Expert system technology for integration
by Bennett John Groeneveld

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Industrial and Management Engineering
Montana State University

Abstract:
This study discusses the analysis, design, development and implementation of a modular means to integrate the technology of expert systems into conventional computing applications in an efficient and effective manner.

There are many cases of successfully developed expert systems, though, the potential penetration of expert system technology into the business world has been limited due to the inaccessibility of the technology and the risks that are involved in developing successful expert systems.

The purpose of developing a means for integrating expert system technology into conventional applications is to increase the accessibility of the technology by offering a broader scope of possible applications through the use of embeddable modules.

A macro-to-micro methodology of systems analysis was employed in combination with an incremental and top-down approach to design of computer programs. This was also occasionally mixed with a bottom-up approach to software development and testing.

The study has resulted in a three-layer, framework approach to the design of expert system technology modules for integration into conventional computer programming applications.

The design approach was implemented in the C programming language which resulted in two, large, computer programs. The first application uses a driver to dispatch the various modules as an expert system shell in order to develop stand-alone expert systems or knowledge bases for integrated applications. The second application uses the modules to guide a knowledge acquisition process for the development of mathematical planning models.

The results of both implementations show a high degree of efficiency and effectiveness. The study concludes that the three-tier framework approach to design of expert system technology modules for integration offers the most attractive path to integration when compared to its alternatives.

EXPERT SYSTEM TECHNOLOGY FOR INTEGRATION

by

Bennett John Groeneveld

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Industrial and Management Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 1988

APPROVAL

of a thesis submitted by

Bennett John Groeneveld

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

11/28/88
Date

Donald W. Boyd
Chairperson, Graduate Committee

Approved for the Major Department

11/28/88
Date

David F. Gleason
Head, Major Department

Approved for the College of Graduate Studies

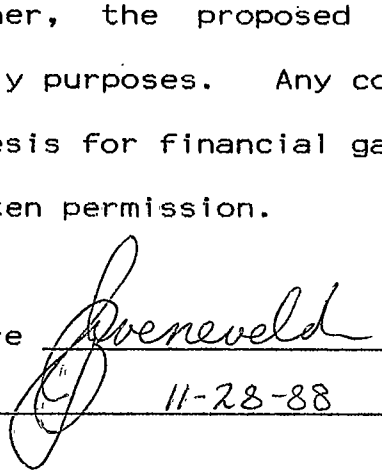
12/8/88
Date

Henry L. Parsons
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date 11-28-88

VITA

Bennett John Groeneveld was born to Gerard and Catharina Groeneveld on September 19, 1961, in Anaheim, California. At age nine his family moved to The Netherlands where he graduated from the Atheneum at the Boschveldcollege High School of Venray in 1980. Dutch is his second language. In 1985 he received his Bachelor of Science in Industrial Engineering from the Rochester Institute of Technology in Rochester, New York. He has worked on several internship assignments for International Business Machines Corporation in Colorado and New York as a resource planner, business planner, manufacturing support engineer, layout engineer and systems engineer. He has also worked for a year as a software engineer in the CIM Interfaces group at ASK Computer Systems Inc. in California. In 1986 he married Wendy June Patterson of Fairport, New York. In that same year he commenced a course of study leading to the degree of Master of Science in Industrial and Management Engineering with a Master's Minor in Computer Science.

ACKNOWLEDGMENTS

Of all the persons who have contributed to my career development at the master's level, I would like to single out the following individuals for their paramount roles. Dr. Richard Reeve initially stimulated my interest in computer applications and integration with the industrial engineering program he put in place as Head of the Department of Industrial Engineering at the Rochester Institute of Technology in Rochester, New York. Mr. Rajiv Kumar, my office mate at ASK Computer Systems Inc. in Mountain View, California, inspired me to consider graduate school seriously. Mr. Nicholas Knight, a former graduate student, has lead me to many valuable computer aids. Dr. Donald Boyd, Director of the Industrial and Management Engineering Graduate Program at Montana State University, helped me advance steadily through this thesis work and my graduate studies through his interest in artificial intelligence and consistent encouragement. And finally, my wife and daughter, who give me a reason to accomplish, are greatly appreciated for their role.

