the analog in the frame language to the analysis of validity for the language of predicate calculus in classical model theory, and we will adapt the vocabulary of model theory here as an alternative to the more problematic *possible worlds* interpretation of non-monotonic knowledge. Thus we describe the process of document cataloging as one of finding maximal satisfiable sets of representations in the frame language that hold for the document, and we will say that the document is a model of such a set. See Chapter 3 for a more complete discussion.

modularity and information hiding, the expert or experts responsible for parsing incoming text are independent from the rest of the system, communicating only through hypotheses within their areas of expertise. This modular separation keeps the parser from interacting directly with other system experts, for instance with those responsible for determining the type of an incoming document or for choosing appropriate indexing relations. It does not, however, prevent a more subtle interaction at the level of the knowledge structures used to represent text.

In an information storage and retrieval system where the information stored and retrieved involves natural language, it is always necessary to arrive at a canonical representation of the input documents and a (possibly different) representation of the users' queries, so that the users' information needs can then be compared to the documents' information content, and matches or close matches discovered. These canonical representations can be as simple as lists of words or word stems, or they can involve complex logical or semantic relationships. They must, however, be based on a single set of primitives and structuring operations, even when the language in which the users' queries are constructed is different from the language of the input documents.† It is likely, therefore, that any change in the meaning representation structures produced during text analysis will require changes in the retrieval subsystem as well. These changes may either be made in the query parsing module, so that it can produce the same structures produced by the document text parser, or they can take the form of transformations from the structures produced by the query parser to the structures produced by the text analyser, performed during the process of retrieval.

In addition, changes in the representations produced by the text parser may require changes in the experts responsible for abstracting and using indexing knowledge. To the extent that these experts use heuristics linked to the type of knowledge structure produced, they will have to be changed whenever the structures are changed. Thus, the choice of an abstract representation for natural language has far-reaching effects in the system, effects which cannot be easily controlled. The choice of a system for converting text into such an

†This is not as simple to achieve as it sounds. Even if we take it that both the input documents and the queries make use of the same subset of the natural language in which the system is based (which is almost a reasonable assumption, although not quite true in practice), it is a truism of empirical information-retrieval research that neither the individual words nor the linguistic constructs used in forming questions and expressing needs are the same as those used in the expository diction characteristic of the target documents. Thus a query parser may share the same language recognizer as a document analyzer, but will generally require a different set of meaning representation productions.

abstract representation, however, is not constrained by interaction effects: many such systems may be tried with only local changes. And the CODER environment provides sufficient flexibility that the more far-reaching decisions of text representation can at least be approached from an experimental point of view.

Despite this flexibility, choice of a natural language parser is limited in two important ways. First, of course, the parsers are limited in the type of output they can produce. It is assumed that whatever parser is chosen will reduce the language of the input text to some sort of abstract structure, *and that that structure will be representable in the factual representation language described above*. Actually, this is a minor restriction, since to our knowledge any conceptual structure yet proposed can be represented as a subset of the domains of relations and frames (semantic nets, for example, are formed from relations, while case structures can be regarded as types of frames). More importantly, the candidate parsers are limited by the raw material with which they are constrained to work. Any natural language parser requires some information about the words in the language in order to do its work. In the CODER system, this information is contained in a *relational lexicon* abstracted from several sources, most notably from machine-readable tapes of several major English dictionaries. Providing the information from these dictionaries to the parser designer does not, of course, prevent use of different sources of knowledge (attached procedures, for instance), but it poses what may be an irresistable temptation to use the knowledge already in the system. What is more, it poses the temptation to use the knowledge in the form in which it already exists: relations among words, or relations between words and elementary domains such as **parts of speech** or **semantic categories**. Again, these relations can be used in a wide range of parsing strategies, including those listed above. And it must be noted that machine-aided translation projects, which have a comparable goal in requiring robust parsing of most text and graceful failure on the remainder, have had good results using relatively simple grammars coupled to large lexicons [GAZD 85b]. Thus we can expect this restriction also to be relatively minor in practice.

2.4 Lexicon Construction

In order for any language analyzer to be other than a toy (or at best, an interesting research project with more implications than results), it must be able to draw on a large body of knowledge about the language it is analyzing. This knowledge can be thought of as belonging to two domains of specialization: knowledge about the words in the language, and knowledge about how those words can be combined to make larger meaning-carrying units such as phrases, sentences, and paragraphs. The latter is consigned in the CODER system to the expertise base of the natural language parsing expert (or experts); the former is contained in the CODER lexicon. The lexicon, which can be consulted by several experts in the system beside the text parser, serves as the repository for syntactic knowledge (parts of speech; whether a noun is countable or uncountable), semantic knowledge (relations of synonymy between words; hierarchical relations induced from definitions), and pragmatic knowledge (appropriate realms of diction; knowledge pertaining to specific domains of discourse).

Lexicons for computational linguistic purposes have been constructed in a number of ways. Most of the classic artificial intelligence text understanding programs have used lexicons constructed laboriously by hand. Generally, these have been small and restricted to a narrow realm of discourse. Exceptions to this rule, such as TOPIC, have still required hundreds of man-hours invested in lexicon construction. Accordingly, it seems appropriate to look elsewhere for word knowledge, and to enlist the help of the computer in obtaining it. Some researchers [WHIT 83, AHLS 84] have repaired to the document text itself to discover such knowledge as permissible subjects for verbs and (candidate) taxonomic relations. Using these tools and interactive techniques for obtaining knowledge from system users, they have successfully amassed lexicons in the thousands of words. There is, however, another common source of knowledge about words: that used by humans. Dictionaries provide not only the discrete information needed by a syntactically-driven parser, but also a wealth of semantic information that may be used to establish, for instance, that the parsing of a phrase uses word senses from consistent semantic categories. Ahlswede, Evens, and Smith have all advocated (semi-) automatic analysis of dictionary definitions to streamline the lexicon construction process, and such pioneering work as that of Robert Amsler supports the credibility of such an enterprise.

In a few years, the *Oxford English Dictionary* (OED) will become available in machine-readable form, courtesy of the efforts currently by Waterloo University and Oxford University Press [HULT 84]. The OED, itself a result of decades of effort, covers virtually the entire English language, with the exception only of terms and uses that have entered the language since its completion in 1928. (Even these are covered in supplemental volumes through 1984.) Meanwhile, the *Oxford Advanced Learner's Dictionary of*

Current English (OALDCE) [HORN 74] and the *Collins English Dictionary* (CED) [HANK 79] have recently been made available by the Oxford Text Archive for research purposes. Both of these are being used in the construction of the CODER lexicon. Several other texts of considerable computational linguistic interest, including the *Oxford Dictionary of Quotations* (ODQ) and the *Oxford Dictionary of Contemporary Idiomatic English* (ODCIE), have also been obtained, and knowledge can be extracted from them as required. ODCIE in particular can be of value for matching phrases and idioms, for determining the detailed case structure of verbs taking auxilliary prepositions and particles, and for obtaining prototypical samples of use.

OALDCE is a dictionary intended for use by people learning English as a second language. Though relatively small (c. 24,000 entries), it contains a great deal of highly specific information, including simple definitions, examples showing the use of different word senses (often in the form of sentences with explanations), idiomatic phrases, national differences in spelling and usages, and verb case structures. CED is a large up-to-date one-volume dictionary, with over 162,000 references (85,000 headwords) and 14,000 biographical and geographical articles and with excellent coverage of science and technology. It provides semantic category information for fully 15% of listed definitions, several different sorts of cross-references to related words, and notes explaining proper usage of the words (see [WOHL 86] for a full analysis of the information explicit in the CED and the relations used to capture it for the CODER lexicon). Together these two dictionaries have a wealth of information to use in automatic text analysis.

The CODER lexicon is being developed in stages (see Fig. 2.1). First, the tapes must be converted from typesetting format to a structural form more suited to high-level manipulation. This involves both a clean-up phase, to remove spurious data such as page breaks and space left for illustrations, and a parsing phase, where the semantic content of indentations and font changes are translated to explicit semantic relation markers. Roger Mitton recently completed a cleanup effort for the OALDCE; the Collins dictionary was cleaned up locally during the Fall of 1985. Also locally, the UNIX™ tools *lex* and *yacc* are being used to convert the grammars implicit in the typesetting conventions of the dictionary entries to files of relations in a syntactic form acceptable to direct manipulation by a Prolog interpreter (again, see [WOHL 86] for details of the CODER CED effort). The files of Prolog statements produced as output by these parsers constitute the end point of the first stage.

Next, the CED and OALDCE data must be merged and ambiguities resolved. It is

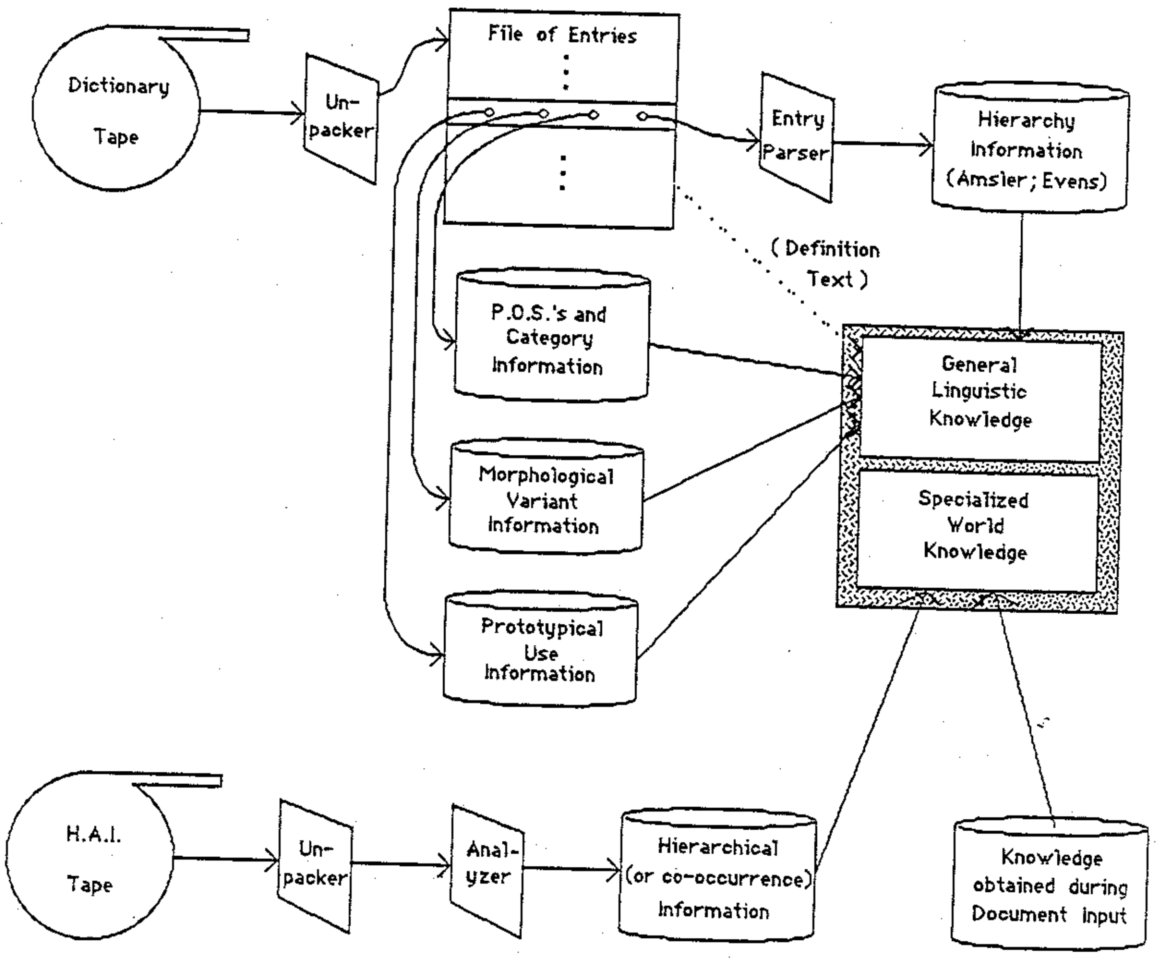

Dictionary Tape
Un-packer
File of Entries
Entry Parser
Hierarchy Information (Amsler; Evens)
(Definition Text)
P.O.S.'s and Category Information
Morphological Variant Information
Prototypical Use Information
General Linguistic Knowledge
Specialized World Knowledge
H.A.I. Tape
Un-packer
Anal-yzer
Hierarchical (or co-occurrence) Information
Knowledge obtained during Document Input

Fig. 2.1: Construction of the CODER Lexicon.

likely that a few words that occur in the OALDCE will not occur in the CED, and that many will occur only in the larger CED, but the overlap of the two is nonetheless considerable. Within that overlap, however, there may be little commonality between the two dictionaries in the differentiation of senses within a given word entry, or even in how many entries a given lexeme is given.

Third, other sources like ODQ and ODCIE can be utilized to add more information. This will provide knowledge on language pragmatics; it is an interesting open question how much help the "quotable quotes" of Shakespeare and Browning will provide in document analysis and retrieval. Additional knowledge on pragmatics can be obtained from problem domain-specific texts (for the first test collection of *AIList* messages, such knowledge may be able to be abstracted from the machine-readable form of the *Handbook of Artificial Intelligence*, recently released to researchers on a limited basis). Knowledge about words can also be input interactively during the process of document analysis, for instance when a new name is encountered in the input text.

Finally, parsing of dictionary definitions will be undertaken so that kernel words and lexical semantic relations can be identified and recorded. Smith has found in work on *Webster's Seventh Collegiate Dictionary* that a high proportion of definitions fall into a few syntactic forms, and Ahlswede has used these *defining forms* to analyze adjective definitions [AHLS 1983]. These defining forms, besides helping to identify the key terms in a definition text and their function in explicating the word sense being defined, can themselves provide semantic information about the word sense: for instance, Evens notes that the form "one who" identifies the sense as referring to a human subject. Obviously, these last three steps may proceed in parallel.

2.5 Test Collections

One of the major criticisms leveled against the experimental work done on information storage and retrieval in the past has been that the data sets used were small and controlled, and that the results obtained and techniques evolved did not scale up well to large files of real data. Specifically, models for retrieval based either on vectors or Boolean combinations of words worked well when tested on cases where the number of word types was high compared to the number of documents in the system. Recent results indicate, however, that they may not fare as well when applied in situations where large numbers of the available documents can be found containing any reasonably common word [BLAI 85]. While the system investigated in this study was a commercial system, optimized for performance and not reflective of current advances in conventional IS&R, the large-collection effect can be significant for any system. Consequently, the CODER system has been designed to function on reasonably large collections of realistic data.

As an example, the collection that has been created for the initial testing runs is a set of electronic mail messages drawn from about three years of postings to the ARPANET *AIList Digest*. About 4000 individual messages occur in the collection, each of which can be considered as a single composite document; between all the documents, the collection comprises over a million words. The documents vary considerably in length, content, style and diction, and include such disparate entities as calls for papers, announcements of seminars, requests for information, philosophical wanderings, and lists of bibliographic references (including references to previous postings in the *AIList!*). There is throughout the collection, however, a certain unity of content and a common vocabulary and body of understood knowledge. Future document collections will include documents drawn from different sources, and eventually even from several sources at once.

Use of such large collections, however, raises several issues. First, of course, there are issues of efficiency. Since the CODER project is only required to operate in a research environment, the efficiency of analysis and storage of documents is not crucial, and CPU hours can be spent assembling, analysing, and storing the collection that could not easily be spared in a commercial environment. Retrieval speed, however, is if anything more crucial in an experimental system (where exhaustive testing of different configurations involves multiple sets of retrieval runs, each of which may require many documents being retrieved and presented to the user) than in a production system. Thus the databases, not only of the documents themselves, but of the knowledge indexing the documents, must respond quickly even given the large number of documents and pieces of knowledge in the system.

Next, the use of large collections raises issues of hardware. In a research environment no less than in a commercial one, storage space on any given computer system is a scarce and valuable commodity. CODER requires large amounts of space, both for the document databases and for the lexicon. Lacking special-purpose hardware, the most reasonable solution seems to be to structure CODER as a distributed system, allowing separate knowledge bases to exist on separate machines.

Finally, the use of large collections raises issues of testing. The early work in Boolean and vector retrieval used small, well-controlled collections precisely in order that the techniques under investigation could be tested completely. With a small collection of documents, one can determine whether or not a document is relevant to any query by asking knowledgeable human beings. In collections the size of actual document data sets, this is no longer possible. While it is precisely because these data sets are too large to be cataloged by humans that the issue of automatic analysis is so critical at this time, these sets can only be used in an experimental context if we are to give up measuring system performance in terms of absolute values of recall and precision. It is possible, of course, to compare configurations of the system among each other, and to compare sets of documents retrieved by versions of the CODER system with those retrieved by versions, for instance, of the SMART system. Such comparative measures, however, will have to be our standard in working with large collections of realistic data.

2.6 AI Support Environment

The CODER project has been conceived as an investigation into the applicability of the techniques of artificial intelligence to information retrieval. Other than the question of natural language parsing, there are several ways that AI techniques can aid the information storage and retrieval process. Abstraction from the results of document parsing to the key concepts under which the document can be indexed is a process beyond the reach of conventional programming techniques, but not intuitively beyond those of rule-driven systems. The knowledge maintenance techniques required to ensure consistency of a document interpretation or a set of hypotheses about a user's information need have been explored only in the context of AI, as have the techniques required to relate facts in a knowledge base to the entities that they describe (and entities to the facts that describe them) and to trace which types of knowledge are most helpful or what sources of knowledge least suspect. Planning a search, expanding a search through the discovery of semantically related concepts, and understanding a user's response to a search all come under the general heading of areas where artificial intelligence techniques hold great promise. Use of these techniques, however, requires a commitment to an environment for their support and to the paradigms of their use.

The development of an artificial intelligence system typically follows a different paradigm than the conventional design/build/test approach [BOBR 85]. Typically, AI developers prefer an incremental, exploratory approach where system design and implementation evolve together with the developers' understanding of the problem. Maintaining a coherent system under exploratory development by several different people, however, requires more than an exceptionally steady hand. Thus it has been necessary to limit the exploration possible by any single developer working on the CODER system. Rather than limit the directions in which such exploration may proceed, though, coherence is maintained by limiting the interactions between system modules and by limiting the size of the domain within which any given developer may work. Developers working on the CODER system are required to work within the constructs of a limited set of module types obeying strict communication standards (see Chapter 4). The internal structure of each module (for instance, whether it is inferential, pattern-directed, or even procedural) is left to the designer's judgment, but the external interface it presents to the remainder of the system and the knowledge structures it represents are rigidly specified. This provides a maximum of freedom for exploratory development within the domain of a given module, while still ensuring that the modules will fit together.

Supporting AI techniques also requires a very high-level language in which the knowledge and inference techniques can be coded. This requirement conflicts directly with the requirement for efficiency in retrieval, as VHLLs are notorious for their slow execution. For the CODER project, however, efficiency is most crucial in the database aspects of the project, and there exists a language dialect, MU-Prolog [NAIS 85], that provides a very high-level paradigm oriented to artificial intelligence work and also provides strong support for large built-in knowledge bases. Prolog has been used widely in AI programming, notably for expert systems [LEEN 85] and natural language parsing [PERE 83], and MU-Prolog itself has proven effective in a variety of knowledge representation tasks [HELM 85]. However, Prolog is often cited as a better language for prototyping than for system construction (e.g., by [SUBR 85]. [DOYL 85] makes the same point, extending it to include such languages as Lisp and OPS5). This is in part due to the power of the Prolog interpreter combined with the simple and untyped Prolog environment. While a Prolog program is easily decomposed into apparently independent modules, the rule database constructed by the interpreter from a program is monolithic, and large Prolog programs often collapse into a sea of unwanted interaction effects. This effect is difficult enough in a program crafted by a single person who can, at least, ensure the purity of the local name-space. In a multi-programmer environment, it is magnified beyond endurance. This effect is avoided in the CODER project by the simple expedient of invoking a separate copy of the interpreter for each Prolog-based module. Thus, an individual experimenter can build, for example, a natural language parser or a search planner without either fearing unwanted interaction with other modules of the system or needing to worry about how those modules are built (see § 4.4).

For several reasons, therefore, we have found it necessary to break the CODER system into modules. Modular decomposition is, of course, the accepted methodology in conventional software design, but its application in artificial intelligence systems is problematic. It is relatively easy to specify the decomposition and external characteristics of a user interface; less easy, but still relatively straightforward to specify those of a frame manipulation module; very difficult to specify the decomposition of the task of retrieval, or the external characteristics of the (sub-)task managers involved. We have noted above the necessity of restricting the possibilities for exploratory development, but there is every reason to believe that our decomposition of the major tasks of the system will change as our understanding of the problem evolves through modeling and experimentation. Software engineering and AI make very uneasy partners.

Our solution to this apparent dilemna has been provided through the concept of *expert systems*. This term has been used widely in the last few years to mean many different things: here we use it to mean that the CODER system functions by applying high-level domain knowledge, expertise, to solve the problems of document indexing and retrieval. This decision serves two goals. First, it keeps expertise explicit, implying that it is coded separately from any inference engines used by the experts. This accords well with two of the primary lessons of the last decade of AI research: that intelligent behavior depends heavily on the knowledge of the behaving system, and that in artificial systems this knowledge is best engineered separately from the mechanisms for manipulating it. Second, it suggests that problem decomposition proceed along the lines of the domains of knowledge used in discovering solutions.

Each of the two primary tasks that CODER addresses is thus assigned to a group of cooperating experts. Each expert is closely bound to a single sub-task, either specific to one of the larger tasks or adaptable to either. The community of experts involved in a task can change over time, as the decomposition of the task changes, without affecting either system resources such as the lexicon or the document knowledge base, or, more importantly, the underlying structure of the system. Individual experts can also change, or even be replaced, in order to better adapt them to their missions, but such changes will not effect the other experts in the community. In fact, most changes in the composition of the community will not affect most experts: changes in the performance of the task of query parsing, for instance, need have no effect on an expert whose charge involves discovering synonyms for words.

An expert is assigned a small area of specialization. This both isolates the development of the expert from that of the surrounding system and mitigates the problems of rule interaction within the expert. Tasks which are found to be too complex can be further subdivided along the lines of the areas of expertise required to solve them. 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 84a] for a formal analysis of this effect). In practice, we expect that a few generic knowledge-handling engines can be specialized into a multitude of different experts. The work of Chandrasekanan (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.

This method of problem decomposition is particularly well suited to the tasks of document analysis and retrieval, where the relevance of a given document to a given information need is typically overdetermined by many weak factors: occurrence of certain terms or meaning structures in the document text, authorship by an authority in the field of the user's need, or currency of the information in the document, to name but a few. It is expected that it will also be relevant to a much larger class of problems where solutions are also weakly overdetermined by the solution to many different subtasks, and where the factors can be isolated by the domains of knowledge required to solve each subtask. The final design criterion of the CODER system is thus that it be built with as much generality as possible, both so that structures created at one point in the system can be used in other areas, and so that the general structure of the system, *qua* expert system, can be reused to solve other problems.

3. Representation

Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge.

(The world is the totality of facts, not of things.)

-- Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

The CODER system is designed to bring considerably more power to bear on the problems of information storage and retrieval than can more conventional systems. It does this both through more powerful *methods* and more powerful *representations*. The methods include a marriage of the more successful results of the latest generation of information retrieval systems with recent developments in expert systems and computational linguistics. The representations are symbolic structures of the type evolved in the artificial intelligence research of the last two decades with the aim of representing knowledge.

Basically, CODER will need to use at least two sorts of knowledge: general knowledge about a subject area, and specific knowledge about entities in the problem universe. It will also need to model these entities, which include words, names, and other lexical items; documents and fields of documents, and users of the system. The system needs to represent and control facts about these entities, if possible associating the facts with some sort of confidence level. In addition, it must represent and manipulate rules for recognizing general classes of entities, facts about such classes, and metaknowledge about the interaction of classes of facts. The problem faced in designing a system of knowledge representation for the system is therefore to provide facilities for modeling these entities, modeling attributes of the entities and facts relating one to another, and modeling knowledge about classes of entities, including both general factual knowledge and procedural knowledge detailing, for instance, how to classify or manipulate objects of a given class.

3.1. The Problem Universe

The primary entities in the CODER universe, as in the universe of information retrieval systems in general, are documents and users; the primary task is to provide users with relevant documents. Users have a variety of abstract characteristics, including attention span, experience in using the system and level of expertise in any of a number of subject areas. They also have information needs, which they must express to the system, generally in the form of a query. Similarly, documents have abstract characteristics, such as type, author, and date of entry into the system, and they have information content, which the system must be able to identify and represent. In the documents of the CODER domain, this content is generally carried by sections of raw text in a natural language. Natural language, however, is primarily a linear form. In order to permit effective access to the information content of input documents, the system must perform some sort of analysis on the documents and digest their text sections into more tractable knowledge structures. Thus the entities that make up text -- both atomic entities such as names and words (or word stems, or word senses) and the structures such as phrases, sentences and paragraphs that are built up of them -- are also important citizens of the problem universe.

These entities -- text items, documents, and users -- are the primary items of which the system is required to have knowedge, and thus the entities whose characteristics drive the structure and organization of knowledge in CODER. It may be necessary for the system to maintain knowledge about other classes of entities (it will certainly be useful to maintain an *authority file* of knowledge about document authors, for instance, and probably also to maintain knowledge about *bibliographic entities* beyond the documents actually entered in the system), but the focus of the system is on supplying text documents to users. The structure and organization of the system knowledge bases, in contrast, is directed by the use to which that knowledge is to be put: the matching of information sources to information needs. It is the interplay between these two requirements that informs the knowledge representations chosen for the system.

3.1.1. Text

The problem of automatically extracting content from unconstrained text has been the focus of artificial intelligence research since the dawn of AI as a self-conscious field. Much of the early work in knowledge representation (for instance, Quillian's work on semantic nets [QUIL 66]) was directly motivated by linguistic considerations. And while early research into machine digesting of text was as disappointing as early work in machine translation,† recent advances in both natural language understanding and knowledge engineering make possible a cautious optimism. One of the purposes of the CODER project is to evaluate the gains such sophisticated analyses can provide over atomistic word-frequency indexing. This goal in turn induces two requirements on the knowledge-representation facilities of the system: CODER must support both the knowledge structures used in natural language parsing and the (possibly quite different) structures required for the representation of text content.

In compiling a computer program, it is customary to make a sharp split between the analysis phase, where lexical and syntactic items of the language are recognized, and the generation phase, where equivalent structures of a different language are created. A similar split can be made in the process of deriving knowledge structures from natural language text. For instance, while different members of the Yale AI group have used a number of parsing strategies (top-down, bottom-up, demon-based) to analyze incoming text, they have maintained a set of philosophically consistent structures for representing text content throughout. Of course, in the area between parsing text and generating abstract representations of content, a large grey area intervenes that is usually described as 'semantic.' What is within this area (indeed, whether there is anything there except for the interaction between the syntactic constraints of the language and the logical constraints of the knowledge into which it is being mapped) is currently an area of active research. It seems unlikely, however, that the representational structures required in this phase of natural language understanding will differ greatly from those required on either side.

Recent advances in formal language theory, as well as the renewed interest in computer articulation of natural language generated by the Fifth Generation project, have resulted in a multitude of promising new parsing techniques, among which we can mention

†As recently as 1983, for instance, it was possible for Gerald Salton to state that "frequency-based phrase-generation systems are simpler to implement and are currently more effective" than linguistic analysis in indexing a document for retrieval [SALT 83a, p. 91]. Salton, however, does not cite comparative studies later than 1970, a date at which computer text understanding was still in an extremely primitive state. More recent relevant work includes the TOPIC system [HAHN 85] as well as the work of C.D. Paice (e.g. [PAIC 81]) and Karen Sparck Jones [SPAR 84a].

those based on head and phrase structure grammars (e.g., [POLL 84]), indexed grammars [GAZD 85a] and unification [SHIE 84], as well as the more established techniques of DCG-based and linguistic string parsers. The choices for text representation formalisms are somewhat narrower, but nonetheless include many variations on several distinct groups. Semantic nets are still advocated (e.g., by [SIMM 84]), as are related structures such as J. F. Sowa's 'conceptual graphs' [SOWA 84]. Roger Schank's 'conceptual dependencies' are widely used: related structures have been advocated by Lehnert [LEHN 78] and Wilensky [WILE 84]. The KL-ONE system has been used with some success [BOBR 80, SOND 84], as have other frame-based or mixed frame/network models. Finally, systems based on transformational grammar, such as [SAGE 81], make use of formalisms based on the transformed deep structure.

Selecting among this wide variety of choices, both among the parsers and content representations available, to determine the best mix(es) for the task of information retrieval is no small task itself. It is reasonable to assume, in fact, that it can only be done on an experimental basis in a context that supports a number of different paradigms. The CODER project attempts to provide such a context through architectural features ensuring the decoupling of parsing paradigms from the rest of the system, through a lexicon of word knowledge for parser designers to draw upon, and through a set of common knowledge representation formalisms within which various content-representing structures can be designed.

3.1.2. Documents

Early work on text retrieval tended to represent documents first as flat streams of text and then as equally flat abstract structures such as term vectors. While this approach is adequate (at least in small collections) for running text, it is patently inadequate for dealing with such text items as names and dates, and fails completely to handle structural features such as the delineation of the abstract (or even the title) of a journal article from the body of the text. In contrast to this, most commercial bibliographic services have used highly structured database technology to control such document features as authors, publication information, and subject descriptors. Quite apart from the well-known difficulties of adapting database technology to text items, which vary widely in length, internal structure, and conventions for lexical ordering, this approach requires the service of human catalogers to convert authors' names to authoritative form and dates to canonical format, not to mention choosing subject descriptors from an approved and controlled thesaurus. This amount of work is prohibitive for all but the most important collections.

More sophisticated analyses of document structure have been performed in recent years, although no attempt has yet been made to adapt these analyses to information retrieval. Reacting to a perceived need for standards of interchange among different organizations involved in the production of electronic documents, several international organizations (notably the ISO) have proposed standards for electronic document structure [HORA 85]. Gary Kimura [KIMU 84, KIMU 86] has investigated a model of documents as heirarchical objects formed from both ordered and unordered sets of more primitive objects. Both Kimura and the ISO model recognize a *logical* structure and a *layout* structure in documents, the former governing the decomposition of a document into title, sections, paragraphs and the like, and the latter providing the syntactic clues (centering of a title, use of font information, and so forth) that humans use visually in discerning the relationships among text items on a page. The COBATEF system [PEEL 85] presents an extremely fine-grained synthesis of these two dimensions based on a unifying box-and-glue model for text objects.

Documents prepared on one or another of these models wear their internal structure on their sleeves. Other electronic images of documents are not as revealing, at least to an artificial system. An electronic document prepared under, for instance, TROFF or T_EX includes a great deal of formatting information designed to make its logical structure obvious to the human eye. Mapping this structure into abstract form is, however, non-trivial, and it is reasonable to expect that only part can be automatically recovered. An intermediate case is posed by electronic mail messages, which are produced with some explicit structure in the fields of their headers, but also carry free text (with, perhaps, implicit structure) in their body and subject fields. Recently, there have been proposals (e.g., [MALO 86]) to push more of the information content of mail documents into such structured fields through the use of template-based editors and mail systems.

Beyond straightforward decomposition into, for instance, title, author and body segments, and even beyond more complex and informative hierarchical decompositions, documents and document aspects can stand in logically more involved relationships to each other. Kimura, for instance, points to the possibility of structure sharing within a document, as when an equation or table is referred to several times within a text section.

In addition, documents can be nested recursively within one another, as when one mail message forwards another that itself contains fragments of a third.† Even a single document (a single body of work by a single author at a single time), may be best described by a nested set of interpretations, as when a mail message parenthetically includes a list of bibliographic references or when a request for information includes a description of the work motivating the request. Finally, a document may have more than one consistent interpretation, either because the purpose of the document is ambiguous or simply because, in the way of natural language documents, it serves more than one purpose at once.

Thus documents, like natural language text, pose two different (though related) problems: *recognition* and *representation* of structure. Representation requires a set of knowledge structures sufficient to capture the different aspects of different types of documents. Recognition requires knowledge of both types and prototypes: it requires the ability to fit a document to a matching pattern (top-down) and to recognize elements (bottom-up) that may signal particular document interpretations. Note here that prototype knowledge of document structure should not overly constrain interpretation: it should be possible to build an interpretation of a document based on a partial match to a prototype, or on sufficient similarity to a prototype. Although this depends on the inference techniques employed, rather than the knowledge representations themselves, it points up an important difference between the patterns used in parsing document structures and the patterns used to build abstract interpretations of the documents. The parsing knowledge includes a great deal of metaknowledge about the aspects of a document and the relations among those aspects (for instance, that one aspect *depends* on another or that two aspects are *mutually exclusive)* that has less to do with the internal structure of the representation than with its relation to a model. Parsing also requires knowledge about the relations among different document representations: which are consistent with each other, which can lead to one another, and what the clues are for following those leads. In short, representation involves knowledge of distinctions and relations among individual documents (or document components); recognition involves knowledge of distinctions and relations among

†Babatz and Bogen [BABA 85] present a number of different relations that may obtain among electronic messages, including *revising, commenting on,* and *rendering obsolete.* In the domain of journal articles, the usefulness of citation and co-citation relations for retrieval is well-known [BICH 80]: it is necessary to identify and parse bibliographies and to recognize references in text, however, before such relations can be used in a fully automatic system.

document prototypes.

3.1.3. Users

Recognition of users is generally easy, because it is generally artificial. In a computational environment, users are usually recognized by their ID's, or conceivably by their names. This is the most likely mechanism for use in CODER, although it is reasonable to imagine such an identification being defeated by a combination of other evidence. This is possible because, unlike most systems, CODER maintains more information about each user than her ID code and permissions. Individual users have a number of aspects that are important in tuning the information retrieval process to their needs. These include general characteristics such as attention span and competency in system use, as well as characteristics specific to the information-retrieval process, such as how many documents the user can absorb in a session and the relative value she puts on recall and precision. The user also has a history of interaction with the system: what search techniques have helped most in filling her needs, in which conceptual areas useful documents have been found ... in short, any common aspects of success that the system can abstract.

In addition to knowledge about a user discovered by the system, the information-retrieval task can be aided by knowledge offered by the user. The user may suggest semantic relations between words, variants on representations, or expert judgements from her own area of specialization. Such knowledge should not, as a rule, be regarded with the same confidence as system knowledge, particularly when working with a different user. Neither should it be discarded, however, for it may aid greatly in later searches for the same user. It is reasonable, thus, to maintain it as local to the user. Conceivably, as the system gains experience, knowledge suggested by several users or knowledge from an expert user's specialty can be allowed to pass into the system's central knowledge bases.

3.1.4. Constraints on the Universe

Restriction of the CODER problem universe to the symbolic world of text and documents provides several fortunate constraints on the knowledge required by the system. The system, for instance, has no need to understand spatial knowledge, nor does it need any understanding of primitive physics. Further, while document representation involves a fair amount of temporal information (an electronic mail message may have a *date/time of origin*, a *date/time of receipt* and a *history* of access since its receipt, and may as well itself describe an *event* with a date and time of occurrence) that information occurs only in well-defined subareas which require a limited subset of the full vocabulary of temporal knowledge. All the time information involved in bibiliographic descriptions, for instance, can be represented as instants on a quantized line, thus eliminating both problems of duration and any trace of Zeno's paradox. In addition, the number of temporal *relations* of bibliographic interest is limited: we want to find documents with origin before (or after) that of a given document, or between two arbitrary dates, but that is about the extent of our interest. Finally, the problem domain does not include any 'natural' continuous functions. One can, of course, introduce continuous functions into the system. The CODER prototype, for instance, uses continuous functions to represent belief and similarity in time-honored fashions. Such functions can, however, be restricted to the (sub-)domain of their introduction and handled there, without the need to involve the basic knowledge representation mechanisms. The problem universe is thus, for all its complexity, extremely well-behaved, and it has been possible to cover it with a parsimonious set of knowledge representation formalisms.

3.2. Knowledge Formalisms

One of the most striking findings of the last decades of AI research has been the importance of explicit knowledge to intelligent behavior. In the CODER system, two distinct types of knowledge are recognized: *factual* knowledge about entities in the problem universe, and general knowledge or *expertise* within a domain of discourse. Different architectural constructs are provided for the storage and manipulation of these two types of knowledge: expertise is controlled by *experts* and factual knowledge by *external knowledge bases*. The CODER system as a whole is made up of communities of these specialized experts, communicating with each other and with system users through a set of active blackboards (see Chapter 4). Each expert consists of an inference engine and a local knowledge base structured to suit the type of inference performed: a backward-chaining engine would draw on a body of rules, while a classifier might require a discrimination net. The exact form of the expertise is thus a design decision local to each expert. Forms of domain knowledge can be varied or even invented by the knowledge engineer working on the particular specialty being modeled. When knowledge is communicated by an expert to the rest of the community, however, or when it is stored for use in later phases of processing, it must be in a form independent of any particular decision taken by the individual designer or implementer. That form is provided by the factual knowledge described in this chapter.

The CODER system uses three levels of representation formalisms for factual knowledge (Fig. 3.1), each built recursively on the lower levels. These three levels are used to model attributes, entities, and logical relations among entities. The semantics of each level is defined functionally, in terms of the operations for manipulating objects from each level and in terms of the *subsumption hierarchies* that relate types and objects of that level to the others. A lattice-theoretic approach to inheritance is taken throughout the knowledge administration system [SCOT 76; though cf. CART 85], although the significance of arcs in the lattices differs among the levels. The external knowledge bases are also defined functionally, so that the knowledge that the system has of the world can be made available to more than one expert through a common interface. Designing the functionality of knowledge manipulation in concert with the representation formalism ensures that the knowledge bases will remain free from paradoxical behavior [LEVE 84b]. It also provides for a clean architecture, as it permits a generic external knowledge base to be reused throughout the system.

3.2.1. Elementary Data Types

At the lowest level, attributes are modeled in CODER by *elementary* data types (EDT's). These are the sorts familiar from the theory of abstract data types [YEHR 78]: sets of objects associated with sets of valid operations ranging over them. The classical types **char, real, int** and **atom** are provided in this category; new elementary data types can be created through the specification of the set of operations defining them, or through quantification or restriction of existing elementary types. The EDT **part of speech**, for instance, is a restriction of the **atom** type; the type **phone number** is a quantification (**10**

```
relation  ::= relation_name {argument}+

   argument ::= relation
              | frame
              | elementary_object
              | quantifier argument


frame     ::= frame_name  {slot_name slot_filler}*

   slot_filler ::= frame
                 | elementary_object
                 | quantifier slot_filler


elementary_object ::= {quantifier}  primitive_object  {restriction}

   quantifier ::= list_of   |  non_empty_list_of
                | set_of    |  non_empty_set_of
                | integer
```

Fig. 3.1: Definition of factual knowledge representation formalism.

digit) of a restriction of **char**. Restricted EDT's inherit the operations of their parent type without change; quantified EDT's respond to the operators of their quantifiers by returning objects of their parent type. Thus the two constructors together form a complex but well-behaved semi-lattice above each primitive EDT, and we can say that if EDT **a** is either a quantification or a restriction of EDT **b** then **b** subsumes **a**. EDT's may also be constructed by juxtaposition, where one type is followed by another (possibly from a different lattice altogether). In this case, the constructed EDT is defined to be weaker than either of its parent types, and responds to the constructor operations by returning objects of the parent types.

Elementary types, thus, are interconnected in a set of semilattices above the primitive types. No such relation holds, however, among the elementary data *objects*. This is not to say that objects of a given type may not have a linear ordering. In fact, all primitive types that submit to restriction must have such an ordering for the restriction syntax to have meaning. Such relations, however, are independent of the place of the type in the subsumption hierarchy. A **3** drawn from the restricted type **[1-10]**, for instance, cannot be considered weaker than a **32** drawn from the primitive type **int**. Similarly, it is not meaningful to ask for a *new* instance of a primitive type. One can, of course, ask for a number or an atom that one has never seen before (the Lisp **gensym** function has just such a semantics), but one cannot ask for an instance guaranteed to always be unique: one occurrence of the atom **g000039** will always be indistinguishable from every other occurrence of **g000039**. Both of these properties are entirely appropriate for models of attributes. Neither, however, hold for objects of the higher levels.

3.2.2. Frames

At the second level, the CODER knowledge administration facility provides primitives for modeling entities. The model of an entity may have many aspects, including both attributes and, possibly, subsidiary entities. The set of aspects germane to any particular sort of entity, however, is limited, as are the possible values for any aspect. Not every entity, of course, will have all germane attributes (certainly not all aspects of an entity will be *known* for every entity) and not every interesting aspect of an entity will be captured within the set of germane aspects. The set simply captures those aspects typically associated with an entity of that type. In light of this, the clear choice of formalism for modeling entities is some variant of *frames*. In CODER, we have taken an approach to frames based on types: a frame instance is always the member of some frame type, and a frame type is defined by the set of its slots and their associated types. Slots are named, as are frame types, in order to provide a method for distinguishing two slots of the same type when either may be missing from a particular frame instance.

More formally, a *frame type* consists of a type name together with a set of slot definitions, each of which consists similarly of a slot name together with the (maximal) type of object that may fill that slot. A *frame object* is an instantiation of some frame type: the type name together with a subset of the allowed slots, each of which is filled by an object of (a type subsumed by) the type specified in the frame definition (see Fig. 3.2). This definition of frames differs from early frame systems such as [MINS 75] and [BOBR 77] in two important ways, both involving the notion of types. First, CODER frame objects have limited and controlled slot sets. Many frame systems allow arbitrary slots to be added to any frame object, making each frame object in effect the unique member of its own kind. This creates problems both for a coherent description of kinds and for the abductive classification of entities [BRAC 84, FAHL 81]. The frame system in CODER, by contrast, distinguishes between frame types, which represent (natural) kinds† and which fall into a subsumption hierarchy based solely on the attributes that objects of those kinds are expected to possess; and frame objects, which represent descriptions of entities

†It is not our intention here to join the debate on the best definition of the "natural kinds" involved in classification of peonies and penguins. It is in recognition that this debate wages unresolved that we use the parentheses above. We do want, however, to distinguish a notion of "kind" different from the formal notion of type, under which penguins can be members of the kind **bird**, even though they have no **average flying speed** aspect. There are two ways that the distinction between frame types and objects in the CODER frame system contributes to a natural modeling of this notion. First, while possession of certain attributes by an entity may be important to the inductive recognition of that entitiy as being a member of a kind, it is not necessary for the frame object that describes the entity to inherit all the slots of the frame type describing the kind. In particular, default slot fillers may be replaced or even removed. Second, while the aspects necessary to recognition of an entity as being a member of a kind are represented by filled slots in the corresponding frame object, accidental aspects of the entity (for instance, that a particular penguin *has a large nose)* are modeled by facts predicated on the entity. Such facts are described later in the main text; note here only that those facts have both a different status and a different location than facts about the kind **penguin**. Facts about entities are stored in one of the external knowledge bases describing the problem world. Facts about kinds, however, are stored in classification experts, where they may be used in wholly different inference chains. Facts that a member of a kind must have a certain aspect (that the corresponding slot must be filled), that a member may only have one of two mutually exclusive aspects, or that members of one kind are also likely (or unlikely) to be members of another, while vital to the recognition of natural kinds, are simply confusing when applied to descriptions of individuals. The CODER system makes sure that these stay separate.

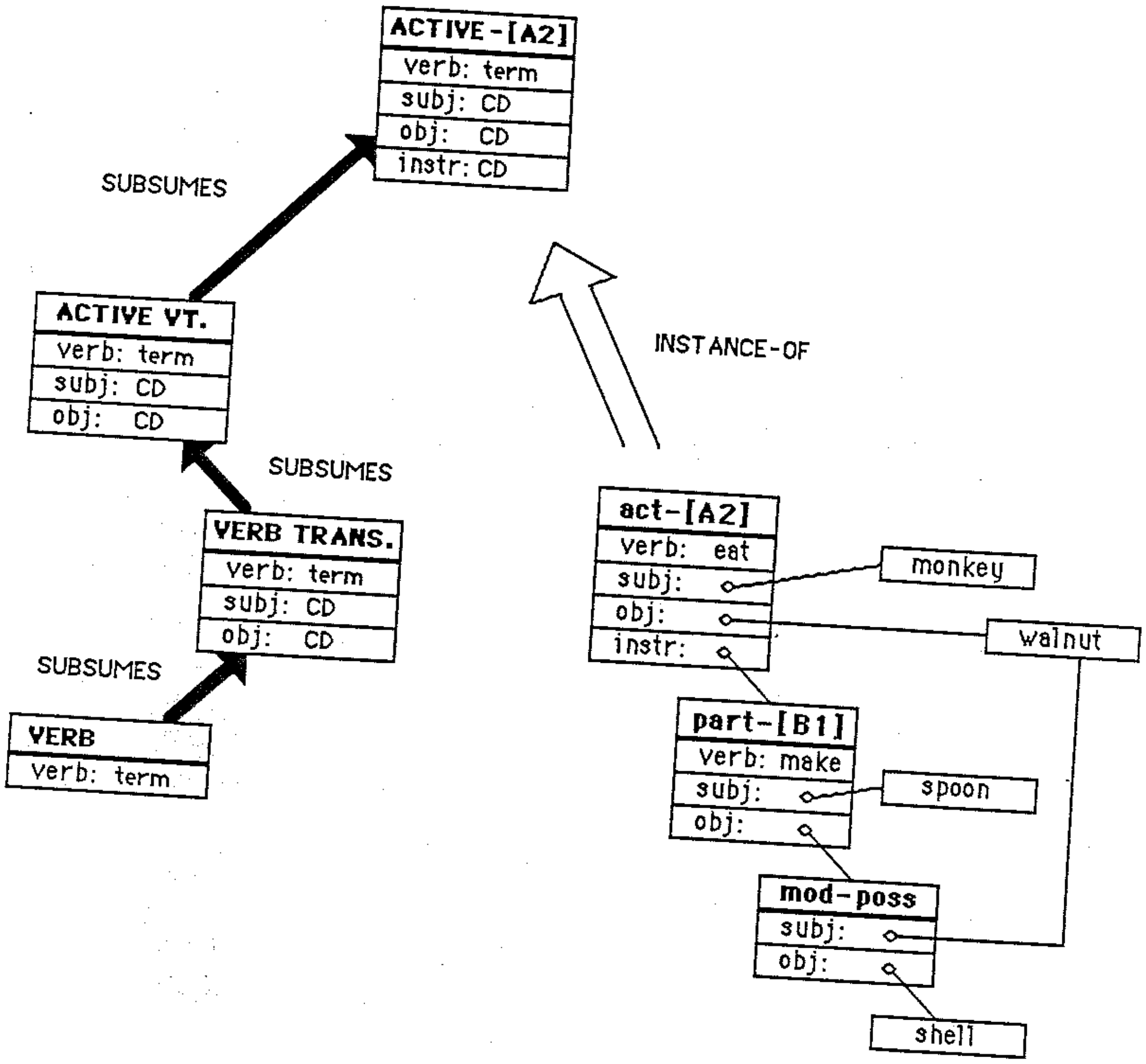


Fig. 3.2: Frame subsumption hierarchy for Conceptual Dependency forms, with example instantiation.

as members of those kinds. Second, the slots in CODER frames are restricted by the types of objects that may fill them. This restriction serves both a formal goal, ensuring that instantiations of a given frame type are also instantiations of all types subsuming that type, and an engineering goal, ensuring that objects created in one part of the system can be understood in another part.

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 (see Fig. 3.3). Thus 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 [BRAC 83a]. 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, CODER maintains this knowledge in a set of *frame type experts*, 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. 3.4). 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.

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

Fig. 3.3: Semantics of frame subsumption.