TABLE OF CONTENTS

TABLE OF CONTENTS-Continued

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This study discusses the analysis, design, development and implementation of a modular means to integrate the technology of expert systems into conventional computing applications in an efficient and effective manner.

There are many cases of successfully developed expert systems, though, the potential penetration of expert system technology into the business world has been limited due to the inaccessibility of the technology and the risks that are involved in developing successful expert systems.

The purpose of developing a means for integrating expert system technology into conventional applications is to increase the accessibility of the technology by offering a broader scope of possible applications through the use of embeddable modules.

A macro-to-micro methodology of systems analysis was employed in combination with an incremental and top-down approach to design of computer programs. This was also occasionally mixed with a bottom-up approach to software development and testing.

The study has resulted in a three-layer, framework approach to the design of expert system technology modules for integration into conventional computer programming applications.

The design approach was implemented in the C programming language which resulted in two, large, computer programs. The first application uses a driver to dispatch the various modules as an expert system shell in order to develop stand-alone expert systems or knowledge bases for integrated applications. The second application uses the modules to guide a knowledge acquisition process for the development of mathematical planning models.

The results of both implementations show a high degree of efficiency and effectiveness. The study concludes that the three-tier framework approach to design of expert system technology modules for integration offers the most attractive path to integration when compared to its alternatives.

CHAPTER 1

INTRODUCTION

The discussion of expert systems (ESs) inevitably surfaces the term artificial intelligence (AI). AI is a cross disciplinary area of research which is currently and primarily realized through the use of advanced computer programming techniques. The expert system (ES) is a relatively mature area of AI research, which is successfully being used in today's industrial world. However, ES technology is far from perfected. There are many cases of successful application of the technology, though, it appears that the scope of suitable candidates for application has been limited [1,2,3,4,5,6]. This can be attributed to several factors. Expert systems have acquired the stereotype of a rather esoteric technology reserved for those privileged individuals who are well versed in an AI programming language. They are typically accessible only as stand alone skeletal development tools or shells often requiring specialized computer hardware to achieve acceptable levels of performance. Additionally, the technology, though in some cases successful, is not yet completely refined as its implementation carries with it no guarantee for success. One reason for this is the absence

of any universal top-down approach for the development of an expert system's knowledge base, the activity commonly referred to as the knowledge acquisition process [7,8]. Knowledge representation schemes available in today's ES development tools are limited. Also, ES shells are typically the result of some successful ES which was developed to achieve an expert level of performance in some problem domain. On the other hand, the implementation of a derived skeletal ES applied to some other domain of knowledge is not based on some underlying matching theory for ES shell design to problem domain type or task. Thus, inaccessibility of the ES, coupled with its potential implementation risks, has resulted in slowing the progress of the application of the current ES technology.

Industrial engineers are interested in the design, improvement, and installation of integrated systems. Armed with a breadth of skills, they consistently seek improvement by integrating new technologies into the industrial environment. This thesis effort is aimed at making ES technology more accessible by developing a mechanism for integrating ES technology into conventional computer programming applications.

Integration may not be the only means by which ES technology can become more accessible. Andrew Potter, for example, writes that ESs will not come into common usage until obstacles associated with their user interface have

been surmounted [9]. According to Potter, an interface based on currently available technology can solve these obstacles: direct manipulation using symbolic representation of objects, choice constructs to guide and inform user decisions, and concurrent contexts to let the user integrate the benefits of several interactive systems.

Notable headway in the direction of increased accessibility has already been made by porting some ES shells, designed and developed in the research world, to conventional hardware platforms, such as the popular Personal Computer (PC) from International Business Machines Corporation (IBM). Many tools have also been translated from Lisp programming language dialects into conventional languages such as the Pascal and C programming languages in order to achieve higher levels of performance [10,11]. Traditional hardware architectures are optimized for such conventional programming languages, whereas AI languages such as Lisp require specialized hardware for optimum performance. The specialized hardware found in Lisp machines is primarily needed to make possible concurrent garbage collection without interruption of parallel program execution [12].

A generic means for integrating ES technology into conventional computer applications may further expand the accessibility of this new technology. Integration has been spoken of as the key to ESs' success [13,14,15,16,17,18,

19]. Dissimilar to other technologies, the best approach to integration, if one exists, has not yet been identified. This research effort is aimed at exploring a modular approach for the integration of ES technology into conventional computing applications. Modular is meant to imply that an engineer developing a software application could incorporate prepackaged units of ES technology on his own, without requiring the use or knowledge of an AI language or machine. Accessibility is increased by treating ES technology as any other programming technology, i.e., as consisting of data structures and algorithms. Thus, rather than viewing expert systems as a separate, esoteric science, a bottom-up, generic approach embodies the technology as a set of interacting modules that the engineer can readily incorporate into a conventional application. It is not new for an ES or ES shell to be written in a conventional programming language such as Fortran or C; but, offering the technology as a set of integratable modules at the hands of the application developer will likely allow for an increased access to the technology.

CHAPTER 2

PROBLEM DEFINITION

Statement Of The Problem

The thrust of this thesis effort is to develop a means for integrating the computer programming methodologies of ESs into conventional or traditional computer programming applications in an efficient and effective manner.

The purpose of integrating ES technology into mainstream programming is to increase the accessibility of the technology, thereby expanding the horizon of its possible applications. ES technology can potentially add a new programming capability to an existing or known application. A new programming technique may be desired when the methodologies borrowed from programming techniques developed for ESs are more suitable to solve a particular problem than conventional programming approaches. Suitability may be determined by a number of factors such as ease of implementation, maintenance once implemented, capability to handle expansions or extensions, and space and time complexities.

Integration efficiency refers to the design for integration. During development of an integrated

application, efficiency can be measured by the effort required to perform the integration. During operation of the completed application, efficiency can be measured as the degree of utilization of individual, integrated, ES modules. Because of the difficulty involved in quantitatively evaluating software, this utilization is most easily assessed relative to alternative methods of integration. The synchronization, or match of performance between application and integrated ES technology, is also a measure of integration efficiency in a completed application. Time and space complexities of the implementation of the design for integration will be a determining factor in synchronization. These complexities are affected by the choice of programming language for implementation and by the application developer's program implementation.

Integration effectiveness can be qualitatively assessed by how well the proposed method for integration accomplishes its task. Inter-application communication, or the ability to share control of program execution, between the conventional portion of the application and the ES modules, will directly affect this. Additional measures are transparency of integration, and flexibility for modification and expansion.

Review Of Relevant Work

Rule-Based Expert Systems

At the International Joint Conference on Artificial Intelligence in 1977, Edward Feigenbaum presented an insight into ESs which has since proved to be a key, namely, that the power of an ES derives from the knowledge it possesses, not from the particular formalisms and inference schemes it employs [20]. Since that time, interest in ES research has increased steadily, and in 1983 the publication of Feigenbaum and Pamela McCorduck's book The Fifth Generation caught the attention of an even larger audience with warnings that the Japanese are closing in on ES technology and knowledge engineering in general [21,22].

ESs are computer programs that can solve problems which normally require human expertise. ESs solve these problems at competence levels rivaling that of human experts. The most successful ESs have been those that were developed to solve problems requiring a narrow and deep domain of knowledge. This definition of ESs is performance specific, yet ESs are usually classified in terms of architecture, for example, "rule-based" ESs. The term, rule-based, refers to the knowledge representation used in this class of ESs. Rule-based ESs are the most prevalent type of ES. The paradigm of rule-based ESs offers a means to develop ESs using a combination of relatively well known and

established programming techniques. Many of today's rule-based ESs do not employ heuristic search techniques, rather, they typically rely on exhaustive, problem-solving strategies. Because the control systems provided with ES development tools are largely domain independent, they cannot easily incorporate domain specific knowledge to increase the efficiency of their search strategies.