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

match(elt1, elt2)  ⊃
    elt1 = elt2.
```

Fig. 3.4: Semantics of frame matching.

(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.

Both the properties of elementary data objects noted above are violated for frame objects. First, a frame instance is always identified as to its type. Thus, it is possible to have two frame instances which have the same slot values, but where one type (and thus, one instance) is weaker than, stronger than, or even incommensurate with the other. Second, separate frame instances are separate computational objects. While two frame instances may be equal, they are never the same. Of course, two different frames may both model the same entity. Indeed, the entire process of retrieving on frames is based on being able to interpret whether it is possible for two frame instances to describe the same individual. This is exactly what is performed in matching: a test for matching frames establishes the consistency of the two descriptions represented. Consistent descriptions can then be synthesized (into a new frame object of the type at which they match) or compared for similarity as required.

3.2.3. Relations

The top level of knowledge representation in CODER is provided by the domain of logical relations. Relations model propositions over objects: **synonymy** is a relation between terms; **representing,** a relation between a sentence and the meaning structure produced from it by a parser. Propositions can be used to model attributes of entities that are not paradigmatic of their type, or to model relationships among entities. Relations may have any number of arguments, and may include weights, but always either obtain or do not obtain: unlike frames, they cannot partially apply to an object. The CODER system provides facilities for definition and use of relations, including specification of the types that may occur for each argument, and for maintaining knowledge of the attributes of relations, for instance whether or not a given relation is symmetric, transitive, and so forth.

The relation types known to the system can be used in representing general knowledge within the local knowledge bases of the system experts; skeletal propositions are typically used to query the external knowledge bases. Ground instances of logical relations, however -- those containing neither variables nor meta-terms -- are said to be *facts*.

Facts are the basis for complex knowledge structures. It is possible, first, to build *relational structures* nested to arbitrary depths. These can be used to represent situations or concepts, but can be tested neither for equality nor for matching. Unlike frames, relational structures have no orientation, so they must instead be compared for isomorphism, a computationally expensive process. While the expense can be alleviated by first comparing the objects on which the structures are built, it is still desirable to avoid deep nesting. The *hypotheses* used to communicate among experts are also built from facts, each combined with a confidence level, the name of the expert proposing it, and its dependencies on other hypotheses. Finally, facts can be combined into *interpretations*, as when a document is interpreted as being an **electronic mail message** or a **report of event**. Interpretations are said to be consistent just in case an entity can be found for which all the facts in the interpretation are true. An entity may have more than one consistent interpretation, but an entity without an interpretation is one of which the system has no knowledge.

Canonically, these three domains are used to represent knowledge of attributes, individuals, and facts about individuals, respectively. The formalism can, however, be adapted to other purposes, such as the representation of meaning by semantic nets. Relations can be used to model a wealth of associative schemes, including J. F. Sowa's "conceptual graphs" [SOWA 84], and frames, of course, can be used to model such specialized entities as case structures and Conceptual Dependencies. Elementary data objects provide both the power of abstract data typing and a primitive method of operation inheritance. Moreover, note that classical database tuples can be considered to be ground instances of relations that may not have relations as their arguments (database relations, therefore, are sets of CODER relations that obey this restriction and the various normal forms. This is terminologically confusing, but perhaps apt). The various structures used in classical retrieval, such as term vectors and p-norms, can also be easily embedded in the factual representation language. This language thus encompasses most existing AI, CL and IS&R formalisms, and there is good reason to believe that it will continue to be adequate for any representation formalism that the problem domain requires.

To summarize, the CODER system is able to represent knowledge *that* an individual exists through creating a new object with (potentially) no attributes. It is able to represent *what* the individual is (within the definition of its kind) through associating attributes with the object, and it is able to represent knowledge *about* the individual through facts that make reference to the object. Knowledge about individuals is the only knowledge stored in the external knowledge bases or passed among experts as hypotheses: to know something in CODER is to know a fact about a (set of) object(s). Facts and objects are mutually supportive in the CODER environment, and neither can exist without the other. It is arguable that this is true in human cognition as well; *passant* Dr. Wittgenstein above, we make no such claim here. All that we note is that this knowledge formalism is both logically sufficient and computationally effective for modeling and communication within the system.

3.3. Knowledge Structures

Resolving the problems of analyzing documents and matching their representations to users' information needs involves both general and factual knowledge. This chapter concentrates on the factual knowledge, since this is the knowledge that must be most closely controlled between modules and versions of the system. Specifically, we examine knowledge of the terms in the language of the documents, knowledge of the meaning structures built from these terms to describe document content, and knowledge of how such terms and structures, in the context of the type of document involved, determine consistent interpretations representing the documents.

3.3.1. Terms

The words encountered during document analysis are often not the words that one finds in a dictionary. Words in text (what we shall here call *lexical items*) may relate to their corresponding dictionary entries only through a set of transformations including lexical mappings (such as capitalization conventions), morphological mappings (such as declensions and nominalizations) and other mappings, more difficult to classify, such as misspellings and geographical patterns of use. Words in the dictionary, in contrast, occur as entries with both internal structure and external relationships, such as that between a word and the words used in defining it.

Still less are the words encountered in document analysis the terms under which the document should be indexed. Even given a solution to the problem of identifying which words in the document are important to, or crucially descriptive of, the document content, and even given a method of systematically stripping away the syntactic transformations described above, there remain difficulties in discovering which words are important to the descriptive process in themselves and which are better clustered together or transformed to some canonical descriptor. A typical human cataloging system uses both a canonical thesaurus of subject descriptors and free terms drawn directly from the document at the discretion of the cataloger. Mapping from the words in the text to a set of content descriptors for a document is a subtle and complex process.

The first step taken in the CODER system to identify the content of document text involves the transformation of *words* to *terms*. A term is an unambiguous semantic item that can be said to be exemplified by the word in text. Terms and lexical items have a many-to-many relation: the word "will" as in "force of will" maps to the same term as does "voluntary," but a different term than "will" as in "last will and testament." To place the discussion of terms in some perspective, consider the CODER lexicon.

The CODER lexicon is built primarily out of machine-readable versions of English dictionaries, and a classical dictionary hierarchy [AMSL 80] has been used to structure the knowledge derived. Items in a dictionary occur at one of a number of levels (typically three or four, although the maximum depth of the hierarchy varies from dictionary publisher to publisher). The topmost level is invariably the level of the entry headword, or *homograph*. Separating a word into several entries denotes a deep distinction in meaning, such as the different meanings of "will" mentioned above. Some dictionaries, such as *Webster's Seventh New Collegiate Dictionary* [PETE 82], create separate entries for each change of syntactic function, separating for instance "will (n.)" (the legal document) from "will (vb.)" (to leave by a will). The more common practice, and that followed in the CODER lexicon, is to divide each entry into one or more parts of speech, which are themselves separated into definitions and subdefinitions (see Fig. 3.5). Linguistic knowledge may attach at any point in the hierarchy and is inherited in the standard way by the subsidiary levels. Inherited knowledge is not impeachable but may be refined, as when particular definitions in a cluster of verb entries are labeled transitive or intransitive.

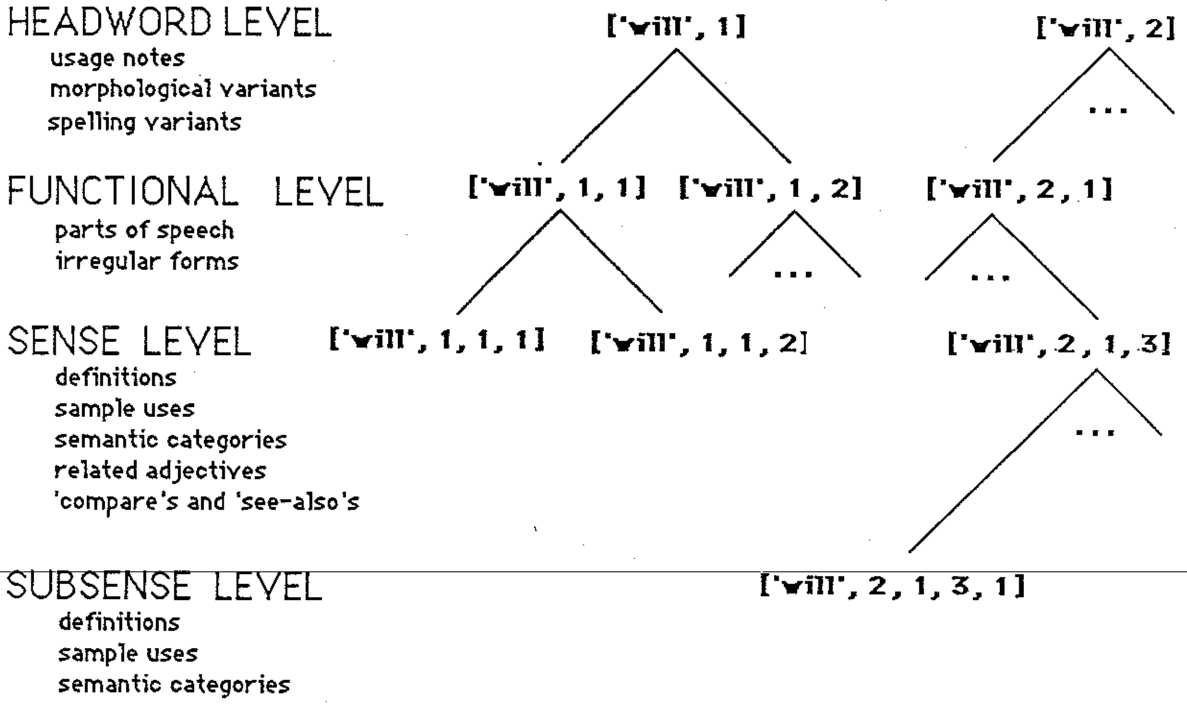


Fig. 3.5: Hierarchical structure of terms in the Lexicon, showing knowledge attaching at each level.

with the uncontrolled terms in the remainder of the lexicon so that correct descriptors can be selected even when they are not themselves present in the document. Exemplar knowledge and pragmatic knowledge (of, for instance, the appropriate contexts for a controlled term), both of which are available from the *Handbook*, may also be of benefit in the cataloging process.

3.3.2. Concepts

More often than not, the knowledge embodied in a section of text will not be precisely captured by a set of atomistic terms. In this case, it is necessary to supplement such a base-level representation with more complex structures in order to adequately represent the meaning of the text. Such complexity, of course, brings about a corresponding difficulty in matching the structures to a given statement of a user's information need. In CODER, we are exploring two major types of possible approach.

During the parsing of natural language text, terms are discovered, not atomistically, but in relation to other terms in the text. Thus one obvious type of structure to use is a direct mapping of those relations into either a semantic net or case frame model. Following this path, for example, would allow the system to index a document which referred to "knowledge representation implementation methods" with a single noun-group structure, rather than a set of primitive terms for "knowledge", "representation" and so forth. Note here that both Simmons-style semantic networks and J.F. Sowa's 'conceptual graphs' can be embedded in the CODER relation domain, and conceptual dependencies and case structures [FILL 68] embedded in the frame domain Several experimental comparisons can thus be made without modifying the over-arching knowledge representation formalisms.

Alternatively, one can set up a group of prototype models involving concepts that are likely to occur in the document collection. Part of the task of document cataloging then involves attempting to match sections of text to such prototypes in a mutually constraining process with the establishment of document type. For instance, in the *AIList* collection there are several recurring types of document, including calls for papers, requests for information, reports on lectures, and so forth. These can be thought of as establishing expectation-based parsing skeletons in the spirit of Roger Schank's *scripts* [SCHA 77], though requiring a different set of primitives. The canonical events in these scripts provide structures to be matched to the text of any given message: recognition of the structures motivates classification of the message to a script, and triggering of the script creates expectations of finding additional structures. Although not every message will fit neatly into such a set of canonical scripts, this approach has considerable promise in that, for messages that do match the prototypes, it is easy to establish which meaning structures are relevant to the indexing of the document.

3.3.3. Document Interpretations

Ultimately, the goal of document analysis is to create a set of knowledge structures to store in the document knowledge base. This knowledge can involve facts of two sorts: assertions that a given document (part) is of a certain type, and assertions that it is described by a certain term or meaning structure. Clearly, these two types of facts are related: a document (part) of one type is more likely to have certain kinds of meaning structures associated with it than another. Metaknowledge about such relations is maintained by a document classification expert, which can both *identify* facts already established about a document as being consistent with its description, and *suggest* certain types of facts, usually associated with documents of that type, as things that need to be established. We will call a description of a document as being of a type, together with a constellation of facts about the document consistent with that description, an *interpretation* of the document, and say that the facts *substantiate* and that the document *models* the interpretation.

In the *AIList* collection, for instance, there are many different types of documents: announcements of lectures, requests for information, lists of bibliographic references, and so forth. Each such message type defines a document type (what we will call a "soft type"), which is represented initially as a frame type including slots appropriate to the message type. These frame types make a "tangled hierarchy," where each node can have both more than one parent and more than one child, that is itself part of the overall frame subsumption hierarchy. Each document in the collection also has a "hard type" (in this case, all subtypes of the **email message** document type) that can be established at input time (Fig. 3.6). In addition, each document has content, represented by facts about the terms and meaning structures that can be derived from the document text. Creating an interpretation for a document in the collection thus becomes a task of identifying a maximally constrained type in the document hierarchy and a set of content facts consistent

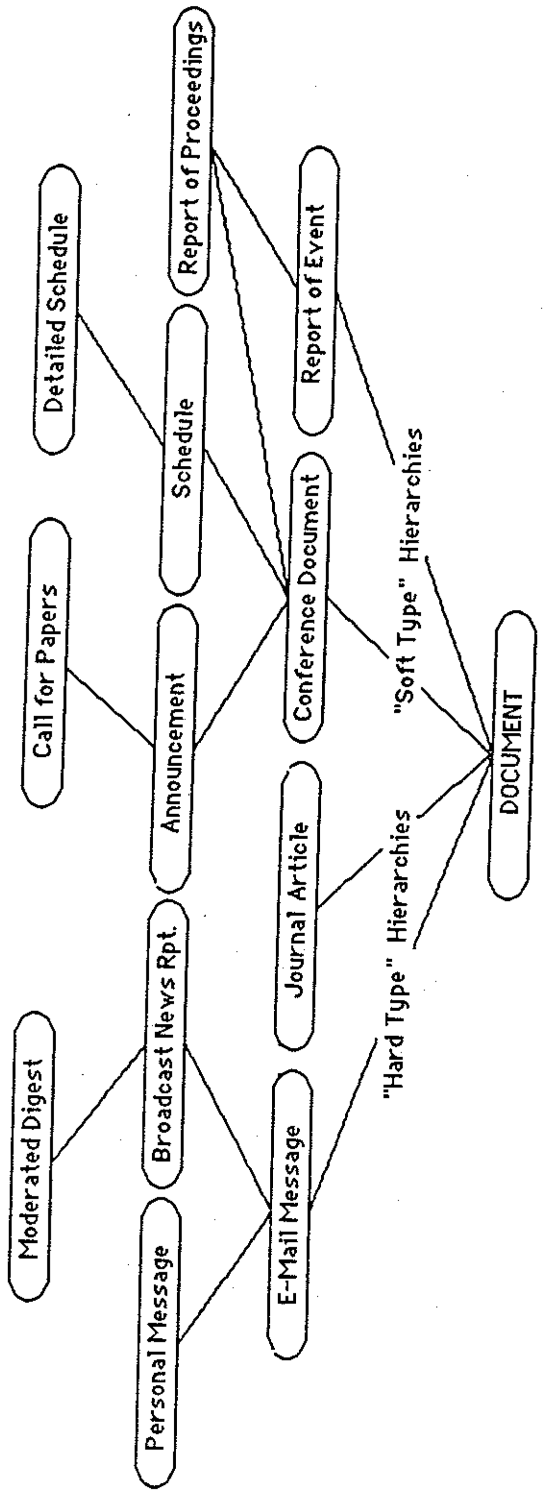


Fig. 3.6: Partial document type hierarchy.

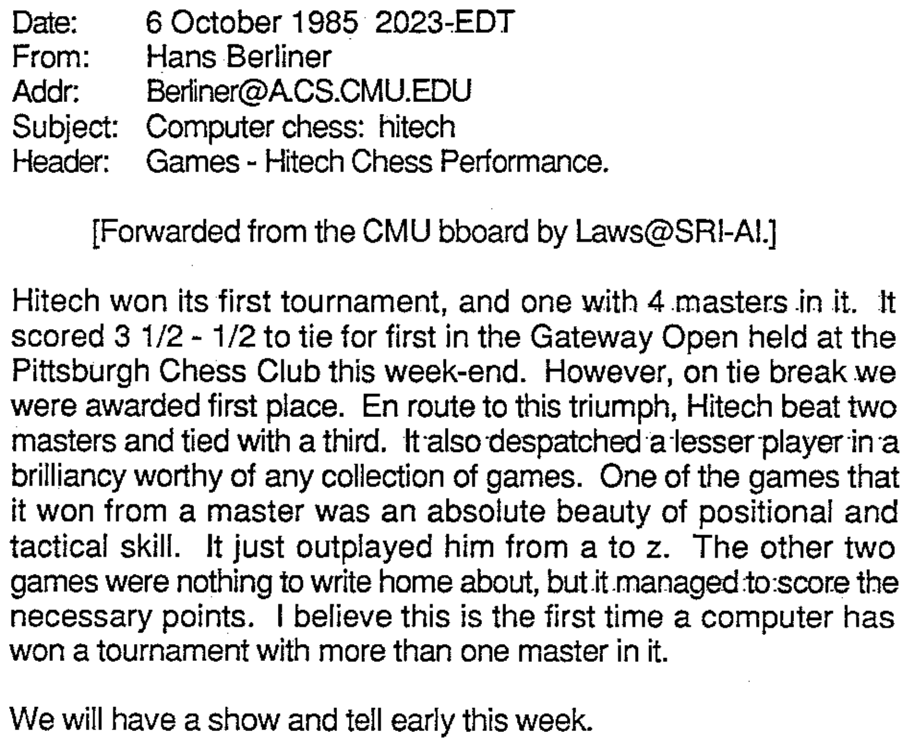
with it.

The CODER system pursues this task both bottom-up, analyzing the text and structured fields of the document to locate descriptive terms and meaning structures, and top-down, attempting to match the document to one or more prototypical document types, and to use the metaknowledge associated with those types to identify significant events and structures within the document. Typically, documents will have more than one interpretation; at least one based on hard type, and possibly one or more based on soft types. Only those interpretations that are best substantiated are stored in the document knowledge base, ensuring that facts are not stored without context. Facts used in more than one interpretation, of course, are particularly good candidates for descriptors. This selection process provides a base of document knowledge which, while not consistent overall, includes a consistent context for any fact it holds.

3.4. An Example

All knowledge that the system holds about documents and users is in the form of interpretations: sets of consistent facts that model the entity and provide computationally effective abstractions of its salient characteristics. This knowledge is built up slowly, during the process of analyzing documents as they are input, and users as they react in retrieval sessions. In contrast, the system's knowledge of words, contained in an external knowledge base known as the system *lexicon* is largely redacted from outside sources [WOHL 86]. This word knowledge, together with the expertise built into each specialist expert, forms the core of the system's ability to react intelligently to new documents and queries. To understand how this is so, consider this example.

On the occasion of Hitech's victory at the Gateway Open, the message in Fig. 3.7a was distributed through the Digest. Some time later, let us say that the query in Fig. 3.7b was presented to the system. Obviously, the information in the document is relevant to the query, but certain problems for automatic recognition of that fact are immediately apparent. First, the document contains very few words in common with the query. The query is about a "program" "defeating" an "opponent:" the document about a "computer" "winning" against "masters." Next, the terms used, while clear enough to the human reader, are lexically ambivalent. "Master" alone has numerous senses. Then again, the



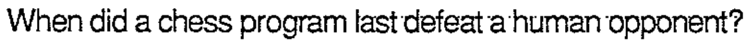
Date: 6 October 1985 2023-EDT
From: Hans Berliner
Addr: Berliner@A.CS.CMU.EDU
Subject: Computer chess: hitech
Header: Games - Hitech Chess Performance.

[Forwarded from the CMU bboard by Laws@SRI-AI.]

Hitech won its first tournament, and one with 4 masters in it. It scored 3 1/2 - 1/2 to tie for first in the Gateway Open held at the Pittsburgh Chess Club this week-end. However, on tie break we were awarded first place. En route to this triumph, Hitech beat two masters and tied with a third. It also despatched a lesser player in a brilliancy worthy of any collection of games. One of the games that it won from a master was an absolute beauty of positional and tactical skill. It just outplayed him from a to z. The other two games were nothing to write home about, but it managed to score the necessary points. I believe this is the first time a computer has won a tournament with more than one master in it.

We will have a show and tell early this week.

Fig. 3.7a: A representative message from the *AIList* collection.



When did a chess program last defeat a human opponent?

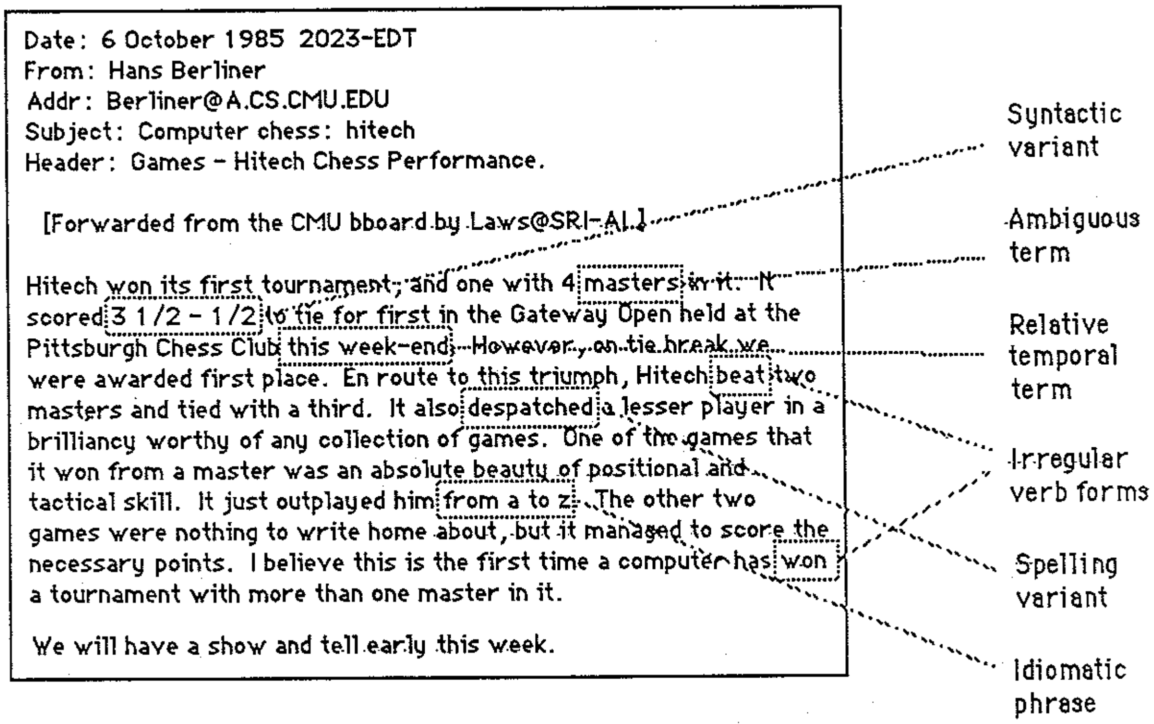
Fig. 3.7b: A representative query.

jubilant and informal style of the message is likely to lead to parsing problems: what does it mean to "despatch a lesser player with brilliancy?" Fig. 3.8 points up several more such problems.

The key to cutting through these problems lies in the word knowledge in the CODER lexicon. Robert Amsler has noted [AMSL 80] that word senses tend to cluster: they fall into short, bushy near-trees, the lower-order levels of which are formed of groups of words defined in terms of each other. Just such a situation occurs with two groups of terms in the message, one based on "win" and the other on "game" (see Fig. 3.9). This clustering serves two purposes: it serves to bring the clustered terms into the foreground of processing, and it aids in disambiguating words in the text. In the context of the Hitech message, there is no difficulty in selected definition #5 "a player of a game, esp. chess or bridge, who has won a specified number of tournament games" from among the 22 definitions for "master." Some of the other definitions in the cluster are presented in Fig. 3.10, using the Prolog syntax in which they appear in the lexicon. Based on this clustering, and on the occurrence of certain terms in the subject headers, a set of high-confidence terms can be abstracted from the text.

While this process is being carried out by experts in word morphology, English syntax, and the structure of the lexicon, other experts are simultaneously examining the message header. With the hypothesis that the document is of hard type **email message**, the types of the attributes "originator," "address" and "date/time of origin" can be used to determine the parsing strategies to be applied to the corresponding document fields: with the success of those strategies, the hypothesis that that is a correct description is strengthened. A frame can then be constructed following that description, and an interpretation formed and stored relating that frame to the terms abstracted (Fig. 3.11).

With this much in hand, the message can be retrieved in response to the query in Fig. 3.7b. The query terms "defeat" and "opponent" are both part of the "win" cluster: expanding the query during the retrieval process will pick up enough other words in the cluster to produce a close match with this document. Given more advanced computational-linguistic techniques, however -- and the ability to build representational structures from parsed text -- we can hope for something like the interpretation in Fig. 3.12. Here a "soft type" has been identified for the message, resulting in at once a simpler and more precise representation for the document's information content. Note that the knowledge representation structures handle this case with the same ease as the less sophisticated analysis.

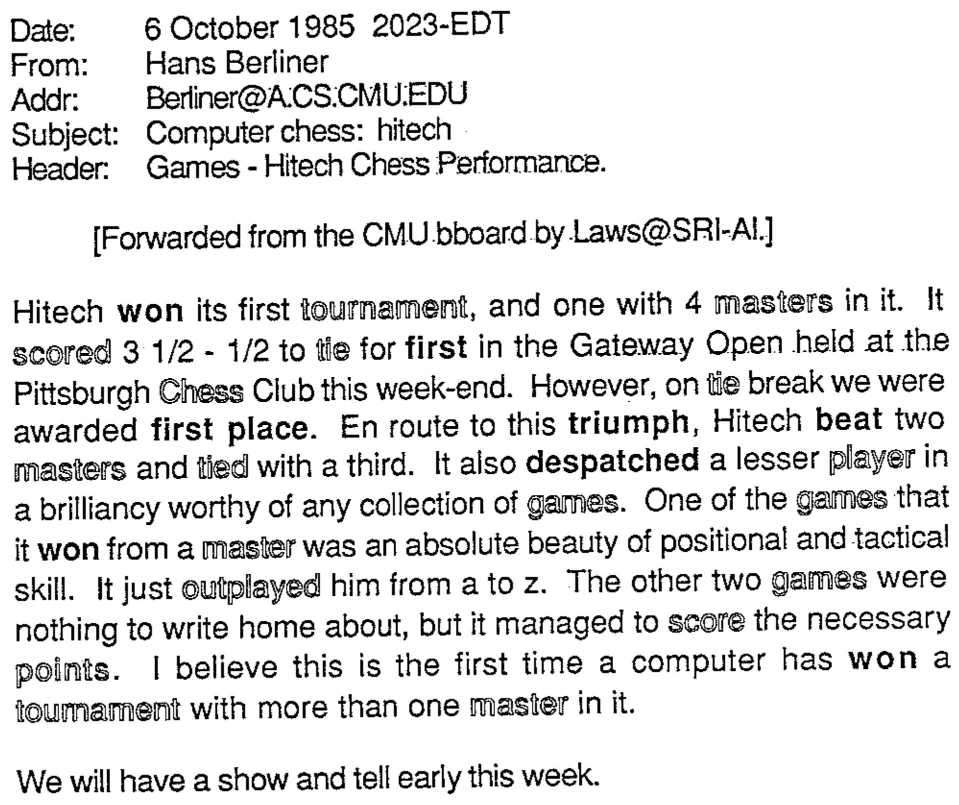

Date: 6 October 1985 2023-EDT
From: Hans Berliner
Addr: Berliner@A.CS.CMU.EDU
Subject: Computer chess: hitech
Header: Games - Hitech Chess Performance.

[Forwarded from the CMU bboard by Laws@SRI-AI.]

Hitech won its first tournament, and one with 4 masters in it. It scored 3 1/2 - 1/2 to tie for first in the Gateway Open held at the Pittsburgh Chess Club this week-end. However, on tie break we were awarded first place. En route to this triumph, Hitech beat two masters and tied with a third. It also despatched a lesser player in a brilliancy worthy of any collection of games. One of the games that it won from a master was an absolute beauty of positional and tactical skill. It just outplayed him from a to z. The other two games were nothing to write home about, but it managed to score the necessary points. I believe this is the first time a computer has won a tournament with more than one master in it.

We will have a show and tell early this week.

Fig. 3.8: Some problem areas in the text.



Date: 6 October 1985 2023-EDT
From: Hans Berliner
Addr: Berliner@A.CS.CMU.EDU
Subject: Computer chess: hitech
Header: Games - Hitech Chess Performance.

[Forwarded from the CMU bboard by Laws@SRI-AI.]

Hitech **won** its first tournament, and one with 4 masters in it. It scored 3 1/2 - 1/2 to tie for **first** in the Gateway Open held at the Pittsburgh Chess Club this week-end. However, on tie break we were awarded **first place**. En route to this **triumph**, Hitech **beat** two masters and tied with a third. It also **despatched** a lesser player in a brilliancy worthy of any collection of games. One of the games that it **won** from a master was an absolute beauty of positional and tactical skill. It just outplayed him from a to z. The other two games were nothing to write home about, but it managed to score the necessary points. I believe this is the first time a computer has **won** a tournament with more than one master in it.

We will have a show and tell early this week.

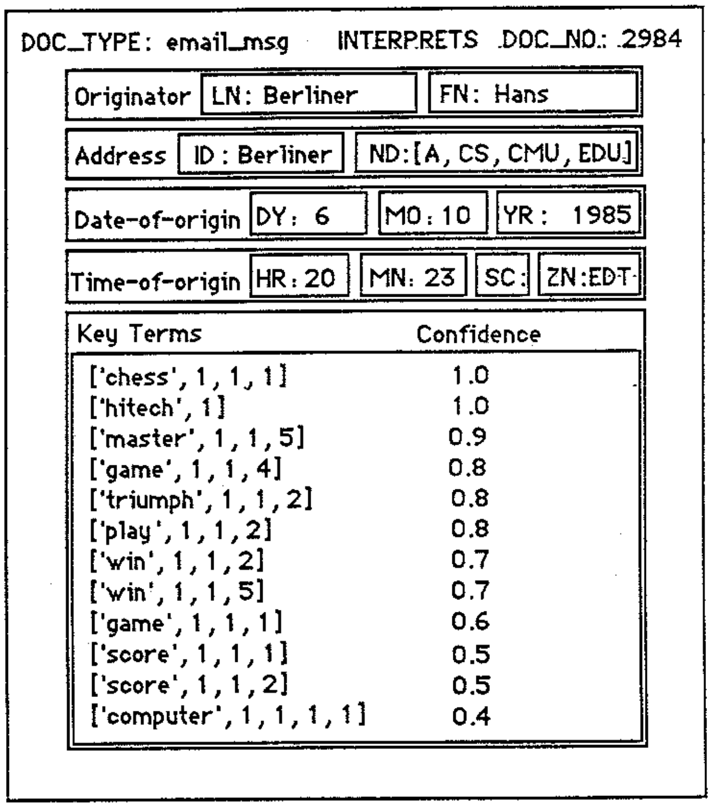
Fig. 3.9: Conceptual clustering in the document text. The game cluster is indicated in outline script; the win cluster in boldface.

```
c_HEADWORD(['win', 1]).
c_PLURAL(['win', 1], 'won').
c_POS(['win', 1, 1], vb).
c_DEF(['win', 1, 1, 1 ], '(intr.) to achieve first place in a competition', 1).
c_DEF(['win', 1, 1, 2 ], '(tr.) to gain or receive (a prize, first place, etc.) in a
        competition', 1).
c_DEF(['win', 1, 1, 5 ], 'to gain victory or triumph in (a battle, argument, etc.)', 1).
c_POS(['win', 1, 2], n).
c_CATEGORY(['win', 1, 2, 1 ], 'Informal').
c_DEF(['win', 1, 2, 1 ], 'a success, victory, or triumph'

c_HEADWORD(['triumph', 1]).
c_POS(['triumph', 1, 1], n).
c_DEF(['triumph', 1, 1, 1 ], 'the feeling of exhultation and happiness derived from a
        victory or major achievement', 1).
c_DEF(['triumph', 1, 1, 2 ], 'the act or condition of being victorious; victory', 1).

c_HEADWORD(['beat', 1]).
c_PLURAL(['beat', 1], 'beat').
c_POS(['beat', 1, 1], vb).
c_DEF(['beat', 1, 1, 11 ], 'to overcome (an opponent) in a contest, battle, etc.', 1).

c_HEADWORD(['dispatch', 1]).
c_VAR_SPELL(['dispatch', 1], 'despatch').
c_POS(['dispatch', 1, 1], vt).
c_DEF(['dispatch', 1, 1, 4 ], 'to murder or execute', 1).

c_HEADWORD(['out-', 1]).
c_POS(['out-', 1, 1], prefix).
c_DEF(['out-', 1, 1, 1 ], 'excelling or surpassing in a particular action', 1).
c_SAMP(['out-', 1, 1, 1 ], 'outlast').
c_SAMP(['out-', 1, 1, 1 ], 'outlive').