The origins of today's rule-based ESs lie in the development of a rule-based ES called Heuristic Dendral. Dendral was successful in solving a tough interpretation problem normally reserved for experienced human chemists: that of elucidating the structure of complex molecules from mass spectrograms [23]. The program was written by Feigenbaum, et al., in a Lisp Programming language dialect called Interlisp, with subroutines in the Fortran and Sail programming languages. Dendral uses an algorithm to systematically enumerate all possible molecular structures. The algorithm is augmented with chemical expertise to prune this list to a manageable size [24]. Dendral, started in 1965 at Stanford, is considered to be the first, rule-based, ES and the prototype to Mycin [25]. The Mycin ES, developed in 1976 at Stanford University by Shortliffe, et al., was an important development because its architecture has become the principal basis for the design of most of today's rule-based ESs [26,27]. Mycin gives consultative advice on diagnosis and therapy in certain

classes of infectious blood diseases. Mycin produces high quality results with performance comparable to experts in the field. Three specific design features attract interest in a rule-based approach to knowledge representation. Firstly, knowledge representation consists of rules of thumb or heuristics which are separated from the problem solving technique. Due to their independence, rules can be added or removed from the system without affecting the program's problem solving approach. This allows for separating the encoding of human expertise into the system, commonly referred to as knowledge base development, from the engineering of software which operates on the knowledge to solve a problem. Knowledge base development can be performed by professionals who do not have a computer programming background. Secondly, the rules have certainty factors allowing the system to reach plausible conclusions from uncertain evidence. Thirdly, Mycin has the facility to reproduce those rules used to arrive at a conclusion, thereby explaining its reasoning process. Mycin was written in Interlisp, uses an exhaustive depth-first search technique to select rules, and employes a backward-chaining control scheme to operate on selected rules [28]. Mycin spawned a skeletal system or shell called Emycin (for Essential or Empty Mycin), which provides a generic framework for building ESs by allowing the reasoning mechanism of Mycin to be applied to any set of rules or

knowledge base. Thus, Emycin became the first ES shell, a generic tool to develop ESs. Emycin has become a model for the design of ES shells. Edward Feigenbaum has coined the term "knowledge engineering" to describe the process of building ESs where, typically, an ES shell is employed [29].

Probably the most commercially successful and mature ES to date is R1 or Xcon. John McDermott created R1 using the OPS5 programming language as a joint project between Carnegie-Mellon University, and Digital Equipment Corporation (DEC) (who refer to it as Xcon). As Xcon, its task is to configure all VAX family computer systems for DEC plants in the United States and Europe, and it configures at a very detailed level on a daily basis. R1 is a rule-based ES with more than 4,000 rules. It uses a forward-chaining control scheme for which the OPS5 programming language has been optimized through a powerful interpreter that matches rules against data. Unlike Mycin, R1 employs nearly no search but is guided through a pattern matching facility provided by OPS5 [30,31,32].

Besides their respective successes, Mycin and R1 provide additional insight when compared to each other. They are examples of systems at two ends of the search spectrum. Because of its domain, Mycin must try to find all possible diseases that might account for a patient's symptoms. In the case of R1, however, it suffices to find

one good system configuration [33]. Unfortunately, there is still no theory to map the problem solving capability of a particular ES architecture to categories of tasks that it can solve. The emphasis of ESs research has been biased towards achieving impressive levels of performance on different tasks with minimal or no theoretical underpinning [34].

The Technology Of Expert Systems

The first comprehensive publication shedding some light on the advanced programming methods used in ESs was Nils Nilsson's "Principles Of Artificial Intelligence" in 1980 [35]. Nilsson's book is based on general computational concepts involving data structures, types of operations performed on these data structures, and control strategies. Nearly all subsequent publications discussing search strategies used in intelligent problem solving refer to Nilsson's important publication [36,37,38,39,40,41,42, 43,44].

Nilsson discusses rule-based ESs, which he prefers to call rule-based deduction systems, as a special case of a more general underlying type of system called a production system. A production system has three clearly separable components, namely, a global database, a set of production rules, and a control system. The global database may be as simple as a small, matrix of numbers or as complex as a

large, relational, indexed-file structure. The knowledge about a problem domain that is represented in the global database is sometimes called declarative knowledge. The production rules operate on the global database. Each rule has a precondition that is or is not satisfied by the global database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the global database. The domain knowledge represented by rules is often referred to as procedural knowledge. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the global database is satisfied. Knowledge embodied by the control system is often referred to as control knowledge [45].

The main difference between a production system and a conventional computational system is structural. The global database of a production system can be accessed by all the rules, rules which can only communicate through the global database, whereas, a conventional computational system lumps global database and rules together as data structures and algorithms in a single hierarchical computer program. This feature is the result of the evolutionary research and development of AI systems which typically have required extensive knowledge. The separation of components in a production system allows for increased flexibility, whereas, in the case of the hierarchically-

organized conventional system, the alteration of knowledge may require extensive modification to programs, data structures, and subroutine organization. Besides the capability to incorporate knowledge, increased flexibility is precisely one of the benefits that integration of ES technology can offer conventional computing systems [46].

Rule-based systems, according to Nilsson, have certain advantages over other AI production system alternatives such as resolution systems. English statements, by their nature, often carry extra-logical, or heuristic, control information. Rule-based systems use implicational expressions, containing general assertional knowledge about a problem domain in their original form, whereas resolution systems convert them to disjunctions of literals, requiring manipulation using predicate calculus, with the possibility of losing valuable control information. This is because a single disjunction of literals, called a clause, can be logically equivalent to a number of different implications, each with their own, extra-logical control information. The manipulation of clauses in a resolution system can exacerbate this problem. Nilsson also discusses the advantages of rules in their application to the global database, and their ease of use with And/Or graphs, an important representational aid for the solution of AI problems [47].

Nilsson's publication and the many subsequent books referring to intelligent search strategies describe such search strategies in great detail; however, very little material discussing actual design and implementation of rule-based ESs is available. Those that are available are typically oversimplified and not practical [48,49,50,51]. These publications never discuss the topic of user interfaces, which is of crucial importance to any software system. The more fundamental problem is that they tend to concentrate on the control structure, which is a matter of implementation, rather than on the knowledge base or working memory data structures, which require efficient designs for adequate performance. Charniak and McDermott in [52] note this lack of material specifically with respect to efficient designs for rule searching. The control mechanism in a production system acts as an interpreter, continuously scanning rules to find appropriate rules for application. To do this rapidly, an efficient design for both search algorithm and accompanying data structure being searched is required. When confronted with this problem, Charniak and McDermott show a preference for standard computer science references for the design of efficient search algorithms and data structures, rather than publications with the flavor of "hacking."

Integration Of Expert Systems

The successful fulfillment of this thesis' objective carries an implied comparison with other proposed or hypothetical methods for integration. Therefore, it is necessary to discuss work that is aimed at harvesting the benefits of the technology of ESs through integration with other applications. To date, integration can be seen as being approached, somewhat haphazardly, from primarily four, variously overlapping directions, namely, the AI language environment approach, the hybrid approach, the port-to-conventional-hardware approach, and the embedded or buried approach. It is interesting to note that some ES integrators are pursuing several avenues in parallel. Texas Instruments Incorporated (TI), for example, is working on the integration issue from nearly all approaches by providing systems falling into all categories except embedded.

AI Language Environment Integration. This approach subscribes to the notion that ESs should remain within their original development platforms. As Richard Gabriel, the originator of Common Lisp, puts it, "The path from the laboratory to the field is smoothest when both worlds use the same programming language." [53]. Lisp is the primary programming language used for AI research in the United States. According to Gabriel, the Lisp programming