c_HEADWORD(['play', 1]).
c_POS(['play', 1, 1], vb).
c_DEF(['play', 1, 1, 2 ], '(tr.) to contend against (an opponent) in a sport or game', 1).
c_SAMP(['play', 1, 1, 2 ], 'Ed played Tony at chess and lost.').
```

Fig. 3.10: **Lexical relations relevant to identifying the conceptual clusters.**

Type-specific parsing of header fields:

Lexicon-aided identification of indexing terms:



DOC_TYPE: email_msg INTERPRETS DOC_NO: 2984

Originator	LN: Berliner	FN: Hans
Address	ID : Berliner	ND:[A, CS, CMU, EDU]

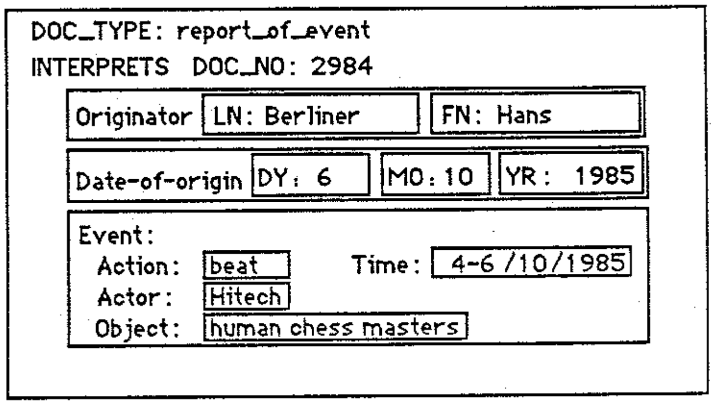
Date-of-origin	DY: 6	MO: 10	YR: 1985	
Time-of-origin	HR: 20	MN: 23	SC:	ZN:EDT

Key Terms	Confidence
['chess', 1, 1, 1]	1.0
['hitech', 1]	1.0
['master', 1, 1, 5]	0.9
['game', 1, 1, 4]	0.8
['triumph', 1, 1, 2]	0.8
['play', 1, 1, 2]	0.8
['win', 1, 1, 2]	0.7
['win', 1, 1, 5]	0.7
['game', 1, 1, 1]	0.6
['score', 1, 1, 1]	0.5
['score', 1, 1, 2]	0.5
['computer', 1, 1, 1, 1]	0.4

Fig. 3.11: Term-based analysis of document.

Recognition of "soft" document type:

Conceptual analysis of text:



DOC_TYPE: report_of_event
INTERPRETS DOC_NO: 2984

Originator	LN: Berliner	FN: Hans	
Date-of-origin	DY: 6	MO: 10	YR: 1985

Event:
Action: beat   Time: 4-6 /10/1985
Actor: Hitech
Object: human chess masters

Fig. 3.12: Content analysis of text determines "soft type" of document.

It should be noted that much more knowledge can properly be stored in the lexicon than what is shown in Fig. 3.10. Two other dictionaries, the analysis of which is currently underway as part of the CODER project, promise to yield information on verb patterns, including common prepositions and particles and the appropriate placement of modifiers and objects. Domain-specific knowledge can also be added to the lexicon in the form of semantic relations among the word senses proper to that domain. All of this knowledge can be represented in terms of facts about words in the language grouped into interpretations of word senses. These interpretations, together with growing knowledge bases of interpretations of documents and users, provide the CODER system with the factual world knowledge needed to intelligently pursue the tasks of information storage and retrieval.

4. Architecture

Everything should be done as simply as possible, but no simpler.

-- Albert Einstein

There are two problems implicit in realizing a system such as CODER. First, there is a problem of system *decomposition*. The preceding chapters (particularly Chapter 2) give some idea of the complexity of the system, both in terms of the number of knowledge sources expected and in the amount of world knowledge involved. Second, there is a problem of system *construction*. Over the lifetime of CODER, it will be used for a number of different experiments, involving modules and knowledge bases provided by a number of different experimentors. If this complexity is not to engulf and destroy the project, strong organizing principles will be required.

Fortunately, there are several problem characteristics that provide serendipitous assistance. First, both the task of document analysis and indexing and the task of document retrieval can be partitioned into a set of weakly interacting knowledge sources. Second, the tasks require only shallow plan trees. As will be seen below, a hierarchical plan space only two levels deep is deemed adequate for the problem. Third, the problem of information storage and retrieval is intensive, not in hypothetical (rule-based) knowledge, nor in procedural knowledge, but in factual knowledge -- knowledge about words, concepts, documents, users and so forth.

These characteristics have motivated our choice of a community of experts model, where each knowledge source is modeled as a different *expert* with faculties for low-level planning, and where global planning is handled by a separate *strategist* module. These experts are kept simple by providing non-inferential modules for knowledge representation and knowledge maintenance 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.

4.1. 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. 4.1). 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.

4.1.1. 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: its author, date of origin, and so forth. Experts in structured data recognition and document classification interact with experts in natural language

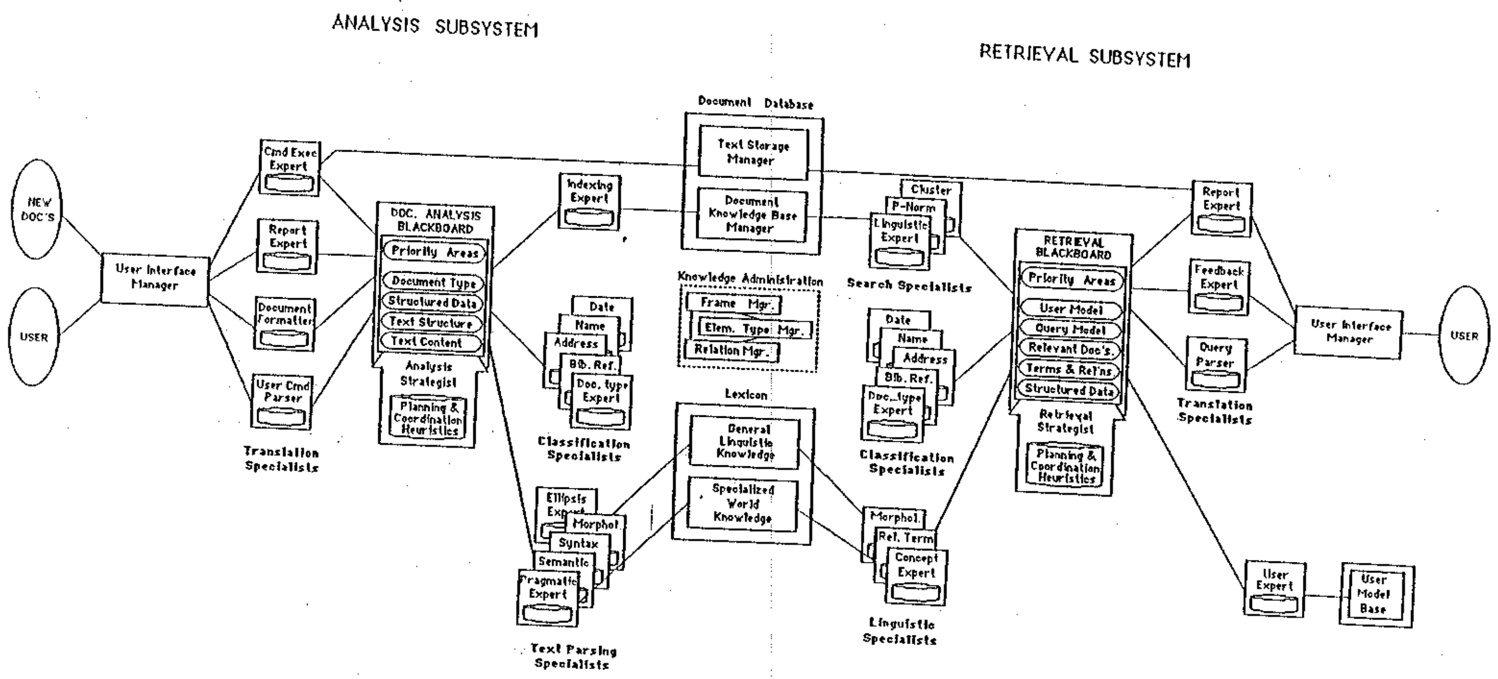


Fig 4.1: Overview of the CODER system.

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.

4.1.2. 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 it may 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.

4.1.3. 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 Spine components also 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. Each of the three levels of the factual knowledge representation formalism (see § 3.2) defines a *type manager,* which includes knowledge of and about the types of entities defined at that level. For instance, the *elementary* data type manager controls elementary objects such as **integers**, **characters**, **lists of lists of characters** and so forth, and the knowledge of subsumption relations holding among them. The *frame* manager controls semantically structured frame objects such as **documents, dates, sentences** and so forth; it also includes the functions to determine subsumption relations among frame types and matches among frame objects. The *relation* manager handles associative relation types such as **synonym of, described by**, and **is author of**, and has knowledge of such characteristics of these relations as reflexivity, transitivity and symmetry. The three domains defined by the current state of these managers (i.e., what distinctions they can make among objects known to the system) define the limits of knowledge available to the system.

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 analyzing and retrieving documents.

4.2. System Decomposition Principles

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. The object-oriented paradigm has been proposed several times as a natural organizing principle for AI: it has been used in the LOOPS environment [BOBR 83], as an organizing construct for logic programming [ZANI 84], and in the construction of knowledge-based systems [TOKO 84]. Dividing a knowledge-based system according to sub-domains of specialization also seems natural. Methodologies for discovering and exploiting the implicit groupings of knowledge in a domain have been explored by Cheng and Fu [CHEN 84] and by Froscher and Jacob [FROS 85].

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 a 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 [NAIS 85]. 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.

4.3. System Object Classes

Four classes of computational objects make up the basic building blocks of the CODER system (see Fig. 4.2). *Experts* capture the knowledge manipulation and inference aspects of the intelligent behavior the system is designed to exhibit. Each community of experts is provided with an active *blackboard,* which provides both a central communication area and an active planning principle for control. Factual knowledge describing the problem universe is maintained in *external knowledge bases*. These three classes of objects correspond to three logical levels in the tasks: a level of methods or tactics used to solve sub-problems, a strategic level of task planning, and a level of domain representation [STER 84]. Finally, a class of *resource managers* structures the interfaces between the system and its users. Objects of these four classes, together with the environment provided by the knowledge representation administrators and communication paradigms, make up the entire system.

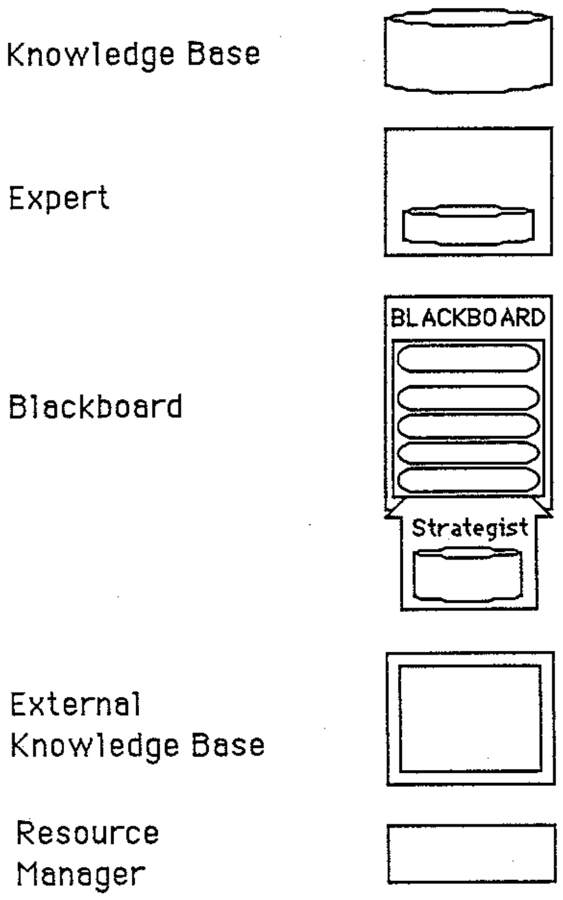


Fig. 4.2: CODER object classes.

4.3.1. Blackboards

A blackboard is an area for communication between experts (see Fig. 4.3). 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 *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 five-tuple: the **fact** hypothesized, the **expert** hypothesizing it, the **confidence** that the expert has in it, a unique **identifier** for the hypothesis (used in subsequent references to 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 [DOYL 77, DOYL 80] 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 [AIKI 80, RYCH 84].

The final component of the strategist is a scheduler for the tasks identified by the

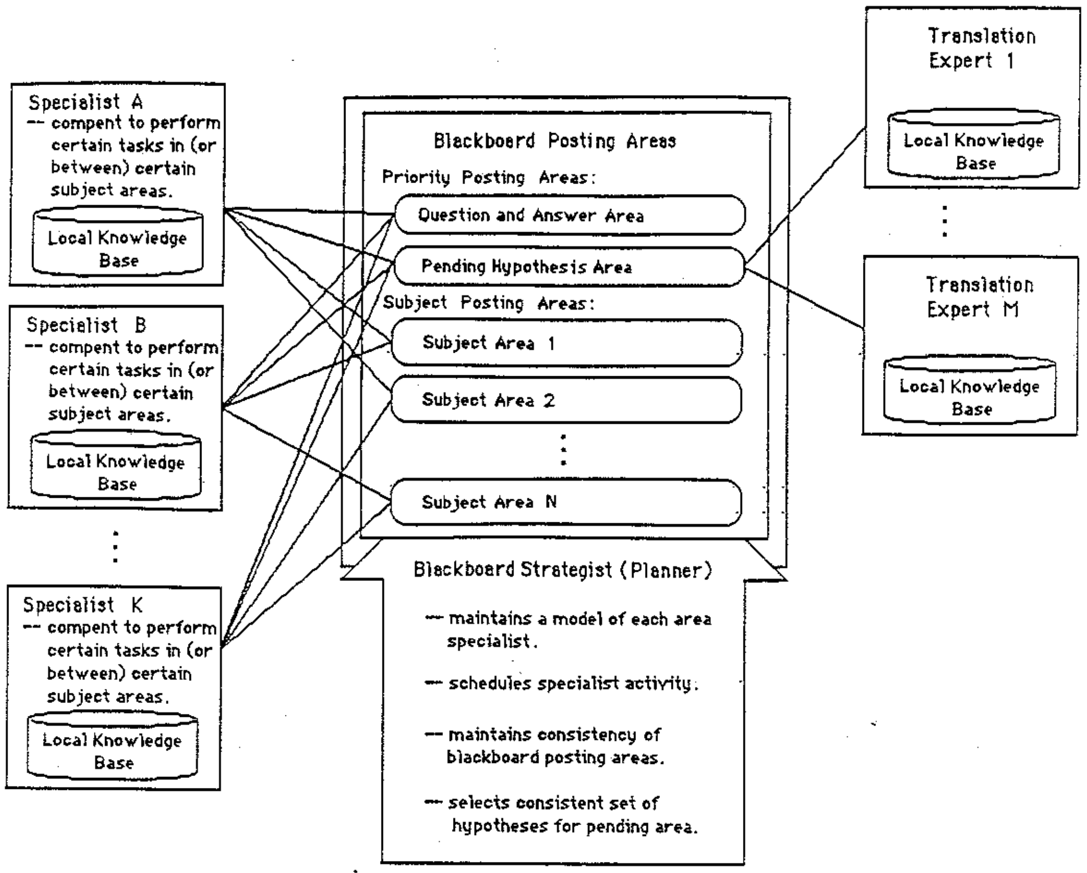

Specialist A
-- compent to perform certain tasks in (or between) certain subject areas.
Local Knowledge Base
Specialist B
-- compent to perform certain tasks in (or between) certain subject areas.
Local Knowledge Base
Specialist K
-- compent to perform certain tasks in (or between) certain subject areas.
Local Knowledge Base
Blackboard Posting Areas
Priority Posting Areas:
Question and Answer Area
Pending Hypothesis Area
Subject Posting Areas:
Subject Area 1
Subject Area 2
Subject Area N
Blackboard Strategist (Planner)
-- maintains a model of each area specialist.
-- schedules specialist activity.
-- maintains consistency of blackboard posting areas.
-- selects consistent set of hypotheses for pending area.
Translation Expert 1
Local Knowledge Base
Translation Expert M
Local Knowledge Base

Fig. 4.3: Blackboard / strategist complex, showing mapping of experts in the immediate community to blackboard areas. Not shown are the strategist-to-expert interface.

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 commands to the experts in the blackboard community. The scheduler may call for an expert to perform a specific task or to attend to the current state of a given area of the blackboard, or it may simply call for it to awaken and attempt all the tasks it knows. 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.

4.3.2. 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.

The canonical expert consists of a communications interface, a local knowledge base, and an inference engine (Fig. 4.4). The interface provides for communication with the blackboard and with an optional external knowledge source, such as a resource manager or external knowledge base. The local knowledge base contains the particular expertise necessary for the proper execution of the expert's tasks; the inference engine is chosen to best execute those tasks. 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. Recent research supports the view that it is possible to build engines that cover a broad range of problems without falling into the computational trap of general inference. Examples include the structure-and-function engines pioneered by Randall Davis [DAVI 82] and the heuristic classification engine designed by William Clancey [CLAN 84, CLAN 85]. Other experts may not use inference per se, but may interpret knowledge written in a procedure representation language such as advocated by [ERMA 84] or [MILL 86]. It is, in fact, entirely possible that some experts, such as the morphology expert, may be written entirely in a procedural language such as C++. This approach, however, is discouraged, as the semantics of knowledge in these languages is difficult or impossible to separate from the semantics of its interpertation.

4.3.3. External Knowledge Bases

An external knowledge base (or "fact base") is an object for storage and retrieval of factual world knowledge (see Fig. 4.5). The Document Knowledge Base, the Lexicon,

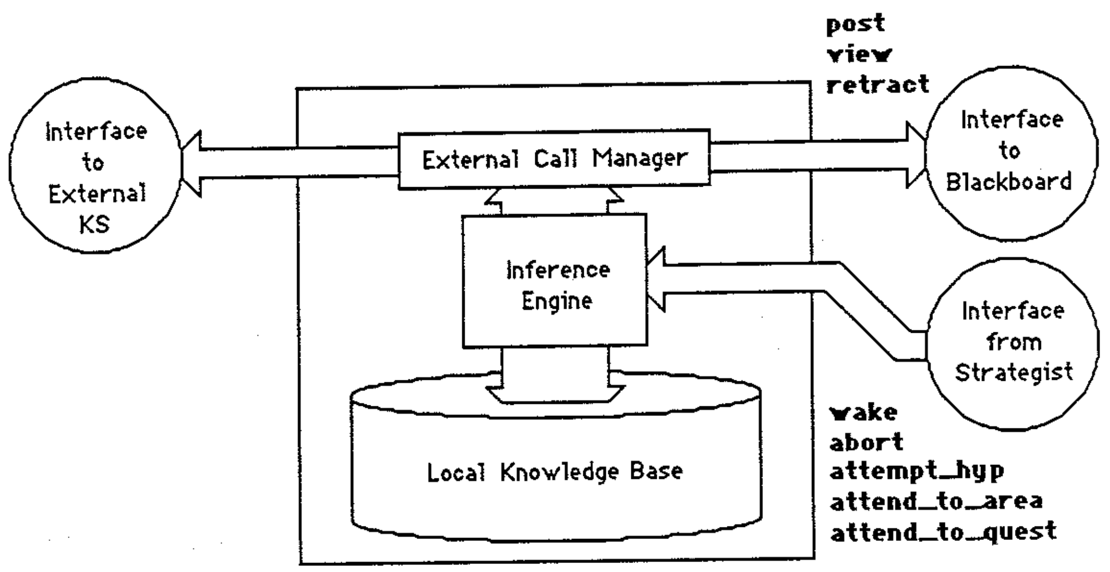


Fig. 4.4: Canonical CODER expert, showing internal structure and functionality of interface with blackboard / strategist complex.

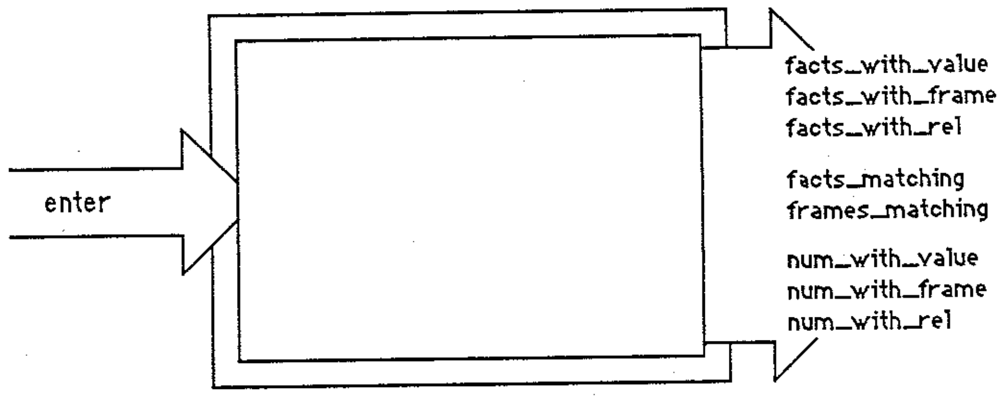


Fig. 4.5: The functionality of an external knowledge base. An EKB has no implementation-independent internal structure.

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 composed of high-level knowledge and meta-knowledge.

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, an external knowledge base can be called with a frame object to find frames referenced in the base that match the frame. Finally, 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.

4.3.4. 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 consensus 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* (see Fig. 4.6). 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

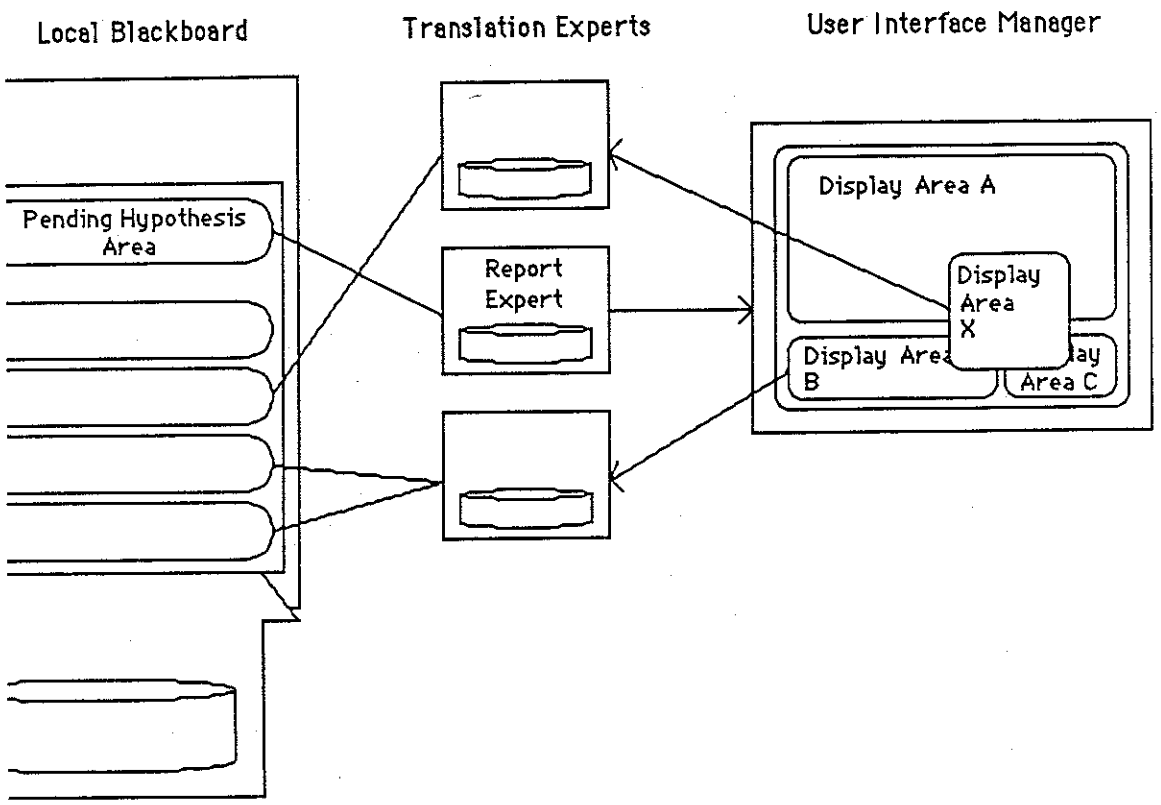


Fig. 4.6: User interface subcommunity.

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.

4.4. Engineering Considerations

Faced with the problem of providing hardware support for artificial intelligence, Daniel Hillis has argued for the use of fine-grained concurrency [HILL 85]. Such concurrency, however, reflects a model of an AI system as a single functional block, with the knowledge required for its intelligent operation distributed relatively evenly throughout. In contrast, the CODER system is made up of relatively few, functionally discrete modules, each specialized both by the type of functionality it provides to the system as a whole and by the problem area in which it has knowledge. This model has encouraged us to implement based on a coarse-grained concurrency, where each functional module is resident on a single virtual machine, and the system is distributed across a few physical machines, each typically providing support for a few functional modules.

This coarse-grained model has the advantage of conceptual simplicity. Each module can be conceived of as a single process (or set of processes) executing in isolation on a single machine. The process is begun when the module receives a message and is (at least conceptually) allowed to run to completion without any required synchronization with processes outside the module. Similar models of concurrency have been proposed by Bernard Witt [WITT 84] and James Gray [GRAY 86]: both advocate the model as efficient in machine use and easily comprehensible to humans.

4.4.1. Communication Model

Communication among modules occurs as a variant of the client / server relationship. Each class of system modules is defined to respond to certain calls by performing certain tasks. During the execution of a task a module may, of course, request service from another module, but (barring deadlock) each task is guaranteed to complete with the return of a reply to the client module. Requests to a module are cued and serviced serially, obviating the need for concern with system concurrency at the level of the internal workings of a module.

Deadlock in CODER is avoided through a hierarchical calling structure (see Fig. 4.7). All processing is initiated by the task strategist, which calls upon translation experts and specialists to execute sub-tasks involved in the system task currently under way. These calls function as process forks: control returns to the strategist as soon as the expert accepts the task, or returns as a call failure if the expert cannot accept. Experts also signal the strategist when they have completed tasks, allowing the strategist to abort their execution if they cannot finish within a reasonable time (if, for instance, they become entangled in an endless tree of inferences). This reporting also allows the task state to be recovered and experts restarted in case of a system crash, and permits a deadlock detection and breaking scheme to be added to the strategist if needed. This eventuality is not expected, however. Physical deadlock cannot occur in the system, since there are no loops in the calling structure. Conceptual deadlock can occur when two experts wait for answers to mutually dependent questions, but this situation can be dealt with as a special case of plan failure, not significantly different from failure for lack of factual knowledge. Posting a question constitutes the end of a task and assimilating answers to the question is considered a separate task, so the resources of an expert waiting for a question to be answered are free for the execution of other tasks or the restarting of the task that produced the question.

Calls to the lowest level of the hierarchy -- the external knowledge bases, the resource managers, and the passive principle of the blackboard -- are considered atomic transactions. They are queued, executed serially on a first-come-first-served basis, and are all guaranteed to terminate. The complexities of shared blackboard transactions [ENSO 85] do not occur in CODER since each task is executed by a single agent. Expert activity on the blackboard is reported to the strategist, where is becomes part of the session history and may motivate task scheduling and strategic planning. Strategist activity, therefore,

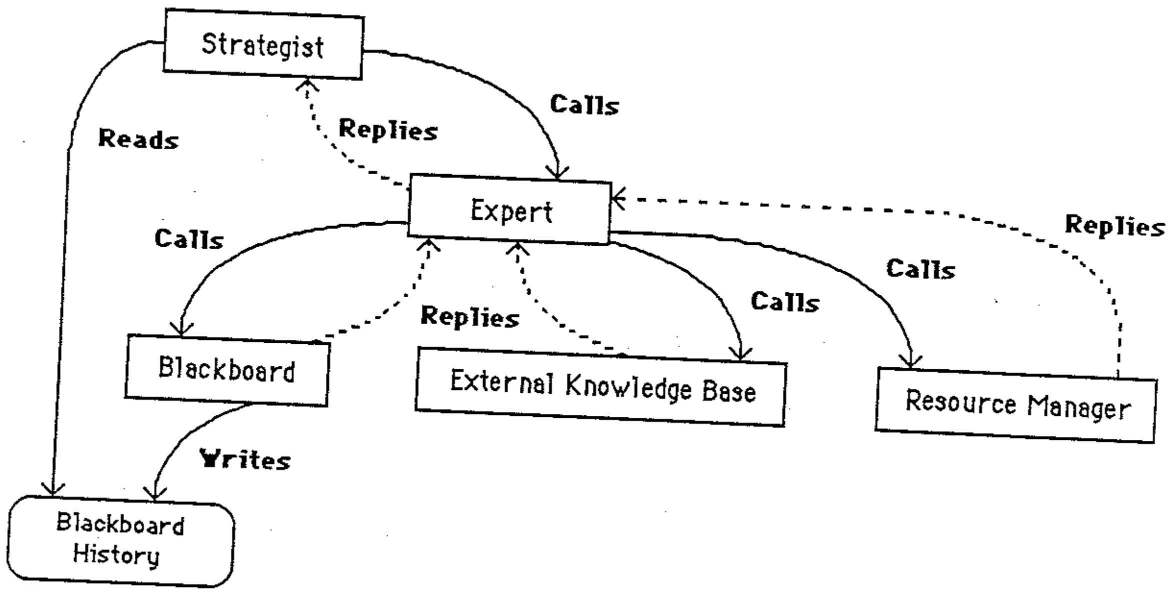


Fig. 4.7: A slightly simplified version of the calling hierarchy in a CODER community of experts.

may be triggered by events on the blackboard, by experts completing their assigned tasks, or by the passage of time. Control, however, returns to the strategist immediately upon awakening an expert, and eventually to an expert from a call to a passive module. This ensures a high degree of concurrency in the system at the level of task-oriented knowledge manipulation while maintaining a strict control at the level of goal-oriented planning.

4.4.2. Call Model

Inter-module communication in CODER is patterned after rule invocation in Prolog. When a client module requires service, it executes a predefined **ask** predicate, passing it the name of the desired server and one of the external calls defined on the server's module class. The arguments of the call will either be unbound variables in the source language of the client module, sentences in the CODER factual repreresentation language, or *skeletal facts*. A skeletal fact is a sentence in the factual representation language where CODER variables have been substituted for one or more of the arguments of some relation, typically the head relation. These variables are not variables in the source language of either the client or the server, but particular syntactic forms that can be recognized by both without invoking, for example, a Prolog unifier. Skeletal facts are used in calls to external knowledge bases, experts and blackboards to refer to a range of acceptable facts. For instance, a question is posted to the blackboard as a skeletal fact describing what knowledge the questioning expert needs in order to complete processing.

CODER variables used in external calls are not bound. Variables in the source language of the client module, however, may be. An execution of the **ask** predicate in a Prolog client, for instance, will either fail (if the server cannot accomplish the task requested) or will return in the fashion of a local Prolog clause, with potentially some of the Prolog variables used in the call bound to knowledge structures provided by the server. Likewise, in a C++ client, the knowledge structures built to pass to the server will not be overwritten, but some local variables may.

Failure of an external call is not typed. Unlike the return-code facilities provided by languages like C, where the value returned by system functions can be coded to describe the reason for failure, the failure of an external call in CODER, like the failure of a clause in Prolog, is simply a failure. This is a weakness of the model, as a client cannot tell on call failure whether the call was inconsistent with the server's knowledge base, whether it was syntactically incorrect, or whether the server simply crashed during execution through no fault of the client. The CODER module classes have been carefully designed so that failure of a syntactically correct call to a functioning module can only mean one thing for each call, and programming conventions for the CODER project attempt to compensate for this weakness by making good use of the UNIX **stderr** channel, but the weakness remains, and will remain in any system modeled on the Prolog call.

External calls differ from ordinary Prolog predicates in that they cannot be resatisfied. The most natural way to specify the functionality of, for example, the external knowledge base class in Prolog would be to describe it as returning a single fact on each call and an additional fact each time control backtracks across the call until either the rule calling the knowledge base succeeds or the knowledge base runs out of matching facts. This model of execution would integrate well with standard Prolog, would place less of a burden on servers than the current model, and would keep communication overhead down. Unfortunately, it would require the system to either commit a server to a client until the client could guarantee that no backtracking would occur (either because the goal containing the service request had succeeded or because a **cut** had been passed) or to violate the principle of atomicity of transactions. The first alternative re-introduces the possibility of deadlock, as a single rule in a client may include external calls to more than one server; the second introduces pure chaos. If servers had to commit to clients until the client finished an entire rule, then it would be possible for two experts, each of whom need both the Lexicon and the blackboard, to each hold one indefinitely while preventing each other (and the rest of the system) from calling the other. If, on the other hand, servers are freed in between backtracking calls, then it possible for another client to call the server and change its state in such a way that the new bindings are inconsistent with the old, that the new call fails, or, worse, that it never fails, violating the requirement that every system call terminate. Deadlock is even possible here, too, with two clients flipping the state of a server back and forth between a state that causes one and a state that causes the other to fail.

To avoid these pitfalls, the **ask** predicate is built as a function that cannot be resatisifed, and external calls in the system are specified to always return complete sets of possible bindings. Thus, the blackboard returns *all* hypotheses in an area, and an external knowledge base returns the full set of facts matching the object with which it was called. This allows every external call to be regarded as an atomic transaction and keeps the system analytially tractable. As noted above, it does not prevent conceptual deadlock in the system, but it means that if such behavior does manifest, it can be analytically recognized and resolved.

4.4.3. System Model

The idea of a distributed expert system where different experts or knowledge sources reside on different physical processors is not new. Dave McArthur [MCCA 82], Michael Huhns [HUGH 83] and Bonnie McDaniel [MCDA 84] have all advocated distributed expert systems based on a what might be called a *network* model, where all the modules in the system have the same status and more or less the same structure, and where communication is possible between any two modules. In contrast, CODER is organized on a *hierarchical* model, where communication between and structure of system modules is determined by their function in the system. This model is closer to that proposed by J. Robert Ensor and John Gabbe [ENSO 85], although control in their system is less centralized than in CODER.

The CODER model has the advantages of a close match to the physical resources, the computational resources, and the knowledge resources of the target environment. In a physical environment composed of a small number of large computers, external knowledge bases are distributed where there is fast memory to serve them. Experts are allocated by their place in the calling hierarchy: either on the same machine that houses their most commonly used knowledge source or close to the strategist that is invoking them. A new copy of the blackboard / strategist complex is created for each new session, typically on the machine from which the session was evoked, and can make use of specialist experts wherever they reside (Fig. 4.8). In the computational environment, use of language-independent knowledge representations and call predicates allows modules to be built in either declarative or procedural languages, or to be built around interpreters for more specialized representations.

The hierarchical structure of the model is determined by the knowledge resources used by the system. The top of the hierarchy is exemplified by a simgle unified strategist interpreting plan knowledge. The second level is composed of the experts, which interpret knowledge of methods to accomplish the tasks set them by the planner. This is the level at which concurrency is most useful: at the level of the planner, concurrency is an unnecessary complication; it is at the level of assembling and manipulating knowledge that

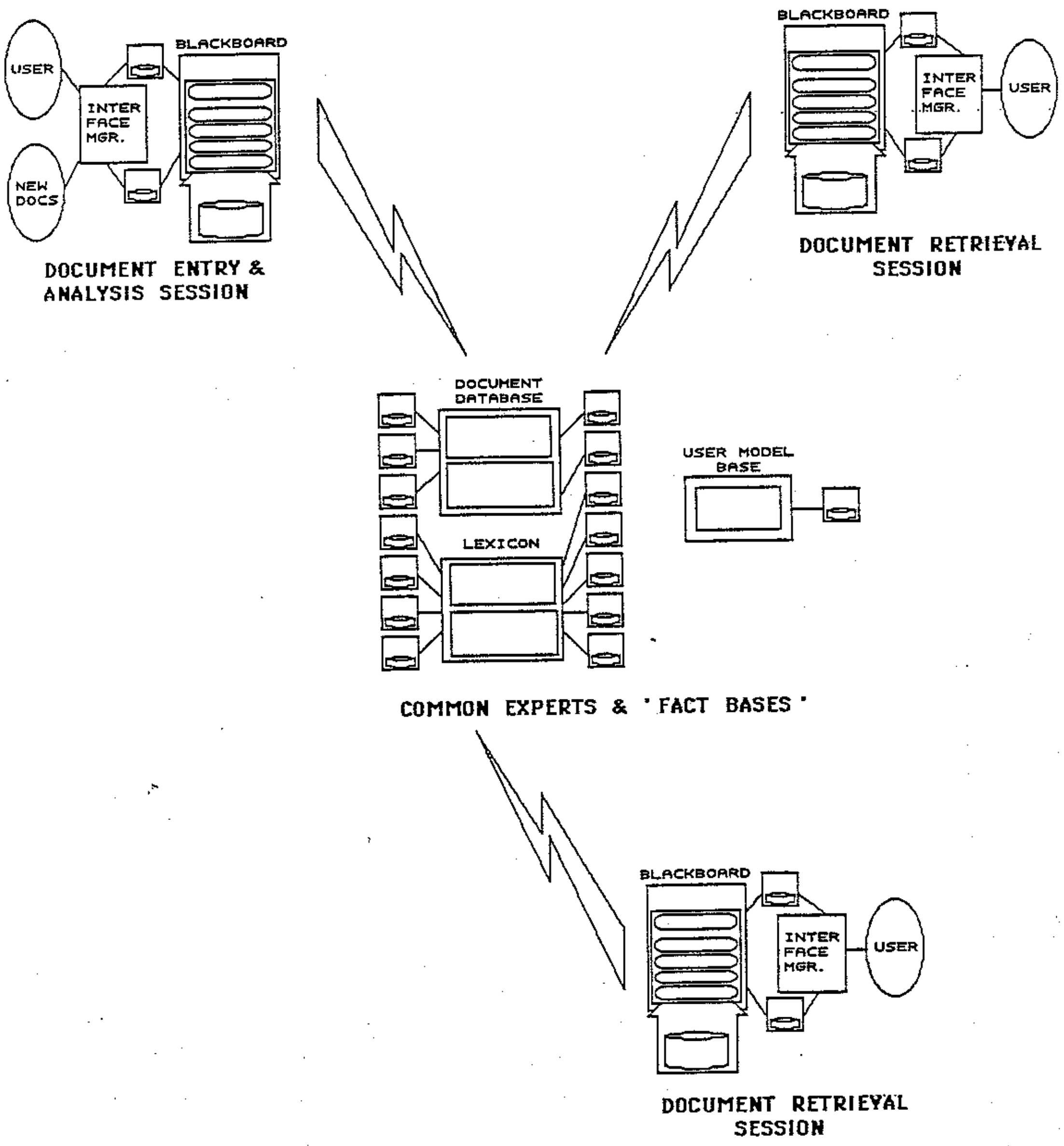


Fig. 4.8: Overview of system operation in a distributed environment.

the system can benefit from many things being done at once. At the bottom level of the calling hierarchy are modules whose purpose is handling factual knowledge. Concurrency is useful at this level in that it allows resources that cannot be duplicated to be shared among the clients that need them.

The architecture of the CODER system thus ensures that concurrent processing will be an asset to the system, rather than a complication. It does this by following the organization of the knowledge used in the system, so that concurrency occurs both between logical levels of the knowledge used to solve a problem and within those levels where concurrency occurs naturally. Decomposition of the system task within each level procedes by following the types of knowledge used in each part of the task and allocating cohesive sub-tasks to separate experts. In this way, a flexible structure is defined within which a broad variety of knowledge-based tasks can be executed. This structure provides the environment for the CODER project, and serves as the testbed within which a variety of knowledge-based techniques can be evaluated for their impact on the tasks of information storage and retrieval.

5. Organizing Principles

At my present rate of working I produce about a thousand digits of programme a day, so that about sixty workers, working steadily through fifty years might accomplish the job, if nothing went into the wastepaper basket. Some more expeditious method seems desirable.

-- A.M. Turing, on the possibility of AI.

In this chapter, we examine the organizing principles that have proven most useful in the design of CODER. We argue that these are important principles in the design of expert systems, and that CODER derives important benefits from having been designed in accordance with them. The chapter concludes with a discussion of extensions that might be made to the CODER environment in order to make it a better environment for developing production-quality expert systems.

5.1 The Principle of Modularity

The concept of modular decomposition marks a key point in the transition to regarding a program, not as an individual work of art, but as a structure that can be engineered. The applications of this single concept are both broad and deep, ranging from project control, where proper decomposition not only allows more labor to be brought to bear on a given problem but allows better allocation and tracking of the labor as well, to system design, where concentration on the functional characteristics of module boundaries becomes as important and enlightening a part of the design process as specification of the internal behavior of the modules themselves, to coding, where an upper limit on procedure size has been shown to benefit code production, readability and correctness. This lesson, now over a decade old in computer science in general, might be considered almost too familiar to notice ... until one is faced with an expert system such as MYCIN, where several hundred production rules fit seamlessly into a single edifice and where an upper limit to the complexity of the system is dictated by problems of unplanned rule interaction.

This upper limit is particularly irksome because of the large number of expert systems constructed in the last few years, all of about the same magnitude and none reaching a level of expertise above, say, a graduate student in the appropriate field. These 'novice systems' clearly indicate that the technology is adequate to their domains, yet fall short of being genuinely useful programs through their failure to cope with the full complexity of the subject area. Their primary reasons for failure would seem to be threefold. First, knowledge interactions bog down the inference engines. As more knowledge is added to the system, each situation presented to it triggers more interpretations, resulting in large numbers of blind alleys being explored and, at worst, in non-terminating inference chains. Second, most expert systems are built upon a single inference engine (usually either a backward- or forward-chaining rule interpreter) and even in the most straightforward domain not all types of problems encountered will submit to the same method of solution. The common responses to this phenomenon -- adding 'hooks' where arbitrary procedures can be added as needed or using 'inside knowledge' of the functioning of the engine to trick it into handling, for instance, a production rule as a procedure -- only compound the first problem, resulting in opaque code and fragile systems. Finally, the complexities of modeling a real-world domain are typically too great for a single programmer or small team to handle, while the current state of technology in knowledge-based systems makes it difficult for many minds to work on a single system. Faced with the above problems, the intricacies of real-world knowledge, and the need to get the system 'out the door,' developers have too often been willing to settle for a working, but unsophisticated system.

It is clear that if one is to construct a system involving several thousand rules some form of organization must be adopted. It is not clear what that form should be. The domains in which expert systems are appropriate are precisely those, such as information retrieval or natural language understanding, where we have some knowledge of how solutions are to be brought about but no rigorous understanding of the best way to the best solution. Domains that we understand in a strong manner can be approached algorithmically; domains where we have no understanding we have no business attempting to automate. Expert systems technology has its appropriate place in the areas in between, *but these are precisely the areas in which we cannot predict the organization of the system* *beforehand.* For an expert system, the evolutionary paradigm of system development is not only desirable, but thoroughly appropriate. How, then, are we to organize a system, the project of producing the system and the code that realizes the system, while allowing the system structure to change as our understanding of the problem changes?

This problem is raised in an acute form in CODER, both because CODER will require the work of many programmers to build and because, as an experimental system, it will have to be torn down and rebuilt, not one, but many times. Modularity is essential for CODER both to ensure the correctness of its realization and to isolate the sections modified during different experiments. And the design described in Chapter 4, with its community of experts and external knowledge bases surrounding distinct blackboards, is clearly modular. In order to understand how this modularity can be accomplished naturally in an expert systems context, however, it is necesary for us to consider the impact of the nature of knowledge-oriented programming on software design and engineering.

5.2. Knowledge-Driven Design

The key observation in resolving this dilemna is that the programming of an expert systems is concerned less with the building of elaborate control structures than with the encoding of knowledge. Thus the organization of an expert system at any point in its development will be determined by the coupling and cohesion of the knowledge entered up to that point. The system as a whole, of course, is more than a collection of knowledge: it must include facilities for storing that knowledge, for performing inferences on it, and for communicating both within the system and with the outside world. These facilities, however, are to a large extent independent of the organization of the knowledge included in the system. If we can design generic facilities that will work regardless of the knowledge we add to the system, then we can reconfigure the system in a knowledge-oriented fashion as the development process continues.

Both psychological experimentation and our own introspective intuitions support the thesis that humans store knowledge in clusters [MINS 86]. In addition, studies by Cheng and Fu [CHEN 84] on an expert system knowledge base of production rules have shown that comparing the rules with a reasonable distance metric reveals a structure of loosely coupled cohesive clusters. Knowing that the clusters are there, however, does not bring us very far, especially since in view of the evolutionary methodology that we consider appropriate here, we cannot expect ahead of time to have a good idea of just where they occur. In particular, besides the problem of recognizing knowledge clusters as we encounter them, we are faced with the dual problems of properly encapsulating them in a fashion that will allow us the best computational efficiency, and properly separating them in a way that does not do damage to the weak but vital interdependencies that exist among clusters.

The approach taken in the CODER environment is, of course, to distribute the knowledge of the system among a community of experts. This helps to ensure efficiency because each expert can utilize the inference engine best suited to the knowledge in its cluster. It is, in fact, possible both to draw on a set of generic inference engines and to hand-build engines which will work with maximum efficiency on given local knowledge bases. Obviously, this second approach is only appropriate in the later phases of system engineering, during which the content of the local knowledge bases have somewhat stabilized. The weak interaction of the knowledge clusters is then modeled with the blackboard-based communication paradigm. The discipline of communicating through hypotheses makes it possible for experts to share knowledge without sharing inference chains.

Thus the community-of-experts organization provides a criterion for discriminating among knowledge clusters beyond the simple (and unenlightening) principle "clusters should be broken up when they become too large." Clusters should also be broken up when they require different kinds of inference. The more precisely this can be done, and the narrower each expert's type of knowledge can be made, the more efficiently can the local inference engine be designed. Fig. 5.1 lists the inference techniques that we have recognized as necessary for the initial CODER system; others may be added as our understanding of reasoning and knowledge increases. In addition, thinking of the knowledge as the property of experts, and remembering that an expert is defined by the set of tasks that it can perform, makes it possible to derive a principle of cohesiveness: a piece of knowledge belongs to an expert when it is useful for performing the expert's tasks.

Distributing the knowledge among the experts in a community, however, makes it necessary for the experts to communicate "on the knowledge level." In other words, it is not enough for the experts to be able to pass data back and forth: they must be able to communicate in statements compatible with each other's local knowledge bases and inference processes. Thus, although the internal representations of a classification expert with local knowledge about classes of descriptions and an inference engine specialized to follow frame hierarchies may not be usable by an expert based on forward-chaining production rules, the experts must all communicate through a common knowledge representation language. In the CODER environment, this language is supplied by the factual representation formalism.

The factual representation language was developed to be sufficent to store all the knowledge that the system requires about the world of information-bearing documents and information-requiring users. It provides facilities for naming (via elementary data types such as the integers and atoms), for describing (via typed frames), and for noting accidental features of (via factual relations) entities in that world. This fundamental level of concrete description exists primarily as a formalism for storing representations of the world and presupposes no particular form of inference except for the ability to compare frames for matching and relations for isomorphism. Statements in the language, however, can also serve as premises for rule-based inference, (partial descriptions of) states for state-space reasoning, or individual actions in a procedural context. This means that we can use the same language for representing the hypotheses used by the experts to communicate with each other that is used for storing knowledge about the world.

This commonality of function should be no surprise in the context of the CODER blackboards, where all knowledge posted relates to some possible state of the world. In HEARSAY, of course, this would not have been true, since individual experts (knowledge sources) could also post information about tasks in the problem-solving space that needed to be done. In a CODER blackboard, however, the process of inferring new tasks from the state of the system's knowledge about the world is handled within the blackboard strategist. Again, this is a knowledge-oriented process, and is accomplished using knowledge structures and an inference engine specialized for the purpose. The need for tasks, though, is motivated by the state of the system's knowledge at any point about the problem under consideration. This state is not only representable in the factual representation langauge, but we submit is represented precisely by the set of statements on the blackboard at any given time. Finally, note that the language will also serve to represent tasks in the strategists' messages to the experts. This is trivially true in the cases where the message is “attempt hypotheses of such-and-such form” or “consider all hypotheses of such-and-such form currently on the blackboard,” but is also true, with a slight semantic shift, in the general case. In this case, the outermost relation of the statement (the head relation) is treated as a verb specifying the action to be performed and the arguments of the head relation are treated as parameters of the task.

The last two paragraphs point up the power of designing an expert system around a flexible language for representation of knowledge about the problem universe. It has been argued above (§ 3.3) that the factual representation language has general utility as such a representational system. In any case, however, we can elicit three requirements that must be satisfied by the fundamental representation language of any expert system. First, it must be adequate, both metaphysically and epistemologically [MCCA 69], to the problem universe under consideration. The CODER FRL is adequate to the relatively well-behaved domain of document analysis and retrieval: a more powerful language might be needed for a domain involving primitive physics or complex temporal relationships. Second, it must not be prejudiced toward any particular type of inference, but usable in any of the types required by the system. And third, it must be computationally tractable. As can be seen from the above, the operations provided for manipulating representations in the language are in constant use throughout the system. Thus it is vital that these operations be performed as efficiently as possible. Unfortunately, the types of structures most used in knowledge representation -- frames and semantic nets -- are both intractable in the general case (see [SOWA 84] for nets and [BRAC 84] for frames). If these formalisms are to be used, they must be restricted in some way to make computations on them reasonable.

5.3. The Discipline of Strong Typing

A language is called *strongly typed* when user-defined computational obects in the language must be given a type, and the types of values assigned to those objects are checked for correspondence with the declaration, an error occurring if the types do not agree. Strongly typed conventional languages include Pascal, Ada and (to a lesser extent) C: provisions are generally made to allow “user-defined” types (actually restrictions of more primitive types already part of the language). Proponents of strong typing argue that such requirements force programmers to consider clearly the function of variables and the limits between which they may vary. Type violations can expose errors, more efficiently in that they can generally be caught at compile time. Furthermore, type declarations of function and procedure parameters, particularly those using user-defined types, can be an important vehicle for communication among programmers on a team. Type constraints have also been used in proofs of correctness and in verifying adherence to specifications. There are thus sound engineering advantages to having some form of typing available, at least during the program development phase, in an expert systems environment as well.

Besides considerations of control and correctness, type checking can be useful in increasing execution efficiency. Providing type definitions is said to increase the execution speed of Turbo Prolog™ code and to improve processing time of representation structures in PEARL [DEER 81]. Although it has not been possible to extend a strong typing throughout the CODER environment, the principle has been used to advantage in the CODER knowledge representation complex, and particularly in the frame system. Recall that frames in CODER are defined as sets of named slots, each of which is filled either by elementary data items, by frames, or by lists or sets thereof. The relevant BNF is:

```
frame        ::= frame_name  { slot_name:  slot_filler }*
slot_filler  ::= frame *
               | elementary_item *
```

If the slots that can occur in any frame and the items that can occur in any slot are completely unconstrained, then we have Marvin Minsky's original frame system [MINS 75]. Frames in CODER, in contrast, are divided into frame types and frame objects. The frame types are required to be defined in terms of other frame types (except for **frame_bottom**, the root frame type with no slots) and to have new slots specified as to the permissible types for their fillers. The frame objects are required to be instances of a recognized frame type, and to agree in all their slots with the types declared for that slot in that frame type.

In a groundbreaking study [BRAC 84], Brachman and Levesque showed that small changes in a frame-based language can make large differences in the tractability of decision problems for sentences in the language. In particular, they considered the problem of frame subsumption in a system which differed from that adopted for CODER in that it made no ontological distinction between individuals and classes. Frame subsumption in the CODER system, where it is defined only among frame types, is a computationally trivial problem: since frame types are required to form a semilattice, subsumption can be determined in O(log m) time, where *m* is the number of frame types in the system. The problem in the CODER system that corresponds most closely in terms of computational difficulty to Brachman and Levesque's, is frame matching among the knowledge objects in the external knowledge bases.

In a knowledge base including *n* frame objects, some may occur as slot fillers for others, but all are in the final analysis grounded in elementary data objects. This is a consequence of the requirements on facts -- that they be finite, grounded instances of relations -- and assures us that, while the matching operation is defined recursively, the recursion must terminate. In the worst case, it is possible that the recursion may run through every frame in the knowledge base (in the worst case, through every slot of every frame in the knowledge base) and thus will add a factor of O(n) to the complexity of frame matching. Actually, the recursion factor will be O(kn) where *k* is the maximum number of slots per frame in the knowledge base, but we will assume here and in the remainder of this analysis that k « n. We are guaranteed, however, that the recursion will end, so that it is sufficient to consider what must be done at a single step. We will also assume that any two elementary data objects can be compared in O(1) time. This is not strictly accurate, as elementary objects may include lists and sets, but again, we expect that the complexity of comparing these sets and lists will be trivial compared to the number of frame objects in the knowledge base. We will consider two cases: the untyped case, where any frame may have any slot attached to it, and that slot may take any value, and the typed case, where each frame object is identified by type and each frame type constrains the slot signature of the object. In both cases, we will consider the basic matching operation to be this: a frame object is provided to the knowledge base and the knowledge base must find all frame objects that match it. This is the function **frames_matching** provided by the definition of a CODER external knodge base; note that it is also an operation that must be performed in the **facts_with_frame, facts_with_rel** and **facts_matching** functions, the latter two in the case where a frame object is provided as part of the skeletal relation or fact.

In the untyped case, the incoming frame must be matched against all frames in the knowledge base. We can speed this up by assigning each slot name in the entire knowledge base a unique identifier and lexically ordering the slots in each frame description. Then for each slot in the incoming frame we can get a set of the frames with the same slot (though not necessarily the same slot filler) in O(1) time. Let us assume that the frames are lexically ordered as well; we are still left with the task of finding the frames that have (at least) all the slots in the incoming frames, which requires the union of k (ordered) sets. Comparing the slot values in order to discard non-matching frames is also an **O**(n) operation, but this is additive with the complexity of the union operation. Thus each step of the recursion is of **O**(n) and the process as a whole is **O**(n^2).

In the typed case, by contrast, the incoming frame can be constrained to a point on the frame type lattice. This point, it will be noted, is the type of the frame object only if all the slots of the object are filled. Rather, the particular combination of filled slots in the frame object determines a weakest (most general) frame type for the object, such that all matching frame types must be stronger than that type. Call this the *weakest consistent description* (WCD) of the incoming object, and note that it must be somewhere between the announced type of the object and **frame_bottom**. Finding the WCD is an **O**(log m) operation, and can be dismissed from our analysis under the wholly reasonable assumption that m < n. Further, let each frame object in the knowledge base be indexed under its WCD, and let the WCDs be ordered as we ordered the slot names above, so that all frame objects with a given WCD can be found in **O**(1) time. Now, by the characteristics of the lattice, the types of the WCDs stronger than the incoming object, and thus the sets of all frame objects with those WCDs, can be found in **O**(log m) time. And unlike the sets of untyped frames with a given slot, these sets are disjoint. No union need be taken, so the comparison time dominates. In the worst case the incoming frame matches all the frames in the knowledge base and n comparisons must be made, but if we assume that the classification hierarchy has been well engineered and that the distribution of frame objects matches that of frame types, then there will only be **O**(log n) objects to check, and the overall time complexity will be **O**(n log n).

The relation level of the FRL is also typed, but similar gains in performance are not expected here. Relations, after all, lack the orientation of frames, which is what makes it possible to use typing to exclude large parts of the frame type hierarchy or knowledge base. Some computation can, of course, be saved by constraining relation operations by type signature, but unless each relation has a unique signature -- and this is unconscionably constricting to the knowledge engineer -- this can only help by allowing the algorithm to discard possible solutions earlier than it otherwise might. The gains in error-checking and project control, however, still occur.

There is reason to ask whether strong typing should be used only as a development tool, or whether it also has a place in a finished product. This is the issue underlying, for instance, whether or not run-time type checking is a worthwhile feature of a language environment. In a system that will always be, in some sense, interpreted, the issue of run-time versus compile-time type checking is less important than the issue of life-cycle phase: a system based on inference will always be covering new ground at run-time, so any diagnostics appropriate during compilation are still appropriate in the run. It is less clear that type checking should be enforced at all in a product that is "out the door." In a general expert systems development environment, one might reasonably expect to be able to turn type-checking on and off. CODER, however, is an experimental system. While it is true that there will be times, mostly during benchmarking experiments and system evaluations, where the system will be run without change, most of its lifetime it is expected to be under either development or modification. Thus where strong typing has been adopted in the system, it has been adopted indefeasibly.

5.4. Levels of Problem Solving

In the last section but one (§ 5.2), we discussed two dimensions in which the knowledge required in an expert system such as CODER can be categorized. First, it can be categorized by type. Hypothetical knowledge is different in type from factual knowledge, which is in turn different from descriptive knowledge. This is true at both the knowledge level, where different types of knowledge require different sets of operations and induce different problem-solving behavior, and at the symbol level, where different representational structures are required to facilitate the appropriate functionality for the type. Second, knowledge can be categorized by inference. Obviously, not all types of knowledge structures can be used with all forms of inference, but neither is the correspondence one-to-one. Hypothetical knowledge, for instance, can be used to reason forward from a given situation to its consequences, or backward from the desired consequences to a situation that could produce them. Descriptive knowledge can be used to classify an individual as a member of a kind, or it can be used to elicit the relevant features of an individual whose kind is known in advance. These two dimensions provide important guides in the engineering of expert systems, as they allow the designer to clarify what knowledge is important to what task and when a task is best deconstructed into separate subtasks. Yet there is a third dimension of knowledge engineering that has provided an important organizing principle in the design of CODER.

When a human discusses how a given set of problems is to be solved, he or she will often organize the discussion in terms of *tactics* and *strategy*. Strategy involves selecting and ordering the tasks that are to be accomplished in solving a problem (winning a game, finding a document); tactics involve how those tasks are to be accomplished. Tactical knowledge involves reacting to the specifics of the situation in order to accomplish a specific objective, and is relatively world-driven. Strategy, in turn, is relatively goal-driven: strategic knowledge is knowledge that informs which tasks are chosen to accomplish a given goal and how the success of those tasks is evaluated. Leon Sterling, who originated the phrase "logical levels of problem solving," adds to these layers of problem solving knowledge a ground layer of world knowledge [STER 84]. Sterling suggests that all three levels of knowledge are needed by a problem solving system and suggests that the different levels be reflected in different representations and inference capabilities used by each level† This tripartite division along, so to speak, a dimension of planning has been adopted in CODER and is reflected in the hierarchical organization of the three classes of knowledge-based entities in the environment: the strategists, the experts, and the external knowledge bases.

Adopting this principle in CODER has had two advantages, one conceptual and one computational. Conceptually, it provides another dimension along which to organize the knowledge needed by the system to function. It allows us to say "this is strategic knowledge; it belongs in the strategist" or "this is world knowledge; it belongs in an external knowledge base." And by putting these divisions of problem-solving knowledge in opposition to each other, it helps the system designer clarify the function of the modules at each level. It is, for instance, more useful to say "an expert is a cluster of tasks" than to say "an expert is a specialist in some domain of discourse:" it tells more about both what knowledge belongs properly to an expert and what the expert must do with it. On a different level, it has been helpful in the implementation of the retrieval strategist to contrast the knowledge proper to the module -- the knowledge required for recognizing problem states and choosing tasks to be performed -- with knowledge that properly belonged to

† Specifically, he suggests that each level use a language that is a metalanguage of the level below, in the sense that the metalanguage is capable of representing the axiomatisation of and thus in some way simulating the lower-level language. If followed exactly, this approach would take us into deep waters indeed, as we would like to use languages with the power of first-order logic even at the tactics level. We have, however, followed his plan to the extent that the languages of the experts and strategists are built on top of and have more power than the world representation language.

experts in the community.

Computationally, separating knowledge into levels has enabled us to specialize modules in CODER by their functionality. Specifically, those modules which deal with knowledge at the level of describing the world -- the external knowledge bases and the (passive) blackboard -- can be fully described using only the factual representation language and a limited set of operations (including relation isomorphism and frame matching) defined on statements in that language (see Appendix). Not only do the modules thus inherit the tractability of these operations, but they have been sufficiently well defined to be coded in a procedural fashion. This saves us from having to use weak methods on the vast amount of knowledge that makes up the system's picture of the world. The subset of facts appropriate to a given situation can be isolated by strong methods, and then turned over to the more powerful but less efficient inference methods of the experts.

Many experts in CODER will take full advantage of the power of the Prolog prover. It is expected, for instance, that experts in document parsing or related term identification will be written in unrestricted Prolog. Other experts, however, for instance those in morphological decomposition of words or in certain types of query processing, will not require as much power, and these experts can make use of more computationally efficient methods. In addition, CODER gains computational advantages at the tactical level by limiting the search space of those experts that do require powerful inference mechanisms. This is accomplished by limiting the knowledge available to the mechanisms, which is in turn accomplished by preselecting the world knowledge imported for a task using strong methods and by limiting the local knowledge base to knowledge of how the task is done.

5.5. The Paradigm of Communicating Objects

In any large programming project, some discipline must be taken to break the project into modules small enough to be addressed by individual programmers and understood by others. In CODER, this modularity is achieved in a knowledge-driven fashion along the three dimensions sketched above. Working in this way has allowed tasks in document analysis and retrieval to be isolated and analyzed independently, and a system of some complexity designed (see Fig. 4.1). To the reader who has followed this far, however, this apparent complexity should not mask the essential simplicity of the system. Each

Type of Expert	Type of Knowledge	Type of Inference
Classificational (abductive)	Meta-facts about frame types, slots, and expected values.	Closest-fit within context of frame hierarchy.
Hypothetical	If-then rules.	Forward- or backward-chaining.
Reflective-procedural	How procedures can be accomplished.	Unknown.
Relational (network-based)	Metaknowledge about relation types; distance metrics.	Relation-following and/or cluster-seeking.
Procedural	Hard-coded.	None.

Fig. 5.1 : A beginning taxonomy of experts.

module in CODER is a specialization of one of a limited set of object classes, and most are instances of either external knowledge bases or experts.

This concept of building a system from objects drawn from a limited number of classes is derived from the principles of object-oriented programming. Object-oriented programming, which has had much success both as a tool for system prototyping and (perhaps because of this) as an artifical intelligence tool, is a particularly appropriate paradigm to use in designing an experimental expert system such as CODER. The available implementations of the paradigm, however, are computationally insufficient for the project, as they typically run on a single machine and spend much of their time in overhead. In addition, they are incompatible with the logic programming paradigm required by our approach to implementing the experts. (The one possible exception to this is the LOOPS environment, which runs on a class of machine neither locally available nor obviously adequate to the project's needs). The MU-Prolog language and the UNIX™ environment are computationally adequate, permit a multi-processor implementation, and provide an efficient implementation of the logic-programming approach. It was thus deemed desirable for the project to graft the relevant features of the object-oriented paradigm to the MU-Prolog / UNIX environment.

Three principles are important to the object-oriented paradigm. First, objects are defined as instances of classes. Second, classes are defined as sets of *methods*, or operations that they can perform. Objects inherit these methods from their parent classes. Third, objects use these methods in reaction to *messages* sent them by other objects. In a mature object-oriented programming systems like Smalltalk, the inheritance of methods can go several levels deep, and can include both additions and deletions of methods from parent classes. CODER does not reach this level of sophistication. In CODER, there is only one level of classes, and objects inherit all and only the methods of the class to which they belong. CODER does, however, use both the class/method system of definition for system modules (objects) and the message-passing paradigm of system operation.

The method of defining a system module by its class and a class by its functionality have proven very useful in CODER. Initial designs for the system had much the same overall structure as the final design, in that they included an analysis subsystem, a retrieval subsystem and a spine of knowledge bases. Much of the sophistication of the final design over those earlier versions has come from a recognition of groups of modules as being members of the same group and from narrowing the functionality of those groups to those methods required by all members. Thus the object-oriented paradigm has contributed to the design of the system. It is also contributing to the implementation of the system by providing a vocabulary for communication among project members. It has been useful in project control to say "this is an expert; it responds to these messages by doing these things" or "no, that's an external knowledge base; it can only do those things." The framework of classes and methods promotes understanding among the developers by providing an organizing principle for the system modules.

Restricting the modules to communication by message passing has also been fruitful in the CODER design, particularly in view of the concurrency implicit in the system. Concurrent operation seems to be a natural way to understand blackboard-based systems, yet it raises issues of atomicity and consistency [ENSO 84]. In CODER, each message to an object is regarded as a primitive transaction that always terminates with a reply. Thus an expert, for instance, cannot lock the blackboard for several operations, but must rely on a sequence of messages to accomplish its ends. And by modifying the message passing paradigm to always include a reply, we have been able to structure the communication primitive of the system in a way that mimics the proving of a Prolog subgoal (see §4.4). This means that the flow of execution of a Prolog module is (conceptually) uninterrupted by inter-module communication, allowing us to treat calls to external modules in the same way as references to the internal knowledge base and to maintain a natural sharing of knowledge between modules without the difficulties of inferential interaction.

5.6. Further Research

The CODER environment is built from a set of communicating modules drawn from a restricted set of functionally defined classes of computational objects and set in a three-dimensional hierarchical structure defined by the type of knowledge each module uses, the way in which it uses it, and the logical level at which the knowledge occurs in a description of the overall system task. The factual representation language, the *lingua franca* of CODER, includes facilities for representing two different types of knowledge (descriptive and factual) that together structure the ground level of representation for knowledge about the world. This is a reasonable start, since it assures us that the modules of the CODER system will be able to communicate among each other and to store and retrieve the knowledge which they are able to extract about documents, terms and users in the problem universe. There are several directions, however, in which the treatment of knowledge in the CODER environment could be strengthened.

First, the representation language itself can be both strengthened and extended. The typed frames of the language are both flexible and computationally tractable. Relations sacrifice tractability for a greater representational power, yet much of the power is not used most of the time. Specifically, there is a middle ground involving what we have here called *facts:* finite relations involving only ground terms. Can these be specified in such a way as to be more tractable then general relational structures? How tractable? Two approaches seem possible here. One is to limit what counts as a fact syntactically. One might, for instance, limit facts from relations to functions from domains of either names or partial descriptions of individuals. An alternative approach, which if successful could be applied simultaneously with any worthwhile syntactic refinements, would be to limit the functionality of facts: to specify a set of tractable operations and let those be all that could be done directly with them. This might require specifying different classes of facts and defining a different functionality for each class, possibly in consort with different inference processes appropriate to the classes. To do this, however, one would somehow need to demonstrate that the (sets of) operations were sufficient for, at least, creating and maintaining all necessary knowledge about the world.

An important extension to the knowledge representation language would be a good formalism for production rules constructed on top of the existing formalism. This was not seen as an important precursor for the CODER project because of the ease with which MYCIN-like production rules can be set up in a Prolog environment [LITT 84]. Our design decision for CODER was to allow each expert to be crafted separately, concentrating on maintaining consistency only at the level at which the experts interact. A unified representation for hypothetical knowledge, however, would allow a wider programming community to work on system production, particularly if generic inference engines were provided that could be parameterized to the individual expert tasks. A more challenging extension would be to add some sort of reflective procedural representation that could be used in reasoning about which sort of task steps were appropriate in a given situation. Both these extensions should to as great an extent as possible be built on top of the factual representation language. If needed, the syntax and semantics of the FRL should be modified accordingly so that statements in the FRL could continue to serve as the primitives in the task-level languages.

Concurrently with this work, the concept of generic inference engines should be explored more fully. A great deal has been written in the AI literature about inference, what can be done and what would be nice to have done, but most is in the form of conjecture. There has been a smaller amount of research (e.g., [CLAN 83]) on what sort of inference is actually *used* in functional expert systems. In this work, we have suggested three types of inference engine -- classificational, hypothetical, and procedural -- that we have found to be essential in the CODER project; Randall Davis and B. Chandrasekaran, working in different application areas, have identified different needs. Further work should concentrate, first on collecting a reasonable set of such engines, second on providing (preferably functionally) a clear and concrete semantics for their operation, and third on implementing them within the community-of-experts model. This is a large job but one which, in conjunction with the design of a unified knowledge representation language, would liberate CODER experimenters and expert systems builders in general to focus on the knowledge engineering tasks of the system.

The need for generic, parameterized modules is easiest to recognize among the experts, but is equally important throughout the system. As detailed in Chapter 4, the blackboard and external knowledge base classes have been designed as generic modules, and some steps have already been taken to construct the truth-maintenance and question-handling segments of the strategist class independently from specialization of a strategist to any particular domain (see Appendix). This progress should be continued and coordinated with the design of a representation for procedural knowledge so that the problem of constructing the domain-dependent segment of the strategist is reduced to constructing a knowledge base of problem-solving strategies. Since the best form for representing and manipulating planning knowledge is as yet unclear, this might involve providing several alternate tools to the system builder. Still, even providing a simple state-action planner as an inference engine for the domain segment would be preferable to leaving the low-level construction to system users.

To return to the subject of knowledge representation, the factual representation language is currently built on a relatively simple lattice-theoretic account of types. Further research should investigate the possibility of relating the work done on the semantics of the FRL to a more sophisticated theory of types, such as Milner's theory of polymorphism [MILN 78] or Martin-Löf's intuitionistic approach [TURN 84]. This would allow us, first, to treat FRL types on a par with the knowledge objects in the environent. Thus we could express facts about classes of objects directly in the FRL, and store them with other world knowledge in the external knowledge bases, instead of treating them as special metaknowledge kept under privileged access by experts in that class. Possible further benefits might include a more dynamic type structure, where the type hierarchy could be modified by the system on the fly instead of being under the sole control of the system administrator, and a better treatment of defaults and defeasability. Second, a more sophisticated type semantics would make it possible to extend an automatic typing scheme *à la* ML and HOPE throughout the entire environment. Mishra and Reddy have demonstrated that such a system of type checking, which infers the (Milnerian) type of each argument in a function based on use, can be adapted to recursive data structures of the sort used in the FRL [MISH 85, cf. also MISH 84]. Mycroft and O'Keefe have developed an experimental type system for Prolog programs [MYCR 84], which differs from Mishra's work in that it requires explicit type declarations but permits polymorphic typing. Implicit typing of this sort is extremely attractive in symbolic programming, as it provides much of the error-recognition capacity of classical strong typing without requiring the programmer to constrain variables intended to be polymorphic. As a bonus, Mycroft and O'Keefe observe that inference engines which make use of type information can be more powerful than those without access to type information: this is consistent with the results of Christoph Walther [WALT 84a], who showed that problems can be solved using many-sorted resolution that are beyond the reach of conventional resolution theorem-provers.

Type-checking of knowledge structures and inference code is only one of a set of tools that would need to be provided in order to do production-scale engineering of expert systems. These include the standard range of development tools (syntax checkers, debuggers, test generators) as well as tools specialized for knowledge acquisition and knowledge engineering. There are many questions open in the design, or even the selection of such tools, before a full range will become available for the knowledge engineer. In this thesis we have treated only the basics of an expert system production environment: the languages and computational structures that allow a group of developers to work together to produce a large-scale knowledge-oriented system. It is our hope that these basics will prove sufficient to the needs of the CODER project, and it is our hope that through observation of the CODER project we will become more aware of what the needs of expert system development truly are. This sort of interplay is essential, however, for only by marrying the techniques of programming in the large to the high-level tools of artificial intelligence research will we be able to produce truly expert systems.

6. Summary and Conclusions

The CODER project is a multi-year, multi-person project to apply the techniques of inference- and knowledge-based programming to the problems of information storage and retrieval. The emphasis in the project is on experimentation. Of research interest are both the techniques that may be helpful to the system tasks and comparisons between different combinations of those techniques and between CODER itself and more classical information retrieval implementations. This need for the system to be reconfigured, as well as the multi-person nature of its development and use, motivates a substantial amount of control over the system development environment.

The work described in this thesis has been a design for the CODER project that both ensures such an environment and provides a framework for both the problems that the system has been planned to address and new problems encountered along the way. As such, there are three points to the work: the knowledge representation language described in Chapter 3, which provides the grounding for both storage and manipulation of knowledge in the system, the architecture described in Chapter 4, which provides a skeleton within which the system as it is now specified can be built and run, and a set of principles described in Chapter 5 that have guided the current architecture and can be used to extend and adapt it throughout the system lifetime. We begin this chapter with a discussion of one of the most imporant of these principles, then move from that to describe the factual representation language, and finally the system architecture.

CODER has been designed to bring three levels of knowledge to bear on the problems of information storage and retrieval. At the topmost level, *strategic* knowledge is used to tailor the process of document analysis to different types of documents, and the process of document retrieval to different users. It is one of the project hypotheses that flexible retrieval strategies will provide better recall and precsion measaurements than can be achieved by a system with a fixed, algorithmic strategy. In addition, it is hoped that a knowledge-based strategic principle will be better able to adapt the system's behavior to individual users. The pertinence of a given document to a given subject varies from user to user: by designing into the system the capacity to form and react to user models at the strategic level, we hope to provide the individual user with information better tailored to his or her needs.

The middle level of knowledge designed into CODER is the level of *tactical* or task knowledge. This is the 'how-to' knowledge needed to accomplish the tasks selected at the strategic level. A great deal of such knowledge goes into the task of information retrieval: knowledge of how to parse sentences, decline words, or recognize bibiographic entries on the analysis end and of how to elicit a user's needs, recognize concepts and perform searches on the retrieval end. In order to facilitate the encoding and use of this knowledge, system design has partitioned it into small clusters determined by which tasks (or sets of tasks) make use of it. Programming these tasks on the level of knowledge and inference should allow us to do some tasks better than have been done with conventional information storage and retrieval systems, and other tasks that have not even been attempted by such systems. (Those tasks that can be done well conventionally can still be accomodated in the system in procedural implementations, due to the separation and clustering of tasks). It is a project goal to experimentally determine which of these tasks have the greatest impact on the IS&R process.

The ground level of knowledge in CODER is knowledge of the problem universe itself. On this level are the descriptions of documents, natural language terms and users that are the raw material with which the system works. Creating knowledge-based descriptions of these entities, and representing these descriptions in a powerful *factual representation language,* allows the system to achieve subtle descriptions involving such refinements as document sections and aspects, word senses and subsenses, and classes and categories of users. It is another project hypothesis that this subtlety of representation alone will be beneficial to the process of information storage and retrieval. Knowledge is more than representation, however, and the functionality inherent in the factual representation language promotes the creation and retrieval of representations as parts of consistent, structured interpretations. In addition, this langauge serves as a *lingua franca* for the more complex knowledge-based modules in the system, in which they can share knowledge, hypotheses and questions about the world.

These three levels of knowledge are mirrored architecturally in three different levels of knowledge-handling constructs. Each class of object in the system has a limited functionality, determined (in part) by the knowledge level on which it operates. Classes at the world knowledge level require only relatively low-powered operations of knowledge entry, comparison and recall. The functional definition of these classes -- the *external knowledge bases* and *blackboards* -- is such that they may be implemented in a procedural language if system efficiency requires it. Modules at the task knowledge level, by contrast, are generally expected to be written declaratively, and to use inference techniques to solve their tasks. The functional definition of this class of modules -- the *experts* or domain specialists -- makes no presuppositions about their inferential power, but does require that each terminate with certain behavior at the completion of each task. In addition, development guidelines ensure that the tasks remain short and single-purpose. Modules at the strategic knowledge level, appropriately enough called *strategists*, are required to react to problem states represented in the factual representation language by scheduling new tasks. It is virtually certain that these modules will use inferential methods.

The modules at the upper two levels of this classification may use a variety of types of knowledge representation systems, specialized for a variety of types of inference. The primitives for any of these systems, however, should be drawn from the factual repesentation language, as the FRL is the language in which they communicate with the remainder of the system. Thus the factual representation language comes with a minimum of its own inferential baggage, but uses a broadly flexible syntax capable of modeling many other representation langauges. 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 semantics of the language are defined within a strong type-object distinction and a lattice-theoretic approach to types. Statements in the language (and the component parts of those statements) are knowledge *objects*, each of which is seen as the instantiation of a particular *type*. Frame types and elementary data types form tangled subsumption hierarchies that are navigated by classification experts to determine, for instance, which document class best describes an individual document in the process of being indexed or what definition of a word fits most closely with the evolving parse of a sentence. Relational structures, such as those occurring among terms in the lexicon, may also be navigated, based on type information about the algebraic properties of the relations. Frame and relation objects further submit to operations of matching and isomorphism, respectively, which use the type information in the objects to determine their semantic consistency. Due in part to the typing mechanism, operations on statements in the language can be implemented with relative efficiency for their structural classes.

Architecturally, the CODER system can be thought of 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. Actually, any number of analysis and retrieval sessions may be running at a given time (see Fig. 4.8).

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 are comprised of the *lexicon*, which includes the system's knowledge about individual words, the *user model base*, which includes knowledge about users and classes of users, and the *document database*, which handles knowledge about individual documents. Type managers for frames, relations, and elementary data types support the uniform use of these representations throughout the system. Associated with the spine is that portion of the expert community that specializes in manipulating the knowledge available from these external knowledge bases and type managers. These satellite experts, like the modules of the spine proper, can be thought of as resources available to any session, either analysis or retrieval, ongoing at any time.

Any ongoing session is moderated by an active blackboard. A *blackboard* [NIIH 86] is a repository for communication between experts, usually divided into a number of subject posting areas. In addition to these subject areas, each CODER blackboard includes a *question-and-answer area* and a single *pending hypothesis area*, which are accessible to all experts in the community. The question-and-answer area provides a structure through which an expert can request information from the other experts in the community before continuing processing; the pending hypothesis area contains a consistent set of high-confidence hypotheses, accessible not only to the experts in the community but to the outside world. CODER blackboards are considered to be active since 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 expert in the community using a knowledge-based model of the expert's area of competence. It is also responsible for selecting the contents of the pending hypothesis area from the hypotheses proposed by the community of experts, combining evidence supported by suitable levels of confidence through a related set of rules.

The analysis subsystem includes a specialized user interface that allows for easy document entry and on-the-fly correction. With sufficient permission, the user of the analysis subsystem may also add knowledge to the portion of the lexicon dedicated to specialized knowledge in the problem domain, for instance in defining technical terms new to the system. Coupled to this interface is a blackboard that coordinates the experts involved in natural language processing and document cataloging. During an analysis session, the system accepts input documents of various types, arranges for as-is storage, and constructs a set of interpretations describing document structure and content. Each interpretation is itself 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 the 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 or extended Boolean logic queries and natural language descriptions of information needs. User behavior is monitored by specialized *translation experts,* 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 modeled on the Prolog subgoal satisfaction paradigm. 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. Vagaries of individual machines, such as the means for implementing this communication policy and the exact interfaces presented to users, are shielded within *resource managers* that map invariant abstract operations into code that can be reimplemented differently as the system is transported to different environments. These resource managers can be written in a lower-level language, such as C++, provided only that it has the requisite networking and interprocess communication primitives. 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.

Bibliography

[ADAM 83] Adam, J.P., J. Fargues and J.C. Pages. "BSM: Une Architecture Mixte Frames/Règles pour Système Expert." *Productivité et Informatique: Pour un Enterprise Dynamique: Recueil des Conférences du Printemps Convention 1983 (Paris):* pp. 278-282.

[AHLS 83] Ahlswede, Thomas E. "A Linguistic String Grammar of Adjective Definitions from *Webster's Seventh Collegiate Dictionary*." In Williams, S. (Ed.). *Humans and Machines*. 1983, pp. 101-127.

[AHLS 84] Ahlswede, Thomas E. "A Lexicon for a Medical Expert System." Presented at the Workshop on Relational Models (Coling84): a revised version is forthcoming.

[AHLS 85] Ahlswede, Thomas E. "A Tool Kit for Lexicon Building." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 268-276.

[AIKI 80] Aikins, Janice S. "Representation of Control Knowledge in Expert Systems." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980).* AAAI, 1980, pp. 121-123.

[AMSL 80] Amsler, Robert A. The Structure of the Merriam-Webster Pocket Dictionary. PhD Dissertation, University of Texas at Austin, Dec. 1980.

[AMSL 84] Amsler, Robert A. "Machine-Readable Dictionaries." *ARIST* **19** (1984), pp. 161-209.

[ATTA 77] Attar, R. and Aviezri S. Fraenkel. "Local Feedback in Full-Text Retrieval Systems." *Journal of the ACM* **24**:3 (July 1977), pp. 397–417.

[BABA 85] Babatz, Robert and Manfred Bogen. "Semantic Relations in Message Handling Systems: Referable Documents." Presented at *IFIP WG 6.5 Symposium (Sept. 1985).*

[BALZ 80] Balzer, Robert, Lee Erman, Philip London and Chuck Williams. "HEARSAY-III: A Domain-Independent Framework for Expert Systems." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980).* AAAI, 1980, pp. 108-110.

[BARR 82] Barr, Avron, Edward A. Feigenbaum and Paul R. Cohen. *The Handbook of Artificial Intelligence.* Los Altos, CA: William Kaufman, 1981-2.

[BART 83] Bärtschi, M. and H.P. Frei. "Adapting a Data Organization to the Structure of Stored Information." In Salton, Gerald and Hans-Jochen Schneider (Eds.). *Research and Development in Information Retrieval, Proceedings, Berlin, May 18-20, 1982*. Berlin: Springer-Verlag, 1983, pp. 62-79.

[BATE 78] Bates, Madeleine. "The Theory and Practice of Augmented Transition Networks." In Bolc, L. *Natural Language Communication via Computers*. Berlin: Springer-Verlag, 1978.

[BELK 84] Belkin, N.J., R.D. Hennings, and T. Seeger. "Simulation of a Distributed Expert-Based Information Provision Mechanism." *Information Technology: Research, Development and Applications* **3**:3 (1984), pp. 122-141.

[BICH 80] Bichteler, Julie and Edward A. Eaton III. "The Combined Use of Bibliographic Coupling and Cocitation for Document Retrieval." *Journal of the American Society for Information Science* **31**:4 (July 1980), pp. 278-282.

[BISW 85] Biswas, Gautam, Viswanath Subramanian and James C. Bezdek. "A Knowledge Based System Approach to Document Retrieval." *The Second Conference on Artificial Intelligence Applications: the Engineering of Knowledge-Based Systems (Miami Beach, FL, Dec. 11-13, 1985):* IEEE, 1985, pp. 455-460.

[BISW 86] Biswas, Gautam, James C. Bezdek, Marisol Marques and Vishwanath Subramanian. "Knowledge-Assisted Document Retrieval." Unpublished paper, Department of Computer Science, University of South Carolina, Columbia, SC, April 4, 1986.

[BLAI 85] Blair, David C. and M.E. Maron. "An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System." *Commununications of the ACM* **28**:3 (Mar. 1985), pp. 289-299.

[BOBR 75] Bobrow, Daniel G. and A. Collins (Eds). *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1975.

[BOBR 77] Bobrow, Daniel G. and Terry Winograd. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science* **1** (1977), pp. 3-46.

[BOBR 83] Bobrow, Daniel G. and Mark J. Stefik. "The LOOPS Manual." Memo KB-VLSI-81-13. Palo Alto, CA: Xerox Corporation Intelligent Systems Laboratory, 1983.

[BOBR 85] Bobrow, Daniel G. "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms." *IEEE Transactions on Software Engineering* **SE-11**:11 (Nov. 1985), pp. 1401-1408.

[BOBR 80] Bobrow, Robert B. and Bonnie L. Webber. "Knowledge Representation for Syntactic/Semantic Processing." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980)*. AAAI, 1980, pp. 316-323.

[BOOK 80] Bookstein, Abraham. "Fuzzy Requests: An Approach to Weighted Boolean Searches." *Journal of the American Society for Information Science* **31**:4 (July 1980), pp. 240-247.

[BORG 84] Borgman, Christine L. "Psychological Research in Human-Computer Interaction." *Annual Review of Information Science and Technology (ARIST)* **19** (1984), pp. 33-64.

[BRAC 79] Brachman, Ronald J. "On the Epistemological Status of Semantic Networks." In [FIND 79], pp. 3-50.

[BRAC 82] Brachman, Ronald J. and Hector J. Levesque. "Competence in Knowledge Representation." *AAAI-82: Proceedings of the National Conference on Artificial Intelligence (Pittsburgh, PA, Aug 18-20, 1982).* AAAI, 1982, pp. 189-192.

[BRAC 83a] Brachman, Ronald J. "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks." *Computer* **16**:10 (Oct. 1983), pp. 30-36.

[BRAC 83b] Brachman, Ronald J., Richard E. Fikes and Hector J. Levesque. "Krypton: A Functional Approach to Knowledge Representation." *Computer* **16**:10 (Oct. 1983), pp. 67-73.

[BRAC 84] 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.

[BRAC 85a] Brachman, Ronald J. and Hector J. Levesque. *Readings in Knowledge Representation.* Los Altos, CA: Morgan Kaufman, 1985.

[BRAC 85b] Brachman, Ronald J. and James G. Schmolze: "An Overview of the KL-ONE Knowledge Representation System." *Cognitive Science* **9** (1985), pp. 171-216.

[BUSH 45] Bush, Vannevar. "As We May Think." *Atlantic Monthly* **176** (July 1945), pp. 101-108.

[CART 85] Cartwright, Robert. "Types as Intervals." *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (New Orleans, LA, 14-16 Jan. 1985).* ACM, 1985, pp. 22-36.

[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.

[CHAR 82] Charniak, Eugene. "Context Recognition in Language Comprehension." In [LEHN 82], pp. 435-454.

[CHAR 83] Charniak, Eugene. "Passing Markers: A Theory of Contextual Influence in Language Comprehension. " *Cognitive Science* 7 (1983), pp. 171-190.

[CHEN 84] Cheng, Yizong and K.S. Fu. "Conceptual Clustering in Knowledge Organization." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984).* AAAI, 1984, pp. 274-279.

[CHOD 85] Chodorow, Martin S., Roy J. Byrd and George E. Heidorn. "Extracting Semantic Hierarchies from a Large On–Line Dictionary." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 299-304.

[CLAN 83] Clancey, William J. "The Epistemology of a Rule-Based Expert System - a Framework for Explanation." *Artificial Intelligence* **20**:3 (May 1983), pp. 215-251.

[CLAN 84] Clancey, William J. "Classification Problem Solving." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984).* AAAI, 1984, pp. 49-55.

[CLAN 85] Clancey, William J. "Heuristic Classification." *Artificial Intelligence* **27** (1985), pp. 289-351.

[CORK 82] Corkill, Daniel D., Victor R. Lesser and Eva Hudlicka. "Unifying Data-Directed and Goal-Directed Control: An Example and Experiments." *AAAI-82: Proceedings of the National Conference on Artificial Intelligence (Pittsburgh, PA, Aug 18-20, 1982).* AAAI, 1982, pp. 143-147.

[COYL 85] Coyle, Karen and Linda Gallagher-Brown. "Record Matching: An Expert Algorithm." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting.* Knowledge Industry Publications, 1985, pp. 77-80.

[CREA 85] Creary, Lewis G. and Carl J. Pollard. "A Computational Semantics for Natural Language." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 172-179.

[CROF 86] Croft, W. Bruce and Roger H. Thompson. "I^3R: A New Approach to the Design of Document Retrieval Systems." Unpublished paper, Department of Computer Science, University of Massachusetts, April 22, 1986.

[CULL 84] Cullingford, Richard E. and Michael J. Pazzani. "Word-Meaning Selection in Multiprocess Language Understanding Programs." *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**:4 (July 1984), pp. 493-509.

[DATE 86] Date, C.J. *An Introduction to Database Systems, Fourth Edition.* Reading, MA: Addison-Wesley, 1986.

[DAVI 82] Davis, Randall et al. "Diagnosis Based on Description of Structure and Function." *AAAI-82: Proceedings of the National Conference on Artificial Intelligence (Pittsburgh, PA, Aug 18-20, 1982).* AAAI, 1982, pp. 137-142.

[DECK 85] Decker, Nan. "The Use of Syntactic Clues in Discourse Processing." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 315-323.

[DEER 81] Deering, Michael, Joseph Faletti and Robert Wilensky. "PEARL -- A Package for Efficient Access tro Representations in LISP." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 930-932.

[DEJO 82] DeJong, Gerald. "An Overview of the FRUMP System." In [LEHN 82], pp. 149-176.

[DEVA 85] Devadson, Francis Jawahar. "Computer Generation of Different Types of Subject Index Entries Based on Deep Structure of Subject Indexing Languages: Deep Structure Indexing System." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting.* Knowledge Industry Publications, 1985, pp. 88-96.

[DESA 85] DeSalvo, Daniel A. and Jay Liebowitz. "The Application of an Expert System for Information Retrieval at the National Archives." *Expert Systems in Government Symposium (Maclean, VA: Oct. 24-25, 1985).* IEEE, 1985, pp. 464-473.

[DIET 84] Dietschmann, Hans J. (Ed.). *Representation and Exchange of Knowledge as a Basis of Information Processes.* New York: North-Holland, 1984.

[DOYL 77] Doyle, Jon. Truth Maintenance Systems for Problem Solving. MS Thesis, MIT, 1977.

[DOYL 80] Doyle, Jon. "A Truth Maintenance System." *Artificial Intelligence* **15** (1980), pp. 179-222.

[DOYL 83] Doyle, Jon. "Admissible State Semantics for Representation Systems." *Computer* **16**:10 (Oct. 1983), pp. 119-123.

[DOYL 85] Doyle, Jon. "Expert Systems and the 'Myth' of Symbolic Reasoning." *IEEE Transactions on Software Engineering* **SE-11**:11 (Nov. 1985), pp.1386-1390.

[DYER 83] Dyer, Michael George. *In-Depth Understanding: a computer model of integrated processing for narrative comprehension.* Cambridge, MA: MIT Press, 1983.

[EARL 73] Earl, Lois L. "Use of Word Government in Resolving Syntactic and Semantic Ambiguities." *Information Storage and Retrieval* **9** (1973), pp. 639-664.

[ENSO 85] Ensor, J. Robert, and John D. Gabbe. "Transactional Blackboards." *Proceedings of Ninth International Joint Conference on Artificial Intelligence (Los Angeles, CA, 18-23 Aug. 1985).* AAAI, 1985, pp. 340-344.

[ERMA 80] Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, and D. Raj

Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." *Computing Surveys* **12**:2 (June 1980), pp. 213-253.

[ERMA 81] Erman, Lee D., Philip E. London and Stephen F. Fickas. "The Design and an Example Use of Hearsay-III." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 409-415.

[ERMA 84] Erman, Lee E., A. Carlisle Scott and Philip E. London. "Separating and Integrating Control in a Rule-Based Tool." *Proceedings of the IEEE Workshop on Knowledge-Based Systems (Denver, CO, Dec 1984):* IEEE, 1984, pp. 238-278.

[EVEN 79] Evens, Martha W. and Raoul N. Smith. "A Lexicon for a Computer Question-Answering System." *American Journal of Computational Linguistics*, Microfiche 83, 1979.

[EVEN 82] Evens, Martha. "Structuring the Lexicon and the Thesaurus with Lexical Semantic Relations." Final Project Report, NSF Research Initiation Grant IST-79-18467, 28 Feb. 1982.

[FAHL 79] Fahlman, Scott E. *NETL: A System for Representing and Using Real-World Information.* Cambridge, MA: MIT Press, 1979.

[FAHL 81] 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):* AAAI, 1981, pp. 257-263.

[FALO 85] Faloutsos, C. "Access Methods for Text. " *ACM Computing Surveys* **17**:1 (March 1985), pp. 49-74.

[FIDE 86] Fidel, Raya. "Towards Expert Systems for the Selection of Search Keys." *Journal of the American Society for Information Science* **37**:1 (1986), pp. 37-44.

[FIKE 85] Fikes, Richard and Tom Kehler. "The Role of Frame-Based Representation in Reasoning." *Communications of the ACM,* **28**:9 (Sept. 1985), pp. 904-920.

[FILL 68] Fillmore, Charles. "The Case for Case." In Bach, Emmon and Robert T. Harms (Eds.). *Universals in Linguistic Theory.* Chicago, IL: Holt, Rinehart, 1968, pp. 1-90.

[FIND 79] Findler, Nicholas V. *Associative Networks: Representation and Use of Knowledge by Computers.* New York, Academic Press, 1979.

[FOXE 80] Fox, Edward A. "Lexical Relations: Enhancing Effectiveness of Information Retrieval Systems." *ACM SIGIR Forum,* **15**:3 (Winter 1980), pp. 5-36.

[FOXE 83a] Fox, Edward A. "Characterization of Two New Experimental

Collections in Computer and Information Science Containing Textual and Bibliographic Concepts." Technical Report TR 83-561. Ithaca, NY: Cornell University Department of Computer Science, Sept. 1983.

[FOXE 83b] Fox, Edward A. Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types. PhD Dissertation, Cornell University, Aug. 1983.

[FOXE 85b] Fox, Edward A. "Composite Document Extended Retrieval: An Overview." *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (Montreal, 5-7 June 1985).* ACM, 1985, pp. 42-53.

[FOXE 85c] Fox, Edward A. "Analysis and Retrieval of Composite Documents." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting.* Knowledge Industry Publications, 1985, pp. 54-58.

[FOXE 86a] Fox, Edward A. and Sharat Sharan. "A Comparison of Two Methods for Soft Boolean Operator Interpretation in Information Retrieval." Technical Report TR-86-1. Blacksburg, VA: Virginia Tech Department of Computer Science, Jan. 1986.

[FOXE 86b] Fox, Edward A. "Information Retrieval: Research into New Capabilities." In Ropiequet, Suzanne and Steve Lambert (Eds.). *The New Papyrus: CD-ROM.* Microsoft Press, 1986 (To appear).

[FOXE 86c] Fox, Edward A. "Improved Retrieval Using a Relational Thesaurus Expansion of Boolean Logic Queries." In Evens, Martha (Ed.) *Relational Models of the Lexicon* Cambridge University Press, 1986 (To appear).

[FROS 85] Froscher, Judith N. and Robert J.K. Jacob. "Designing Expert Systems for Ease of Change." *Expert Systems in Government Symposium (Maclean, VA: Oct. 24-25, 1985).* IEEE, 1985, pp. 246-251.

[FURU 84] Furukawa, Koichi, Akikazu Takeuchi, Susumu Kunifuji, Masaru Ohki and Kazunori Ueda. "Mandala: A Logic Based Knowledge Programming System." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984.* ICOT, 1984, pp. 613-622.

[GARF 78] Garfield, Eugene. *Citation Indexing: Its Theory and Application in Science, Technology, and Humanities.* New York: John Wiley & Sons, 1978.

[GAZD 85a] Gazdar, Gerald. "Applicability of Indexed Grammars to Natural Languages." Report No. CSLI-85-34. Stanford, CA: Center for the Study of Language and Information, Oct. 1985.

[GAZD 85b] Gazdar, Gerald and Geoffrey K. Pullum. "Computationally Relevant Properties of Natural Languages and their Grammars." Report No. CSLI-85-24. Stanford, CA: Center for the Study of Language and Information, May 1985.

[GIRI 85] Girill, T.R. "Narration, Hierarchy, and Autonomy: The Problem of

Online Text Structure." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting*. Knowledge Industry Publications, 1985, pp. 354-357.

[GOME 85] Gomez, Fernando. "A Model of Comprehension of Elementary Scientific Texts." Technical Report CS-TR-85-03, Department of Computer Science, University of Central Florida, 1985.

[GRAY 86] Gray, James N. "An Approach to Decentralized Computer Systems." *IEEE Transaction on Software Engineering* **SE-12**:6 (June 1986), pp. 684-692.

[GREI 80] Greiner, Russell and Douglas B. Lenat. "A Representation Language Language." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980).* AAAI, 1980, pp. 165-169.

[GRIF 82] Griffith, Robert L. "Three Principles of Representation for Semantic Networks." *ACM Transactions on Database Systems* **7**:3 (Sept. 1982), pp. 417-442.

[HAHN 84a] Hahn, Udo. "Textual Expertise in Word Experts: An Approach to Text Parsing Based on Topic/Comment Monitoring." *Proceedings of Coling84 (Stanford, CA: July 2-6, 1984):* ACL, 1984, pp. 402-407.

[HAHN 84b] Hahn, Udo and Ulrich Reimer. "Heuristic Text Parsing in 'Topic': Methodological Issues in a Knowledge-based Text Condensation System." In [DIET 84], pp. 143-163.

[HAHN 85] Hahn, Udo and Ulrich Reimer. "Condensation of Full-Texts in the TOPIC System: Generation of Conceptual Text Representations as a Methodological Requirement for Knowledge-Based Office Information Systems." *ACM Transactions on Office Information Systems* (To appear).

[HALL 80] Hall, Patrick A.V. and Dowling, Geoff R. "Approximate String Matching." *ACM Computing Surveys* **12**:4 (Dec. 1980), pp. 381-402.

[HANK 79] Hanks, P. ed. *Collins Dictionary of the English Language*, London: William Collins Sons & Co., 1979.

[HAYE 77] Hayes, Patrick J. "In Defense of Logic." *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (Cambridge, MA: 1977).* AAAI, 1977, pp. 559-565.

[HAYE 79] Hayes, Patrick J. "The Logic of Frames." In Metzing, D. (Ed.) *Frane Conceptions and Text Understanding*. Berlin: Walter de Gruyter, 1979.

[HAYE 84] Hayes-Roth, Barbara. "BB1: An Architecture for Blackboard Systems that Control, Explain, and Learn About Their Own Behavior." Technical Report No. STAN-CS-84-1034. Stanford, CA: Stanford University Deptartment of Computer Science, Dec. 1984.

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

[HAYE 83] Hayes-Roth, Frederick, D.A. Waterman and D.B. Lenat. *Building Expert Systems*. Reading, MA: Addison-Wesley, 1983.

[HAYE 85b] Hayes-Roth, Frederick. "Rule-Based Systems." *Communications of the ACM,* **28**:9 (Sept. 1985), pp. 921-932.

[HELM 85] Helm, A.R., Marriott, Kimbal, and Catherine Lassez. "Prolog for Expert Systems: An Evaluation." *Expert Systems in Government Symposium (Maclean, VA: Oct. 24-25, 1985).* IEEE, 1985, pp. 284-293.

[HEND 79] Hendrix, Gary G. "Encoding Knowledge in Partitioned Networks. " In [FIND 79], pp. 51-92.

[HILL 85] Hillis, W. Daniel. *The Connection Machine* Cambridge, MA: MIT Press, 1985.

[HOBB 84] Hobbs, Jerry R. "Building a Large Knowledge Base for a Natural Language System." *Proceedings of Coling84 (Stanford, CA: July 2-6, 1984):* ACL, 1984, pp. 283-286.

[HORA 84] Horak, W. and G. Kronert. "An Object-Oriented Office Document Architecture Model for Processing and Interchange of Documents." In *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems (June 25-27, 1984),* pp. 152-160.

[HORA 85] Horak, W. "Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization." *Computer* **18**:10 (Oct. 1985), pp. 50-60.

[HORN 74] Hornby, A.S. ed. *Oxford Advanced Dictionary of Current English.* Oxford: Oxford University Press, 1974.

[HUHN 83] Huhns, Michael N., Larry M. Stephens and Ronald D. Bonnell. "Control and Cooperation in Distributed Expert Systems." *Conference Proceedings: IEEE SouthEastCon '83 (Orlando, FL: 11-14 April 1983).* IEEE, 1983, pp. 241-245.

[HULT 84] Hultin, N.C. and H.M. Logan. "The *New Oxford English Dictionary* Project at Waterloo." *Dictionaries: Journal of the Dictionary Society of North America* **6** (1984), pp. 128;183-198.

[ISRA 81] Israel, David J. and Ronald J. Brachman. "Distinctions and Confusions: A *Catalogue Raisonné.*" *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 452-459.

[KATZ 82] Katzer, J., et. al. "A Study of the Overlap Among Document Representations." Unpublished paper, Syracuse University School of Information Studies, 1982.

[KEHL 83] Kehler, T.P. and G.D. Clemenson. "KEE: The Knowledge Engineering Environment for Industry." IntelliCorp, 1983.

[KESS 63] Kessler, M.M. "Bibliographic Coupling Between Scientific Papers." *American Documentation* **14**:1 (1963), pp. 10-25.

[KIMU 84] Kimura, Gary D. A Structure Editor and Model for Abstract Document Objects. PhD Dissertation (Also Technical Report No. 84-07-02), University of Washington Department of Computer Science, July 1984.

[KIMU 86] Kimura, Gary D. "A Structure Editor for Abstract Document Objects." *IEEE Transactions on Software Engineering* **SE-12**:3 (Mar. 1986), pp. 417-435.

[LEEN 85] Lee, N.S. "P-Shell: A Prolog-Based Knowledge Programming Environment." AT&T Technical Report No. TM 59567-851201-01, December 1985.

[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* Merrimack, NH: DEC, 1984, pp. 3-5 - 3-28.

[LEHN 78a] Lehnert, Wendy G. *The Process of Question Answering: A Computer Simulation of Cognition.* Hillsdale, NJ: Lawrence Erlbaum Assoc., 1978.

[LEHN 78b] Lehnert, Wendy G. "Representing Physical Objects in Memory." Yale Artificial Intelligence Project Technical Report #131 (May, 1978).

[LEHN 81] Lehnert, Wendy G., John B. Black and Brian J. Reiser. "Summarizing Narratives." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 184-189.

[LEHN 82] Lehnert, Wendy G. and Martin H. Ringle. *Strategies for Natural Language Processing.* Hillsdale, NJ: Lawrence Erlbaum Assoc., 1982.

[LENA 84] Lenat, Douglas B. and John Seely Brown. "Why AM and EURISKO Appear to Work." *Artificial Intelligence* **23**:3 (Aug. 1984), pp. 269-294.

[LESS 80] Lesser, V.R., S. Reed and J. Pavlin. "Quantifying and Simulating the Behavior of Knowledge-Based Interpretation Systems." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980).* AAAI, 1980, pp. 111-115.

[LEVE 81] Levesque, Hector J. "The Interaction with Incomplete Knowledge Bases: A Formal Treatment." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 240-245.

[LEVE 84a] Levesque, Hector J. "A Fundamental Tradeoff in Knowledge Representation and Reasoning." *Proceedings of the Fifth CSCSI National Conference (London, ON, May 1984):* pp. 141-152. Reprinted in [BRAC 85a].

[LEVE 84b] Levesque, Hector J. "Foundations of a Functional Approach to Knowledge Representation." *Artificial Intelligence* **23**:2 (July 1984), pp. 155-212.

[LEVE 84c] Levesque, Hector J. "A Logic of Implicit and Explicit Belief." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984)*. AAAI, 1984, pp. 198-202.

[LITT 84] Littleford, Alan. "A MYCIN-Like Expert System in Prolog." *Proceedings of the Second International Logic Programming Conference (Uppsala, Sweden, July 2-6, 1984)*. Uppsala University, 1984, pp. 289-300.

[MCAR 82] McArthur, Dave, Randy Steeb and Stephanie Cammarata. "A Framework for Distributed Problem Solving." *AAAI-82: Proceedings of the National Conference on Artificial Intelligence (Pittsburgh, PA, Aug 18-20, 1982)*. AAAI, 1982, pp. 181-184.

[MCCA 69] McCarthy, John and Patrick J. Hayes. "Some Philosophical Problems from the Standpoint of Artificial Intelligence." In Meltzer, Bernard and Donald Michie (Eds.) *Machine Intelligence #4*. New York: Elsevier, 1969.

[MCCO 84] McCord, Michael C. "Semantic Interpretation for the EPISTLE System." *Proceedings of the Second International Logic Programming Conference (Uppsala, Sweden, July 2-6, 1984)*. Uppsala University, 1984, pp. 65-76.

[MCCU 85] McCune, Brian P., Richard M. Tong, Jeffrey S. Dean and Daniel G. Shapiro. "RUBRIC: A System for Rule-Based Information Retrieval.' *IEEE Transactions on Software Engineering* **SE-11**:9 (Sept. 1985), pp. 939-945.

[MCDA 84] McDaniel, Bonnie. "Issues in Distributed Artificial Intelligence." *International Conference on Data Engineeering (Los Angeles, CA: 24-27 April 1984)* IEEE-CS, 1984, pp. 293-297.

[MALO 86] Malone, Thomas W., Kenneth R. Grant and Franklyn A. Turbank. "The Information Lens: An Intelligent System for Information Sharing in Organizations." *Proceedings of the CHI '86 Conference on Human Factors in Computing (Boston, MA, April 1986)*. ACM, 1986 (to appear).

[MARC 83] Marcus, Richard S. "An experimental comparison of the effectiveness of computers and humans as search intermediaries." *Journal of the American Society for Information Science* **34**:6 (Nov. 1983), pp. 381-404.

[MARC 85] Marcus, Robert. "Generalized Inheritance." *SIGPLAN Notices* **20**:11 (Nov. 1985), pp. 47-48.

[MARS 85] Marsh, Elaine and Carol Friedman. "Transporting the Linguistic String Project System from a Medical to a Navy Domain." *ACM Transactions on Office Information Systems* **3**:2 (April 1985), pp. 121-140.

[METZ 85] Metzler, Douglas P., Terry Noreault, Douglas P. Haas and Cynthia Cosic. "An Expert System Approach to Natural Language Processing." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting*. Knowledge Industry Publications, 1985, pp. 301-307.

[MICH 71] Michelson, D., M. Amreich, G. Grisom and E. Ide. "An Experiment in the Use of Bibliographic Data As a Source of Relevance Feedback in Information Retrieval." In [SALT 71].

[MILL 86] Miller, David P. "A Plan Language for Dealing with the Physical World." *Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Applications (Columbia, SC. March 31-April 1, 1986).*

[MILL 85] Miller, George A. "Dictionaries of the Mind." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 305-314.

[MILL 80] Miller, Lance A. "Project EPISTLE: A System for the Automatic Analysis of Business Correspondence." *Proceedings of the First Annual National Conference on Artificial Intelligence (Stanford Univ., Aug. 18-21, 1980).* AAAI, 1980, pp. 280-282.

[MILN 78] Milner, R. "A Thjeory of Type Polymorphism for Programming." *Journal of Computer and System Sciences* **17**:3 (1978), pp. 348-375.

[MINS 75] Minsky, Marvin. "A Framework for Representing Knowledge." In Winston, P. (Ed.). *The Psychology of Computer Vision.* New York: McGraw-Hill, 1975. Reprinted in [BRAC 85a].

[MINS 86] Minsky, Marvin . "The Society of Mind." *Whole Earth Review.* 51 (Summer 1986), pp. 4-10 (Extract from book of the same title, forthcoming).

[MISH 84] Mishra, Prateek. "Towards a Theory of Types in Prolog." *1984 International Symposium on Logic Programming (Atlantic City, NJ: Feb. 6-9, 1984):* IEEE, 1984, pp. 289-298.

[MISH 85] Mishra, Prateek and Uday S. Reddy. "Declaration-free Type Checking." *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (New Orleans, LA: Jan. 14-16, 1985).* ACM, 1985, pp. 7-21.

[MIZO 84] Mizoguchi, Fumio, Hayato Ohwada and Yoshinori Katayama. "LOOKS: Knowledge Representation System for Designing Expert Systems in a Logic Programming Framework." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984.* ICOT, 1984, pp. 606-612.

[MOOR 82] Moore, Robert C. "The Role of Logic in Knowledge Representation and Commonsense Reasoning." *AAAI-82: Proceedings of the National Conference on Artificial Intelligence (Pittsburgh, PA, Aug 18-20, 1982).* AAAI, 1982, pp. 428-433. Reprinted in [BRAC 85a].

[MORR 83] Morrissey, Joan. "An Intelligent Terminal for Implementing Relevance Feedback on Large Operational Retrieval Systems." In Salton, Gerald and Hans-Jochen Schneider (Eds.). *Research and Development in Information Retrieval, Proceedings, Berlin, May 18-20, 1982.* Berlin: Springer-Verlag, 1983, pp. 38-50.

[MYCR 84] Mycroft, Alan and Richard A. O'Keefe. "A Polymorphic Type System for Prolog." *Artificial Intelligence* **23** (1984), pp. 278-282.

[NAIS 85] Naish, Lee. *MU-Prolog 3.2db Reference Manual.* Melbourne University, 1985.

[NAKA 84] Nakashima, Hideyuki. "Knowledge Representation in Prolog/KR." *1984 International Symposium on Logic Programming (Atlantic City, NJ: Feb. 6-9, 1984):* IEEE, 1984, pp. 126-130.

[NECH 84] Neches, Robert, William R. Swartout and Johanna Moore. "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of their Development." *Workshop on Principles of Knowledge-Based Systems (Denver, CO: Dec. 3-4, 1984):* IEEE, 1984, pp. 173-183.

[NEWE 81] Newell, Alan. "The Knowledge Level." *AI Magazine* **2**:2 (1981), pp. 1-20.

[NIIH 86] Nii, H. Penny. "Blackboard Systems: the Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures." *AI Magazine* **7**:2 (Summer 1986), pp. 38-53.

[NIWA 84] Niwa, Kiyoshi, Koji Sasaki, and Hirokazu Ihara. "An Experimental Comparison of Knowledge Representation Schemes." *AI Magazine* **5** (Summer 1984), pp. 29-36.

[OBER 85] Obermeier, Klaus K. "GROK - A Knowledge-Based Text Processing System." *The Second Conference on Artificial Intelligence Applications (Miami Beach, FL: Dec 11-13, 1985).* IEEE, 1985, pp. 384-389.

[OCON 80] O'Connor, John. "Answer-Passage Retrieval by Text Searching." *Journal of the American Society for Information Science* **31**:4 (July 1980), pp. 227-239.

[ODDY 77] Oddy, R.N. "Information Retrieval Through Man-Machine Dialogue." *Journal of Documentation* **33**:1 (March 1977), pp. 1-14.

[ODDY 81] Oddy, R.N. et al. (Eds.) *Information Retrieval Research* London: Butterworths, 1981.

[OGAW 84] Ogawa, Yutaka, Kenichi Shima, Toshiharu Suguwara and Shigeru Takagi. "Knowledge Representation and INference Environment: KRINE, -- An Approach to Integration of Frame, Prolog and Graphics." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984.* ICOT, 1984, pp. 643-651.

[PAIC 81] Paice, C.D. "The Automatic Generation of Literature Abstracts: an Approach based on the Identification of Self-Indicating Phrases." In [ODDY 81], pp. 172-191.

[PAIC 84] Paice, C.D. "Soft Evaluation of Boolean Search Queries in Information Retrieval." *Information Technology: Research, Development and Applications* **3**:1 (1984), pp. 33-42.

[PATE 84a] 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).

[PATE 84b] 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).

[PEEL 85] Peels, Arno J.H.M., Norbert J.M. Janssen and Wop Nawijn. "Document Architecture and Text Formatting." *ACM Transactions on Office Information Systems* **3**:4 (Oct. 1985), pp. 347-369.

[PERE 83] Pereira, F. "Logic for Natural Language Analysis." Technical Note 275. Menlo Park, CA: SRI International, Jan. 1983.

[PERR 84] Perrault, C. Raymond. "On the Mathematical Properties of Linguistic Theories." *Computational Linguistics* **10**:3-4 (July-Dec. 1984), pp. 165-176.

[PETE 82] Peterson, James L. "*Webster's Seventh New Collegiate Dictionary*: A Computer-Readable File Format." Technical Report TR-196. Austin, TX: University of Texas at Austin, Department of Computer Science. May., 1982.

[PIGM 84] 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.

[POLL 84] Pollard, Carl Jesse. Generalized Phrase Structure Grammars, Head Grammars, and Natural Language. PhD Dissertation, Stanford University, 1984.

[QUIL 66] Quillian, R. *Semantic Memory*. Cambridge, MA: Bolt, Beranek and Newman, 1966. (Also in: Minsky, M. (Ed.). *Semantic Information Processing*. Cambridge, MA: MIT Press, 1968, pp. 27-70).

[RADA 85] Rada, Roy. "Gradualness Facilitates Knowledge Refinement." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-2**:5 (Sept. 1985), pp. 523-530.

[REDD 85] Reddy, R., R. Raman, R. Dziedzic and A. Butcher. "LASER: A High Performance A.I. Programming Environment." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*. ICOT, 1984, pp. 16-23.

[RIEG 84] Rieger, Burghard B. "Semantic Relevance and Aspect Dependency in a Given Subject Domain." *Proceedings of Coling84 (Stanford, CA: July 2-6, 1984)*: ACL, 1984, pp. 298-301.

[RIEG 81] Rieger, Chuck and Steve Small. "Toward a Theory of Distributed Word Expert Natural Language Parsing." *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-11**:1 (Jan. 1981), pp. 43-51.

[RITC 84] Ritchie, G.D. and F.K. Hanna. "AM: A Case Study in AI Methodology." *Artificial Intelligence* **23**:3 (Aug. 1984), pp. 249-268.

[ROBE 76] Robertson, S.E. and K. Sparck Jones. Relevance Weighting of Search Terms. *Journal of the American Society for Information Science* **27**:3 (May-June 1976), pp. 129-146.

[ROCC 71] Rocchio, J.J. Jr. "Relevance Feedback in Information Retrieval." In [SALT 71], pp. 313-323.

[RYCH 84] Rychener, Michael D., René Bañares-Alcántra, and Eswaran Subramanian. "A Rule-Based Blackboard Kernel System: Some Principles in Design." *Proceedings of the IEEE Workshop on Knowledge-Based Systems (Denver, CO, Dec 1984):* IEEE, 1984, pp. 59-64.

[SACC 84] Sacco, G.M. "OTTER - An Information Retrieval System for Office Automation." In *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems (June 25-27, 1984).* ACM, 1984, pp. 104-112.

[SAGE 75] Sager, Naomi. Sublanguage Grammars in Science Information Processing. *Journal of the American Society for Information Science* **26**:1 (Jan.-Feb. 1975), pp.10-16.

[SAGE 81] Sager, Naomi. *Natural Language Information Processing*. New York: Addison-Wesley, 1981.

[SALT 63] Salton, Gerard. "Associative Document Retrieval Techniques using Bibliographic Information." *American Documentation* **10**:4 (Oct. 1963).

[SALT 71] Salton, Gerard (Ed.) *The SMART Retrieval System: Experiments in Automatic Document Processing* Englewood Cliffs, NJ: Prentice Hall, 1971.

[SALT 83a] Salton, Gerard and M.J. McGill. *Introduction to Modern Information Retrieval.* New York: McGraw-Hill, 1983.

[SALT 83b] Salton, Gerard, Buckley, C., and E.A. Fox. "Automatic Query Formulations in Information Retrieval." *Journal of the American Society for Information Science* **34**:4 (July 1983), pp. 262-280.

[SALT 83c] Salton, Gerard, Edward A. Fox, and H. Wu. "Extended Boolean Information Retrieval." *Communications of the ACM* **26**:11 (Nov. 1983), pp. 1022-1036.

[SALT 85] Salton, Gerard, Edward A. Fox and E. Voorhees. "Advanced Feedback Methods in Information Retrieval." *Journal of the American Society for Information Science* **36**:3 (May 1985), pp. 200-210.

[SCHA 73] Schank, Roger C. and Kenneth M. Colby (Eds.). *Computer Models of Thought and Language*. San Francisco, CA: Freeman, 1973.

[SCHA 77] Schank, Roger C. and R.P. Abelson. *Scripts, Plans, Goals and Understanding*. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1977.

[SCHA 81] Schank, Roger C., Janet L. Kolodner, and Gerald DeJong. "Conceptual Information Retrieval." In [ODDY 81].

[SCOT 76] Scott, Dana. "Data Types as Lattices." *SIAM Journal on Computing* **5** (1976), pp. 522-587.

[SEDE 85a] Sedelow, Sally Yeates. "Computational Literary Thematic Analysis: The Possibility of a General Solution." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting*. Knowledge Industry Publications, 1985, pp. 359-362.

[SEDE 85b] Sedelow, Walter A. "Semantics for Humanities Applications: Context and Significance of Semantic 'Stores'." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting*. Knowledge Industry Publications, 1985, pp. 363-366.

[SELF 59] Selfridge, Oliver G. "Pandemonium, a Paradigm for Learning." In Blake, D.V. and A.M. Uttley (Eds.) *Proceedings of the Symposium on Mechanization of Thought Processes*. London: H.M. Stationery Office, 1959.

[SHAP 79] Shapiro, Stuart C. "The SNePS Semantic Network Processing System." In [FIND 79], pp. 179-203.

[SHER 74] Sherman, Donald. "A New Computer Format for *Webster's Seventh Collegiate Dictionary*." *Computers and the Humanities* **8** (1974), pp. 21-26.

[SHIE 84] Shieber, Stuart M., Lauri Karttunen, and F.C.N. Pereira (Eds.). "Notes from the Unification Underground: A Compilation of Papers on Unification-based Grammar Formalisms." Technical Note 327. Menlo Park, CA: SRI International, June 1984.

[SIMM 84] Simmons, Robert F. *Computations from the English*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[SKUC 85] Skuce, D., S. Matwin, B. Tauzovich, S. Szpakowicz and F. Oppacher. "A Rule-Oriented Methodology for Constructing a Knowledge Base from Natural Language Documents." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*. ICOT, 1984, pp. 378-385.

[SLOC 85] Slocum, Jonathan. "A Survey of Machine Translation: Its History, Current Status, and Future Prospects." *Computational Linguistics* **11**:1 (Jan.-March 1985), pp. 1-17.

[SMAL 73] Small, Henry. "Co-Citation in the Scientific Literature: A New Measure of the Relationship Between Two Documents." *Journal of the American Society for Information Science* **24**:4 (July-Aug. 1973), pp. 265-269.

[SMIT 84] Smith, Linda C. and Amy J. Warner. "A Taxonomy of Representations in Information Retrieval System Design." In [DIET 84], pp. 31-49.

[SOND 84] Sondheimer, Norman K., Ralph M. Weischedel, and Robert J. Bobrow. "Semantic Interpretation Using KL-ONE." *Proceedings of Coling84 (Stanford, CA: July 2-6, 1984):* ACL, 1984, pp. 101-107.

[SOWA 84] Sowa, John F. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, 1984.

[SPAR 73] Sparck Jones, K. and Martin Kay. *Linguistics and Information Science.* New York: Academic Press, 1973.

[SPAR 84a] Sparck Jones, K. and J.I. Tait. "Automatic Search Term Variant Generation." *Journal of Documentation* **40**:1 (March 1984), pp. 50-66.

[SPAR 84b] Sparck Jones, K. and J.I. Tait. "Linguistically Motivated Descriptive Term Selection." *Proceedings of Coling84 (Stanford, CA: July 2-6, 1984):* ACL, 1984, pp. 287-290.

[SRIN 84] Srinivasan, Chitoor V. "Knowledge Programming vs Programming: CK-LOG vs Prolog." Technical Report DCS-TR-160. New Brunswick, NJ: Rutgers University Department of Computer Science, Dec. 1984.

[STEF 81] Stefik, Mark. "Planning and Meta-Planning (MOLGEN: Part 2)." *Artificial Intelligence* **16** (1981), pp. 141-170.

[STEF 82] Stefik, Mark et.al. "The Organization of Expert Systems." *Artificial Intelligence* **18** (1982), pp. 153-173.

[STER 84] Sterling, Leon. "Logical Levels of Problem Solving." *Proceedings of the Second International Logic Programming Conference (Uppsala, Sweden, July 2-6, 1984).* Uppsala University, 1984, pp. 231-242.

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

[SUBR 85] Subrahmanyam, P.A. "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate?" *IEEE Transactions on Software Engineering* **SE-11**:11 (Nov. 1985), pp. 1391-1400.

[THOM 85] 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.

[TOKO 84] Tokoro, Mario and Yutaka Ishikawa. "An Object-Oriented Approach to Knowedge Systems." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984.* ICOT, 1984, pp. 623-631.

[TONG 83] Tong, Richard M. et al. "A Rule-Based Approach to Information Retrieval:

Some Results and Comments." *Proc. AAAI-83,* 1983.

[TONG 85] Tong, Richard M., Victor N. Askman, James F. Cunningham, and Carl J. Tollander. "RUBRIC: An Environment for Full Text Information Retrieval." *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (Montreal, 5-7 June 1985).* ACM, 1985, pp. 243-251.

[TOUR 84] Touretzky, David S. "Implicit Ordering of Defaults in Inheritance Systems." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984).* AAAI, 1984, pp. 322-325.

[TURN 84] Turner, Raymond. *Logics for Artificial Intelligence.* New York: John Wiley, 1984.

[VANR 79] Van Rijsbergen, C.J. *Information Retrieval: Second Edition.* London: Butterworths, 1979.

[VOOR 85] Voorhees, E.M. The Effectiveness and Efficiency of Agglomerative Hierarchic Clustering in Document Retrieval. PhD Dissertation (also Technical Report TR 85-705), Cornell University Department of Computer Science, Oct. 1985.

[WALT 84a] Walther, Cristoph. "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984).* AAAI, 1984, pp. 330-334.

[WALT 84b] Waltz, David L. and Jordan B. Pollack. "Phenomenologically Plausible Parsing." *AAAI-84: Proceedings of the National Conference on Artificial Intelligence (Austin, TX: Aug. 6-10, 1984).* AAAI, 1984, pp. 335-339.

[WHIT 83] White, Carolyn. "The Linguistic String Project Dictionary for Automatic Text Analysis." *Proceedings of the Workshop on Machine Readable Dictionaries.* Menlo Park, CA: SRI, May 1983.

[WILE 80] Wilensky, Robert and Yigal Arens. "PHRAN: A Knowledge-Based Approach to Natural Language Analysis." Memorandum No. UCB/ERL M80/34. Berkeley, CA: University of California, Electronics Research Laboratory, Aug. 12, 1980.

[WILE 84] Wilensky, Robert, Yigal Arens, and D. Chin. "Talking to UNIX in English: An Overview of UC." *Commmunications of the ACM* 27:6 (1984), pp. 574-593.

[WILK 73] Wilks, Yorick. "An Artificial Intelligence Approach to Machine Translation." In [SCHA 73], pp. 114-151.

[WINE 85] Winett, S. and E.A. Fox. "Using Information Retrieval Techniques in an Expert System." *The Second Conference on Artificial Intelligence Applications (Miami Beach, FL: Dec 11-13, 1985).* IEEE, 1985, pp. 230-235.

[WINO 83] Winograd, Terry. *Language as a Cognitive Process (Volume 1: Syntax).* Reading, MA: Addison-Wesley, 1983.

[WITT 85] Witt, Bernard I. "Communicating Modules: A Software Design Model for Concurrent Systems." *Computer:* **18**:1 (Jan 1985), pp. 67-77; continued as: "Parallelism, Pipelines, and Partitions: Variations on Communicating Modules." *Computer:* **18**:2 (Feb 1985), pp. 105-112.

[WOHL 86] Wohlwend, Robert C. "Creation of a Prolog Fact Base from the *Collins English Dictionary.*" Masters Thesis. Blacksburg, VA: Virginia Tech Department of Computer Science, March 1986.

[WOOD 75] Woods, William A. "What's in a Link: Foundations for Semantic Networks." In [BOBR 75].

[WOOD 83] Woods, William A. "What's Important About Knowledge Representation?" *Computer* **16**:10 (Oct. 1983), pp. 22-27.

[YEHR 78] Yeh, Raymond T. *Current Trends in Programming Methodology. Vol. IV: Data Structuring.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[YIPM 79] Yip, Man-Kam. "An Expert System for Document Retrieval." Masters Thesis. Cambridge, MA: M.I.T., 1979.

[ZANI 84] Zaniolo, Carlo. "Object-Oriented Programming in Prolog." *1984 International Symposium on Logic Programming (Atlantic City, NJ: Feb. 6-9, 1984):* IEEE, 1984, pp. 265-270.

[ZARR 81] Zarri, Gian Piero. "Building the Inference Component of an Historical Information Retrieval System." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (Vancouver, BC, 24-28 Aug. 1981):* AAAI, 1981, pp. 401-408.

[ZARR 84] Zarri, Gian Piero. "Intelligent Information Retrieval: An Interesting Application Area for the New Generation Computer Systems." *Proceedings of the International Conference on Fifth Generation Computer Systems 1984.* ICOT, 1984, pp. 632-642.

[ZERN 85] Zernik, Uri and Michael G. Dyer. "Towards a Self-Extending Lexicon." *Proceedings of the 23rd Annual Meeting of the ACL (July 8-12, 1985).* ACL, 1985, pp. 284-292.

Appendix:

Functional Specifications

Blackboard / Strategist Complex: Functional Specification

There are five functional modules within the blackboard/strategist composite. In the blackboard proper, the **posting area manager** maintains the integrity of the posting areas while responding to the external *post, retract,* and *view* commands. The other four functional modules are considered parts of the strategist. The **logic task scheduler** is responsible for scheduling new tasks necessary to maintain the validity of the hypotheses on the blackboard, while the **domain task scheduler** schedules new tasks based on domain-specific blackboard events. The **question / answer handler** uses domain-specific knowledge of which experts are able to answer what questions to schedule the tasks involved in the question/answer process. Proposed tasks from these three sources are picked up by the **task dispatcher** and passed to the experts in the local community. The external calls **wake, attend,** and **abort** actually originate from the dispatcher, and the **done** and **checkpoint** communications from the experts are received by it.

0.1. Common Data Structures

Hypothesis -- A 5-tuple

<fact, confidence, expert, id, dependencies>

where:

'fact' is a CODER fact,
'confidence' the confidence the expert has in the fact,
'expert' the ID of the specialist making the hypothesis,
'id' a unique identifier for the hypothesis within the session, and
'dependencies' is a list [or possibly, a p-norm expression] of id's for the hypotheses used by the expert in producing the current hypothesis.

When a fact is posted by an expert, 'id' is passed as an unbound variable. A value is supplied for the variable by the blackboard and returned. All other elements of the tuple should be supplied by the posting expert: in the event that the hypothesis depends on no previous hypotheses, 'dependencies' may be bound to the empty list [p-norm expression].

Question -- An skeletal fact, i.e., a set of CODER variables coupled to a fact, one or more arguments of which are replaced by the variables. Each question is posted to the reserved Question area of the blackboard until it accumulates an adequate set of answers, upon which it is returned to the specialist that originated it. A question has an expert-id, dependency information and an id of its own, but no confidence.

Answer -- A binding list that partially instantiates the question with which it is associated (ie, a subset of the variables in the question together with CODER knowledge structures with which they can be replaced) signed, confidence-rated and justified by the answering expert. (In other words, an answer is exactly like an hypothesis, except that the fact is replaced by a binding list of variables from the question).

Task -- A 5-tuple

<expert, command, scheduler, priority, time>

where:

'expert' is the ID of an expert in the local community,
'command' one of the expert entry points (*wake, attend* &c.),
'scheduler' either 'logic', 'question' or 'domain',
'priority' the priority assigned by the scheduler, and
'time' the date/time when the task was entered.

1. The Posting Area Manager

All interactions with a posting area of the blackboard, either by the external community or by the other modules of the blackboard/strategist complex, occur through the posting area manager. Each area of the blackboard, whether a subject posting area, the pending hypothesis area, or the question/answer area, can be directly accessed only by the posting area manager. The area manager thus must provide the functionality for all **post**, **retract**, and **view** commands. In addition, it must notify the two task schedulers when events occur on the blackboard that may have scheduling repercussions. Specifically, whenever an hypothesis is posted to a subject area, both task schedulers must be notified;

the domain task scheduler with the type of hypothesis posted (the head relation) and the logic task scheduler with the dependencies of the new hypothesis. The domain task scheduler must similarly be informed when a question or an answer is posted, and the logic task scheduler when a hypothesis is retracted or replaced. The exact syntax of these calls is defined below under the headings for the two schedulers.

Still more specifically, the following lists the activity required by each of the calls to the posting area manager:

post_hypothesis (Hyp, Area). -- The hypothesis 'Hyp' is added to the subject posting area 'Area', and its id is bound to a new identifier. The hypothesis is time-stamped, and the logic task scheduler is passed its dependency information. If 'Hyp' involves a fact already posted to the area by the originating expert (if it **replaces** an earlier hypothesis), then the earlier hypothesis is removed from the posting area, and the logic scheduler is notified of this as well. If not (if the hypothesis involves a **new** fact), then only the domain task scheduler is notified.

retract_hypothesis (Hyp_id). -- The hypothesis 'Hyp_id' is removed from the blackboard, and the logic task scheduler notified, using the difference between the confidence value of the retracted hypothesis and 0 (or the nill confidence value) as the change in confidence.

post_question (Quest). -- The question 'Quest' is added to the question posting area, and the question/answer handler notified.

post_answer (Quest_id, Ans). -- The answer 'Ans' is added to the set of answers to the question with 'Quest_id' in the question posting area. The question/answer handler is notified. Note: answers can be replaced under the same conditions as hypotheses (if the hypothesized fact and the answering expert are the same), but no notification is made of such changes, as no hypotheses can be dependent on answers still in the question/answer area.

retract_answer (Quest_id, Ans_id). -- The answer with 'Ans_id' is removed from the set of answers to question 'Quest_id' in the question posting area.

view_area (Area, Hyp_set). -- All hypotheses currently in the subject posting area 'Area' are collected into 'Hyp_set' and returned.

view_questions (Quest_set). -- All hypotheses currently in the question posting area are collected into 'Quest_set' and returned.

view_answers (Quest_id, Ans_set). -- The current set of answers for question 'Quest_id' is returned as 'Ans_set'.

view_pending (Hyp_set). -- The current set of hypotheses in the pending hypothesis area is returned as 'Hyp_set'.

All of the calls provided in the external view of the blackboard are available to the strategist scheduling modules. In addition, the posting area manager provides certain further calls to the strategist modules only. These are:

post_pending (Hyp). -- The hypothesis 'Hyp' is added to the pending hypothesis area, and its id is bound to a new identifier. The hypothesis is time-stamped, and the logic task scheduler is passed its dependency information. If 'Hyp' involves a fact already posted to the area, then the earlier hypothesis is removed from the posting area, and the logic task scheduler is notified of this as well.

retract_pending (Hyp_id). -- The hypothesis 'Hyp_id' is removed from the pending hypothesis area, and the logic task scheduler notified.

retract_question (Quest_id, Ans_set). -- The question with 'Quest_id' is removed from the question/answer area, and the (possibly empty) set of answers accumulated up to the time of call returned as 'Ans_set'.

2. The Logic Task Scheduler

It is the responsibility of the logic task scheduler to maintain the consistency of the deduction trees implicit in the hypotheses on the blackboard under the conditions of possibly changing premises. The knowledge represented on the blackboard is non-monotonic: hypotheses may be retracted or replaced at any time. When this occurs, the hypotheses dependent on the retracted or replaced fact must sometimes be retracted or replaced themselves. The logic task scheduler accomplishes this by scheduling reconsiderations of hypotheses that may be effected: i.e., by scheduling tasks of the form *attempt_hyp (Rel)* for the expert hypothesizing the suspect fact.

In order to perform this scheduling, the logic scheduler draws upon dependency information and maintenance knowledge. The information is received from the posting area manager, and the knowledge is represented in a rule base relating the stimuli of retractions and changes of confidence together with the closeness of the dependency, to responses in terms of task postings at various priorities. For example, one rule might be:

```
IF
        The confidence level of Hyp_A has decreased Amount_1  AND
        Hyp_B is dependent on Hyp_A with level Amount_2,
THEN
        Schedule <Expert_B, attempt_hyp(head(Hyp_B))> with priority of
                Base_level*Amount_1*Amount_2.
```

Other information available for triggering rules includes the age of the hypotheses and the number of dependenceis an individual hypothesis has. Note that reconsiderations will propagate natuarally through the deduction tree above the replaced or retracted hypothesis as the logic-maintenance tasks result in the suspect hypotheses themselves being changed or withdrawn. It is the responsibility of the rule base creator to ensure that such propagation is damped appropriately and does not always result in every hypothesis in the tree being reconsidered.

The logic task scheduler reacts only to the operations of the posting area manager, to which it provides the following calls:

new_dependencies (Hyp_id, Rel, Dependencies). -- Hypothesis 'Hyp_id' has just been posted. It has head-relation 'Rel' and is dependent on the hypotheses in 'Dependencies'. This call should initiate no scheduling activity, but the information must be logged so that dependencies can be traced.

hyp_retracted (Hyp_id, Change_in_conf). -- Hypothesis 'Hyp_id' has just been retracted. The dependency graph must be updated, and reconsiderations may need to be scheduled.

hyp_replaced (Hyp_id, Change_in_conf). -- Hypothesis 'Hyp_id' has just been replaced by an hypothesis which differs in confidence by 'Change_in_conf'. Note that 'Change_in_conf' may be either positive, if the new hypothesis has a higher confidence level than the old, or negative, if it has less confidence. The dependency graph must be updated, and reconsiderations may need to be scheduled.

3. The Question/Answer Handler

Of the two application-specific schedulers in the strategist module, the question/answer handler is the more straightforward. Based on a set of rules associating each type of question (canonically, each head-relation) in the set of all questions that may be posed throughout the community with the set of experts possibly able to answer them, the question/answer handler reacts to postings of questions by posting tasks of the form *attend_quest* to the task posting area. When answers are posted to questions, the question/answer handler evaluates them for adequacy based (at least) on the confidence of the answer and which experts of those capable of answering the question have made an attempt. If the answer set is judged inadequate, new answering tasks are posted; otherwise, the question is removed with its answer set from the question/answer area and sent to the originating expert in the form of an **answers** task.

Answers to questions must be evaluated for adequacy in the context of what other experts are available to attempt answers. Each type of question may have an expert or a set of experts that are best fit to answer it, but other experts may need to be called in if the first attempt to provide an adequate answer fails. An inadequate set of answers might alternatively cause the process of searching for an answer to be restarted, if conditions on the blackboard have changed suffiiently in the meanwhile, or might cause the task that produced the question to be restarted in hopes that a better formulation of the question might be obtained.

The question/answer handler is thus the only module in the system that makes use of the **retract_question** entry to the posting area manager. It uses, in addition, the **view_answers** and possibly the **view_questions** entry. In return, it provides the posting area manager with the triggers:

new_question (Quest_id, Rel). -- Question 'Quest_id' with head-relation 'Rel' has just been posted. Experts capable of answering questions with this head-relation should be scheduled.

new_answer (Quest_id, Ans_id, Conf). -- An answer ('Ans_id') to question 'Quest_id' has just been posted with confidence level 'Conf'. If the answer produces (either alone or with previously received answers) an adequate answer set, the question should be retracted from the question/answer area and consideration of the answer posted as a task for the originating expert. Otherwise, other processing should be undertaken to obtain an answer.

4. The Domain Task Scheduler

The domain task scheduler is responsible for proposing new tasks based on the progress of the current session and the mix of hypotheses currently on the blackboard. It is also responsible for selecting the hypotheses to be posted to the pending hypothesis area, again based on what has happened on the blackboard and what is happening at the moment. The domain-specific strategies for these two types of actions are represented in a set of rules, the antecendents of which are combinations of events within contexts, and the consequents of which are expert tasks to be posted and/or types of hypotheses to be moved to the pending area. For instance, a rule in the retrieval strategist might run:

```
IF
      An hypothesis establishing the document type has been posted
                                                       AND
      It has a confidence larger than Min_level        AND
      (There is no other hypothesis of document type posted    OR
       The new hypothesis is a refinement of the former pending hypothesis)
THEN
      Move hypothesis to pending hypothesis area       AND
      Schedule <doc_type_expert, attempt_hyp(fill_missing_fields)>
            with priority of K.
```

Rules need to be provided in building this local base to deal with reactions to individual hypothesis types, to questions and to answers to questions, all in the context of the phases of the overall task in which the community is engaged. At different phases of the process, different hypothesis postings may require different actions by a different mix of experts.

The domain task scheduler is informed by the posting area manager whenever a new hypothesis is posted. This is the primary stimulus for triggering rules:

new_hyp (Hyp_id, Rel). -- Hypothesis 'Hyp_id' with head-relation 'Rel' has just been posted.

In addition, domain scheduling may be triggered by the task dispatcher, for instance when the task queue is empty, or when no tasks in the queue have priority greater than some particular threshhold.

5. The Task Dispatcher

The task dispatcher coordinates the tasks proposed by the three scheduling units and sends the actual commands to the requested experts. It maintains a priority queue of tasks, which may, however, not necessarily be executed in priority order. As well as by the priority assigned by the scheduling unit to the task, order is determined by the availability of resources (experts, for instance, execute tasks serially, so a task for a given expert may have to wait until the expert is finished), by the location of system modules (tasks for modules resident on different machines may be allocated at a different priority levels, depending on the demands for those machines), and by heuristics balancing how crucial tasks proposed by the three experts are relative to one another.

The task dispatcher is triggered by two sorts of events. First, it is triggered whenever a new task is proposed by one of the scheduling units:

> **new_task (Task).** -- 'Task' should be added to the queue. If desirable according to the above heuristics, it should be dispatched.

Second, the dispatcher is also triggered whenever an executing task is completed:

> **done (Expert).** -- 'Expert' has completed its current task. Based on the task mix in the queue and the scheduling heuristics, another task may now be dispatched.

The task dispatcher maintains a history file of the progress of all the tasks executed during a session. When each expert signals successful receipt of a task, the dispatcher notes the time the task has begun in the history file. It notes the time of completion of each task at the receipt of each *done*. It uses the information in this file to determine which experts are running at any point. The file is also used by the domain task expert, and possibly the question/answer handler.

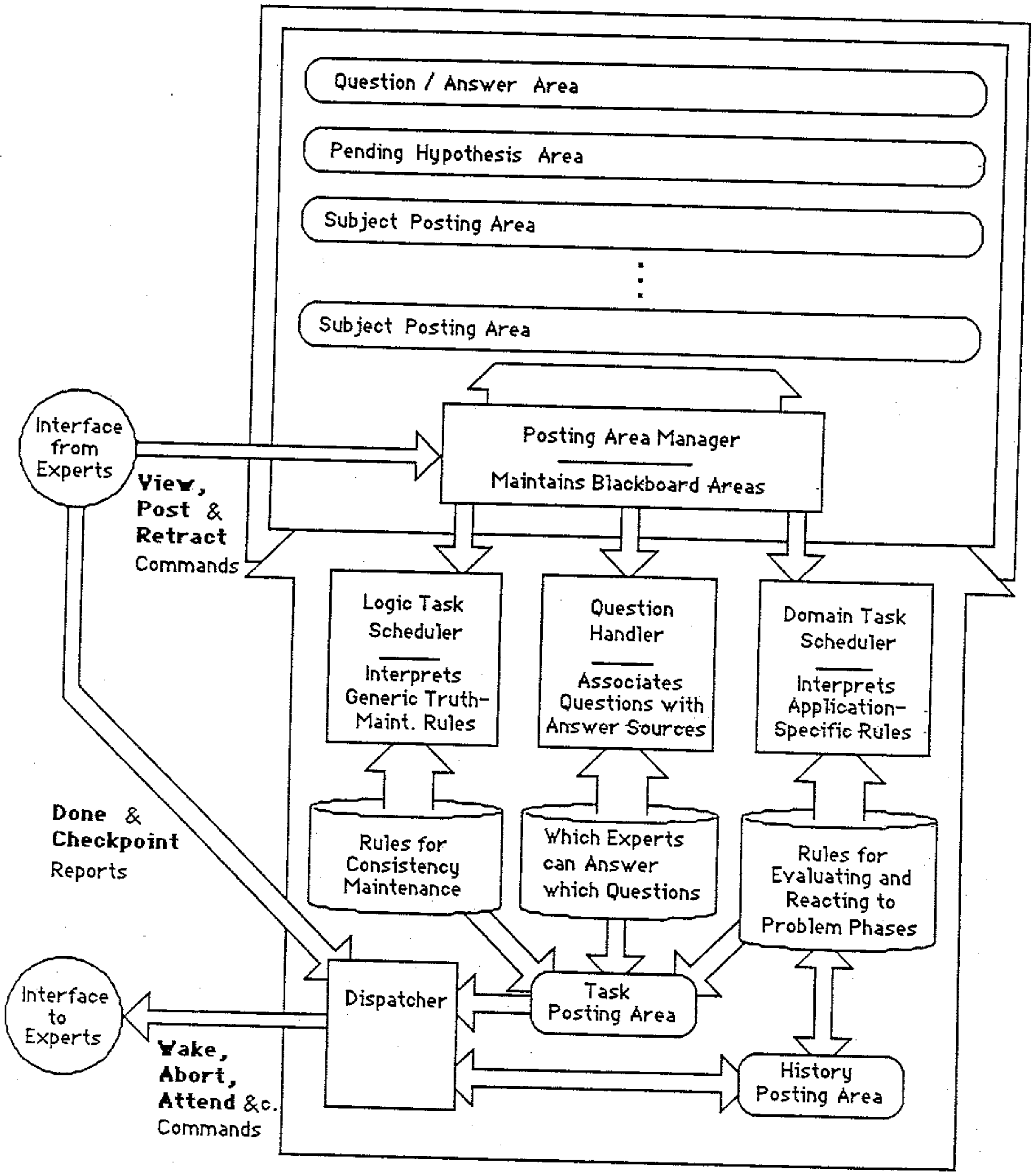

Question / Answer Area
Pending Hypothesis Area
Subject Posting Area
Subject Posting Area
Posting Area Manager
Maintains Blackboard Areas
Interface from Experts
View, Post & Retract Commands
Logic Task Scheduler
Interprets Generic Truth-Maint. Rules
Question Handler
Associates Questions with Answer Sources
Domain Task Scheduler
Interprets Application-Specific Rules
Done & Checkpoint Reports
Rules for Consistency Maintenance
Which Experts can Answer which Questions
Rules for Evaluating and Reacting to Problem Phases
Interface to Experts
Dispatcher
Task Posting Area
Wake, Abort, Attend &c. Commands
History Posting Area

Fig. A.1: Internal structure of blackboard / strategist complex.

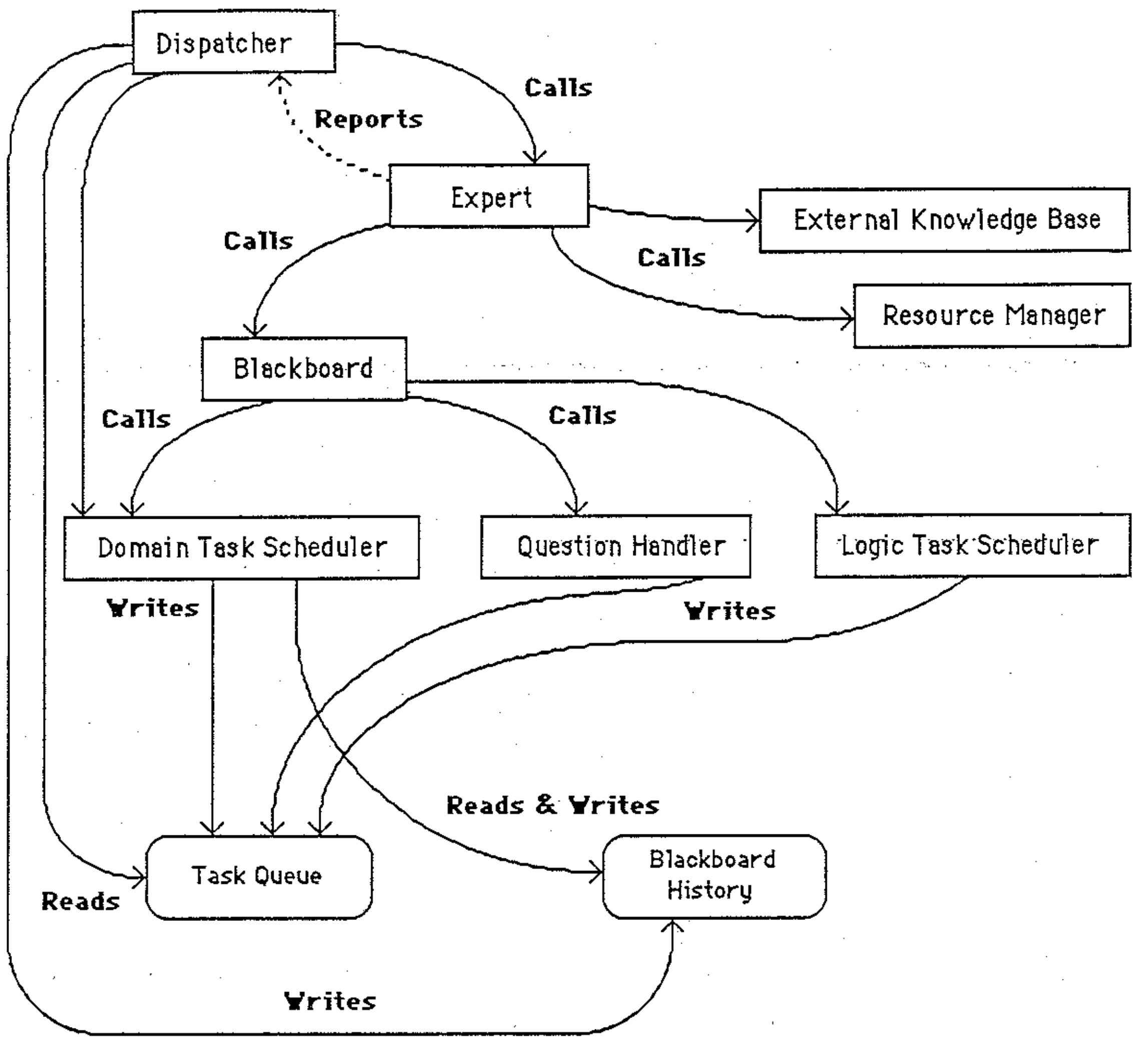


Fig. A.2: Detailed calling hierarchy for a CODER community of experts.

Canonical Expert: Functional Specification

A CODER expert is a specialist in some cohesive domain of discourse capable of executing a set of tasks within that domain. To do this, it draws upon a local knowledge base of tactics, world knowledge and metaknowledge for the domain. It also has access to currently activated blackboards, and may have access as well to an external knowledge source such as a resource manager or external knowledge base. An expert is invoked by the strategist of one or another blackboard community to perform a specified subset of its proper tasks for that community. Successful completion of these tasks will generally involve both reading hypotheses from and posting new hypotheses to the blackboard associated with the invoking strategist. The hypotheses already on the blackboard at the time of invocation represent the current state of the problem task; any new inferences that advance the task are posted to the blackboard as new hypotheses. All state information is thus maintained on the blackboard, releasing the expert to respond to calls by strategists managing other sessions, and providing a centralized record of the problem state for the purposes of planning, history and crash recovery.

Each expert provides an entry point to the task strategists for each task in its repertoire:

attempt_hyp (Relation). -- Expert begins a processing cycle limited to attempting to produce hypotheses with the head relation **Relation.** FAILS just in case **Relation** is not a valid CODER relation or expert has no tasks in its repertoire producing hypotheses based on **Relation.**

These tasks are collected into sets of related tasks by the blackboard subject area to which they relate. All processing relevant to a given blackboard subject area may then be invoked through the call:

attend_to_area (Area). -- Expert is to begin a processing cycle limited to examining the hypotheses in and posting new hypotheses to the subject area **Area.** FAILS if either **Area** is not a valid area on the local blackboard or the expert can perform no tasks relevant to the subject area.

Finally, all the tasks of which the expert is capable can be invoked at once:

wake. -- Expert is to begin an unrestricted processing cycle. All tasks relevant to the current state of the local blackboard are to be attempted in arbitrary order. Failure of any one task will not prevent all others in the expert's repertoire from being attempted. Between each task, the expert is to use the **checkpoint** report to inform the strategist of its progress. Only when every task has been attempted once is it to report **done**.

attend_to_quest (Relation). -- Expert begins a processing cycle limited to attempting to answer questions in the question/answer area with head relation **Relation.**

answers (Quest, Ans_set). -- **Quest** is a question previously posted by the expert and **Ans_set** is a set of answers deemed worth acting upon by the strategist (either a complete set or one with one or more high-confidence answers). Expert is to begin a processing cycle that completes the task begun before **Quest** was posted.

Each of the foregoing entry points begins a time of processing by the expert. The processing may succeed or fail, but the call should be returned to the calling module as an immediate success as soon as the task is recognized as valid. Processing by the expert then proceeds independent from the calling strategist. The expert makes use of two functions to inform the strategist of the progress of its cycle. Both functions are treated as external calls to the strategist, but involve no unbound variables and will never fail barring a system catastrophe.

checkpoint (Expert_id, Ckpt_id). -- Expert has reached a previously identified point in its cycle. For instance, in a full cycle (initiated by a **wake** command) there might be checkpoints for completing all operations in the question/answer area and in each subject area to which the expert had access.

done (Expert_id). -- Expert is done with the task assigned.

Every normal cycle terminates with a **done** report. In case the expert enters a non-teminating process, however, or just in case the processing cycle is no longer needed for external reasons, a call is provided whereby the strategist can terminate the processing cycle before it has completed.

abort. -- Expert is to terminate its current processing cycle. The next processing cycle will begin from the beginning, without prejudice.

External Knowledge Bases ("Fact Bases"): Functional Specification

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 experts in the CODER system are freed to maintain only general knowledge on their particular area of expertise in their local rule bases, relying on the various fact bases of the system to store and recall the "gratuitous complexity" of the problem universe. 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, 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 lanaguge, 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 propostions 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_obj | _]).

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 unusably large retrieval sets.

0.1 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 undating 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_rel**, 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_obj | _]). -- 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.

NOTE: None of the retrieval operations succeed if called with their first arguments unbound or their last argument bound.

---

† Two elementary data objects match if and only if they are equal. A frame A is said to match a frame B if and only if they are of the same type, the filled slots of A constitute a subset of the filled slots of B, and

the value of every filled slot of A matches the value of every filled slot of B. Thus the function **facts_with_frame** can be alternatively specified by saying that it retrieves all facts containing frames that match the incoming frame.

Knowledge Administration Complex: Functional Specification

1. Elementary Data Type Manager

The elementary data type manager handles identification and coordination of EDT's. Operations specific to any particular EDT and conversion operations between that type and other EDT's must be defined at the time of its creation. The EDT manager does nothing to ensure the completeness of those operations at creation time nor their validity during use.

Elementary data types are provided in the CODER environment primarily as a means of modeling attributes. They may also be used to model very simple (or very abstract) objects; other uses are left to the discretion of the CODER module designer. EDT's may be **primitive**, in which case they are defined solely by their native operations, or they may be **constructed** from other EDT's by quantification (ie, by use of the constructors "list_of", "set_of" or "[ordinal]") or by restriction to a subset of the allowed values for the type. Restricted EDT's inherit the operations of their parent type without change; quantified EDT's respond to the operators of their quantifiers by returning objects of their parent type. Thus the two constructors together form a complex but well-behaved semi-lattice above each primitive EDT, and we can say that if EDT A is either a quantification or a restriciton of EDT B then **weaker (A, B)** holds. EDT's may also be constructed by juxtaposition, where one type is followed by another (possibly from a different lattice altogether). In this case, the constructed EDT is defined to be weaker than either of its parent types, and responds to the constructor operations by returning objects of the parent types.

The elementary data type manager provides functions for testing the type of an object:

```
is_elt (elt_type_name, elt_object).
description (elt_type, [[ quantifier, parent_type, restriction] | _ ]).
```

and for navigation of the type lattices:

```
weaker (weaker_type, stronger_type).
supertypes (weaker_type, [stronger_type | _ ]).
subtypes (stronger_type, [weaker_type | _ ]).
```

Functions for creating and manipulating primitive EDTs are part of the definition of each new primitive. Functions are provided by the EDT manager, however, for creating, taking apart and manipulating constructed types: for sets:

```
new_set (set_type, set_object).
is_empty (set_object).
member (element, set_object, remainder_set).
insert (element, set_object, new_set).
equal_sets (set_object1, set_object2).
union (set_object1, set_object2, union_set).
intersection (set_object1, set_object2, intersection_set).
difference (set_object1, set_object2, difference_set).
```

for ordinal collections:

```
new_ordinal (ordinal_type, [elt | _ ], ordinal_object).
is_at_index (ordinal_object, index, element).
set_at_index (ordinal_object, index, element).
equal_ordinals (ordinal_object1, ordinal_object2).
```

and for juxtapositions:

```
new_juxt (juxt_type, [component | _ ], juxt_object).
is_component (juxt_object, parent_type, component).
set_component (juxt_object, parent_type, component).
equal_juxts (juxt_object1, juxt_object2).
```

No operations for lists are provided in the Prolog interface to the EDT manager since lists are native to Prolog. Note that **new_ordinal** provides a list-to-ordinal translation function.

2. Frame Manager

The frame manager supports all functions required for the creation and manipulation of frames. Frames are defined to be typed objects with variable-sized sets of named attributes. These attributes are generally called **slots**, and the values of the attributes **slot fillers**. A frame object may have any number of slots filled, including none, but the slots of a frame object always constitute a subset of the slot list associated with the type of the object. Slots can have default fillers defined during the creation type, or may be defined with no default. In either case, fillers can be changed at any point during the lifetime of the frame object, provided that the new value is of the correct type. A frame type is thus defined to be the combination of its unique name with a set of possible slots for objects of that type, where each slot is itself defined by a name unique within the frame type definition, a type constraint that all fillers of the slot must satisfy, and possibly a default value.

As each new frame type is defined to the frame manager, it is automatically fitted into a tangled **subsumption hierarchy** of types. The hierarchy is so defined that frame type A subsumes frame type B just in case any frame that is an instance of type B can also be regarded as an instance of type A. This classification is based solely on the characteristics that the frame manager controls: the number and types of frame slots. Defaults are not considered; nor are semantic constraints beyond the bailiwick of the manager. The hierarchy is "tangled" in that a frame type may have more than one parent, as well as more than one child.

The frame manager provides operations for testing the type of a frame object and for creating new objects:

new_frame (frame_type, frame_object).
is_frame (frame_type, frame_object).

and for comparing objects:

matching_frame (frame_object1, frame_object2).
equal_frames (frame_object1, frame_object2).

as well as for testing and manipulating slots:

slot_list (frame_type, [[slot_name, slot_type] | _]).
set_slot_value (frame_object, slot_name, value).
remove_slot_value (frame_object, slot_name).
has_slot_value (frame_object, slot_name, value).

Operations are also provided for navigating the subsumption hierarchy:

subframe_list (frame_type, [subframe_type | _]).
superframe_list (frame_type, [superframe_type | _]).
subsumes (ancestor_frame, descendent_frame).

In addition, there is a suite of functions available to the system administrator's interface for defining and deleting new frame_types.

2.1 Detailed description of Operations:

new_frame (frame_type, frame_object). -- With **frame_type** bound to a recognized type designator and **frame_object** initially unbound, succeeds while binding **frame_object** to an object of that type with default values associated with slots that have a default and no value associated with other slots. Where **frame_type** is unbound or is not a recognized type descriptor for a frame data type, or where **frame_object** is already bound, fails.

is_frame (frame_type, frame_object). -- With **frame_type** and **frame_object** both bound, succeeds just in case the object **frame_object** is of type **frame_type**. If **frame_object** is bound and **frame_type** is not, succeeds with **frame_type** bound to the type of **frame_object**. If **frame_type** is bound and **frame_object** is not, behaves exactly like a call to **new_frame**. If neither are bound, succeeds with indeterminate results.

slot_list (frame_type, [[slot_name, slot_type] | _]). -- With **frame_type** bound to a recognized frame type descriptor, and the second argument unbound, succeeds while binding it to the complete list of slots associated with that frame and the types of the objects that may fill each slot. With both **frame_type** and the second argument bound, succeeds just in case the second argument forms a subset of the slots associated with that type. Fails if **frame_type** is not bound.

set_slot_value (frame_object, slot_name, value). -- With all three arguments bound such that **slot_name** is the name of a slot associated with the frame_type of **frame_object** and **value** is of a type subsumed by the slot_type of **slot_name**, succeeds while altering **slot_name** of **frame_object** to **value**. If any of the arguments are not bound, or are bound inconsistently, fails.

remove_slot_value (frame_object, slot_name). -- With both arguments bound such that **slot_name** is the name of a slot associated with the frame_type of **frame_object**, succeeds while altering **slot_name** of **frame_object** to a condition of having no value. If either of the arguments are not bound, or are bound inconsistently, fails.

has_slot_value (frame_object, slot_name, value). -- With the first two arguments bound such that **slot_name** is the name of a slot associated with the frame_type of **frame_object** and the third is unbound, succeeds just in case **slot_name** has been given a value (default or otherwise) and binds **value** to that value. If all three arguments are bound, succeeds just in case **frame_object** has **slot_name** with **value**. If either of the first two arguments are unbound, or if they are bound inconsistently, fails.

subframe_list (frame_type, [subframe_type | _]). -- With **frame_type** bound to a recognized frame type designator, succeeds while binding the second argument to a list of all frame types that are direct subtypes (children) of **frame_type** in the frame hierarchy. With both arguments bound, succeeds just in case the trame_types in the second argument constitute a subset of the children of **frame_type**. Fails if the first argument is unbound.

superframe_list (frame_type, [superframe_type | _]). -- With **frame_type** bound to a recognized frame type designator, succeeds while binding the second argument to a list of all frame types that are direct supertypes (parents) of **frame_type** in the frame hierarchy. With both arguments bound, succeeds just in case the trame_types in the second argument constitute a subset of the parents of **frame_type**. Fails if the first argument is unbound.

subsumes (ancestor_frame, descendent_frame). -- With both arguments bound, succeeds just in case a path can be found linking **ancestor_frame** to **descendent_frame** by parent-child links in the frame hierarchy. If either or both arguments are unbound, succeeds with indeterminate results.

3. Relation Manager

The third data type is composed of logical relations over objects (or other relations). A **relation object** can be thought of as a ground instance of a logical propostion, or as an arc linking nodes which are themselves either objects or molecules formed of objects and arcs.

```
new_relation (relation_type, [argument | _ ], relation_object).
is_relation (relation_type, relation_object).

arity (relation_type, arity).
signature (relation_type, [argument_type | _ ]).

argument_list (relation_object, [argument | _ ]).
argument (relation_object, position, argument).

reflexive (relation_type).
transitive (relation_type, transitivity).
symmetric (relation_type).
antisymmetric (relation_type).
```

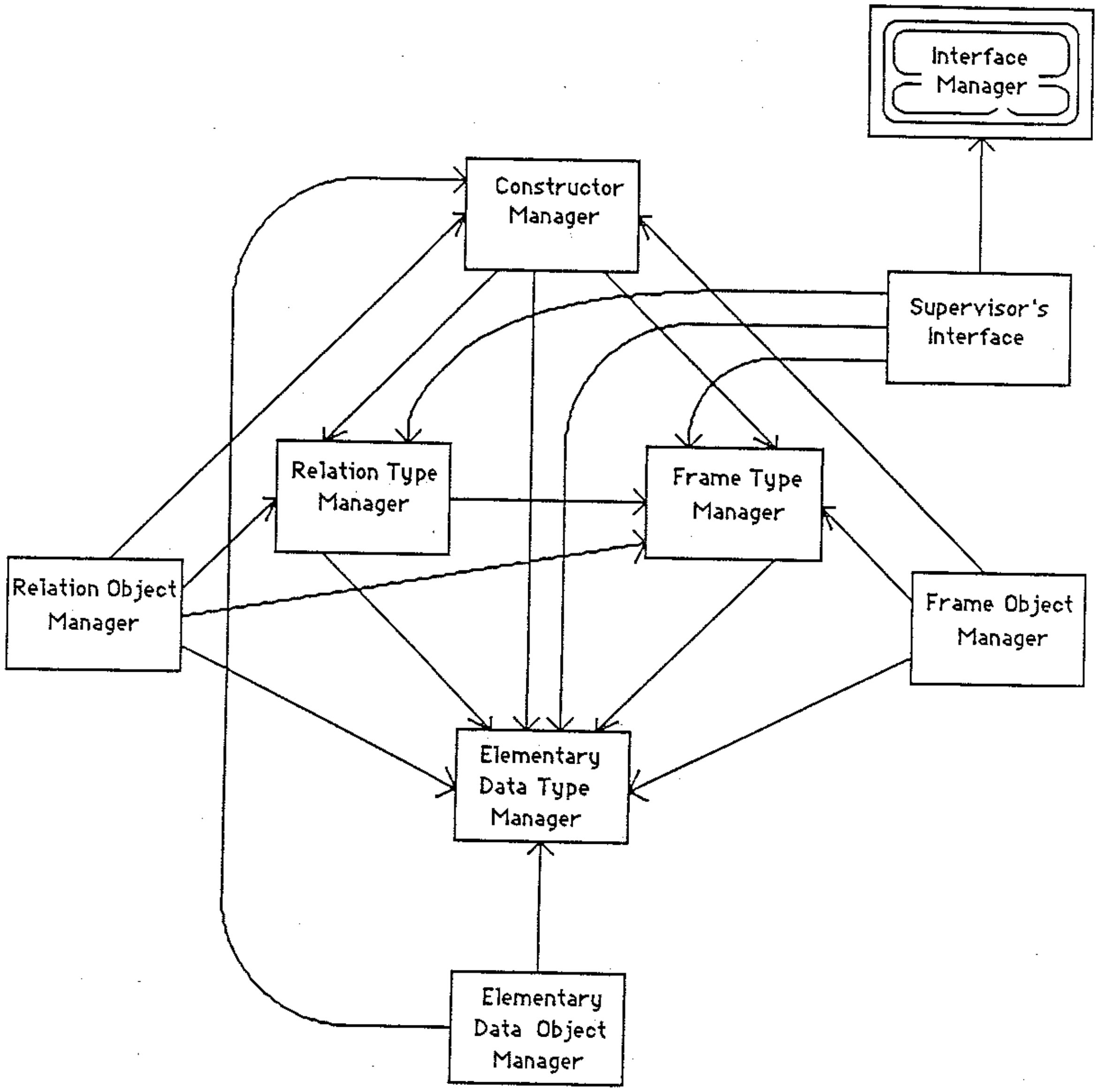


Fig. A.3: Internal structure of the CODER knowledge administration complex (arrows indicate internal calls).

InterModule Communication: Functional Specification

Communication between CODER modules is handled by a set of predicates that assure queuing and sequential execution of tasks. In Prolog modules, the principle aspect of this process is the predicate:

ask (Module, Clause). -- Passes 'Clause' to the module 'Module', where it is dealt with locally. This predicate either fails (if the remote module cannot satisfy 'Clause') or returns, in the manner of a clause proven locally, while optionally binding any or all variables in 'Clause' to further structures. **ask** behaves as a function call: if control backtracks across it, no attempt to resatisfy it is made until control returns "moving forward".

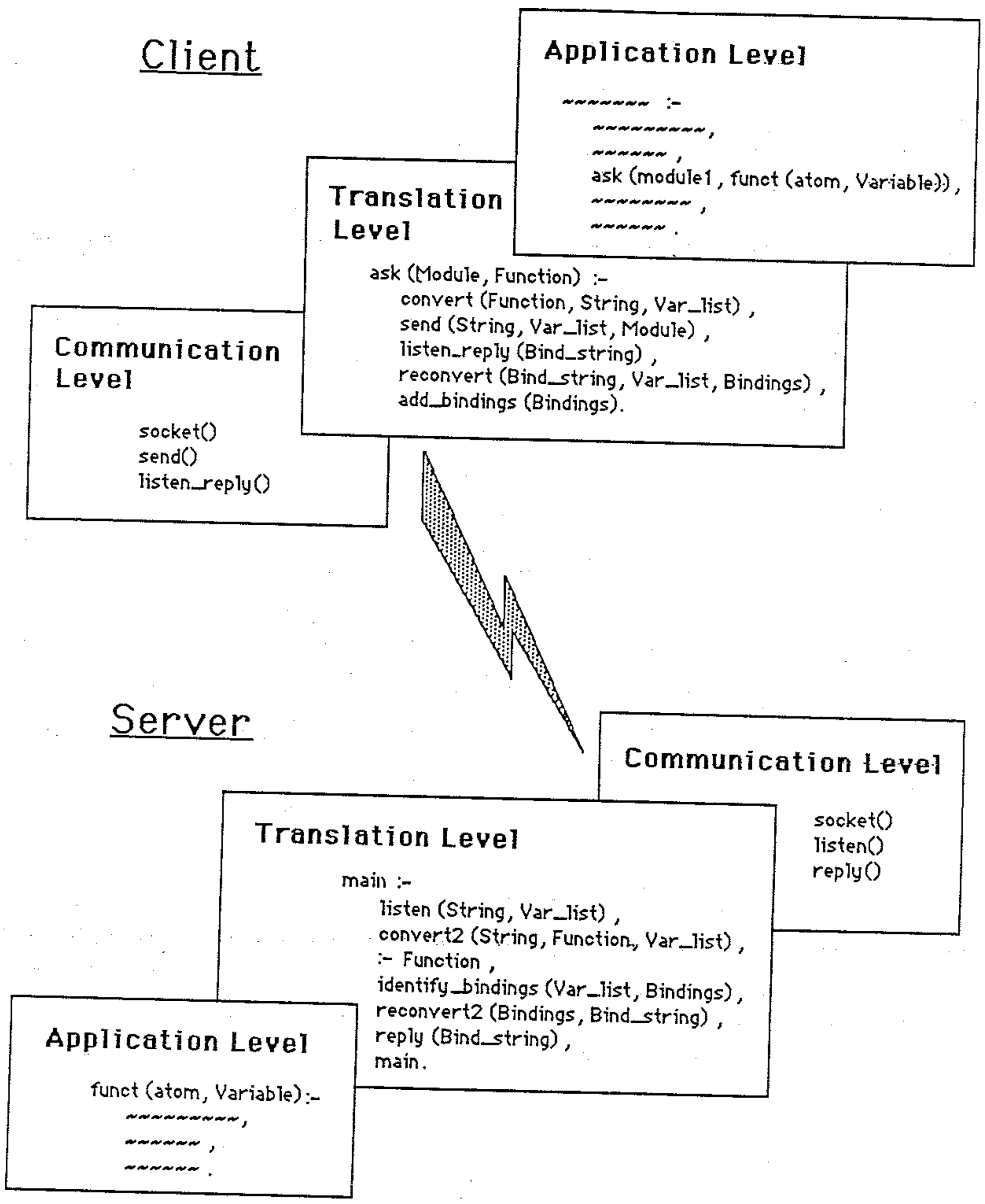
Prolog modules typically receive **ask** calls as calls to the upper-level Prolog prover: the receiving end of the communicaiton process, in this scenario, simply takes 'Clause' and attempts to satisfy it using the rules and facts in the local database. Alternatively, specialized receiving interfaces may be set up using the lower-level structures.

In C modules, the **ask** function becomes a procedure call:

```
ask (module, call, argc, argv)
 Module_id     module;      /* Where 'Module_id' is a defined type    */
 char         *call;        /*  (probably a subtype of atom or int)   */
 int           argc;        /*  restricted to symbolic constants      */
 char         *argv[];      /* denoting modules in the environment.   */
```

-- Presents the predicate 'call' to the remote module 'module' with the argument vector specified by 'argc' and 'argv'. The function returns 0 if the remote module successfully executes the call, else it returns 1. Arguments passed as CODER variables may be replaced with values if they are bound in the remote module.

Remote calls into a C module appear as local procedure calls: the **main()** procedure of a CODER C module is a uniform procedure that does nothing except repeatedly receive remote calls, attempt to satisfy them, and send replies. Note that **main()** will always expect the local procedures to return success/failure information as integer return values. Thus any additional information returned must be through pointer parameters. This fits well with the use of Prolog-style variables in CODER, as unbound variables can be represented by pointers to NULL.



```
~~~~~~~ :-
    ~~~~~~~~~~,
    ~~~~~~ ,
    ask (module1 , funct (atom, Variable)) ,
    ~~~~~~~~~ ,
    ~~~~~~ .
```

```
ask (Module, Function) :-
    convert (Function, String, Var_list) ,
    send (String, Var_list, Module) ,
    listen_reply (Bind_string) ,
    reconvert (Bind_string, Var_list, Bindings) ,
    add_bindings (Bindings).
```

```
socket()
send()
listen_reply()
```

```
socket()
listen()
reply()
```

```
main :-
    listen (String, Var_list) ,
    convert2 (String, Function, Var_list) ,
    :- Function ,
    identify_bindings (Var_list, Bindings) ,
    reconvert2 (Bindings, Bind_string) ,
    reply (Bind_string) ,
    main.
```

```
funct (atom, Variable) :-
    ~~~~~~~~~~,
    ~~~~~~ ,
    ~~~~~~ .
```

Fig. A.4: Implementation levels of intermodule communication.