language suffers from many misperceptions that regard it as being inherently slow due to its interpretive nature, its need for run-time type checking, and its interrupting garbage collection. He points out that Lisp compilers have been available since 1959, that Lisp systems have declaration facilities to direct the compiler to avoid run-time type checking, that explicit memory management techniques are available, and that in bench mark tests Lisp is rated from 20% slower to 30% faster than the C programming language, Lisp's primary conventional alternative. Integration with commercial computing can be accomplished by invoking non-Lisp code, or having foreign code call Lisp code. Gabriel says that translating an AI application to another language cannot be justified if the Lisp version is fast enough, is integrated with the commercial environment, and is small enough to fit on the existing computers. TI encourages this approach as well, by stressing the productivity advantages that Lisp workstations offer the programmer, namely, rapid prototyping, simplified maintenance of coding, and helpful Lisp workstation functions. TI has developed a dedicated, Lisp-language-based workstation for this purpose using a proprietary, Lisp optimized microprocessor, called the Texas Instruments Explorer Symbolic Processing System [54]. Teknowledge, Inc., in working for the Department of Defense, is developing an AI language based environment

called Abe, which is also aimed at integration [55]. Abe provides a global architecture within which engineers can design programming environments for general applications and then customize specific applications. The environment is based on the Common Lisp programming language running on a Symbolics, Inc., workstation. In addition to Commom Lisp, the system will offer a number of other AI languages to allow system designers to change language in the middle of the design process if another language seems better suited to the component being created. These will include MRS from Stanford University, Carnegie Representation Language, the OPS5 programming language, and the Prolog programming language. In effect, these AI language environments require that new systems be created on specialized hardware, or that existing conventional systems be ported to them if integration is desired.

Hybrid Integration. This approach can be divided into three integration paths, producing systems that are by nature hybrid and which are linked through either a network, hardware, or software. Network-hybrid systems combine an AI machine and AI language environment with conventional systems and require complex interface software. The three vendors that manufacture symbolic processing hardware (Texas Instruments, Symbolics, and Xerox) all offer this solution to the integration problem

[56]. For example, American Express is experimenting with an ES to facilitate charge approvals called the Authorizer's Assistant [57,58]. The system uses approximately 800, forward-chaining rules to summarize a credit recommendation, a cardholder's creditworthiness, and to justify its recommendations. A Symbolics 3645 computer running Inference's Automatic Reasoning Tool (ART), a Lisp-based ES toolkit, accesses databases on two IBM 3033 mainframe computers running IMS and TPF operating systems. The Symbolics machine runs Ethernet, local-area-network protocols; and, the IBM mainframes operate under the System Network Architecture environment. The best way available at that time to connect the two networking environments was to use a Sun Microsystems workstation as protocol converter. TI has implemented its Explorer Workstation as a network-hybrid system in a flexible manufacturing application at one of its manufacturing facilities. The operation is controlled in real time through a TI-built, M68000 micro-processor-based computer with a DEC PDP 11/24 minicomputer for supervisory control and collision avoidance. A TI Explorer Lisp machine is used to run the ES that synchronizes operations, such as scheduling, dispatching, and tool loading [59]. Xerox's Lisp machines have also been integrated using this approach [60].

Hardware-hybrid integration is characterized by a machine that can run conventional and AI languages

efficiently. This is accomplished by taking a conventional hardware platform and adding a Lisp-optimized microprocessor as a co-processor to the system. Hewlett Packard's new Spectrum line of mini-computers have been designed with this in mind [61]. TI is a leading advocate in support of this approach. TI has developed a co-processor which contains approximately 60% of the circuits found in TI's full blown Explorer Lisp machine, as part of a Department of Defense contract [62,63,64]. TI's approach has been made commercially available through its microExplorer desktop computer, an Apple Macintosh II PC, which incorporates a specially developed co-processor board that includes the TI-Lisp microprocessor and explorer software environment [65].

Software-hybrid integration is the use of AI-conventional mixed language programming on a conventional computer. Bruno, et al., used OPS5, a rule-based production system interpreter, and Fortran to develop a production scheduling system on a DEC VAX computer for a flexible manufacturing system [66]. They report difficulty in maintaining data integrity between the two types of programs, which involves the use of external Fortran function calls from OPS5 and the use of VAX/VMS system traps to interrupt the program for real time rescheduling. Mark C. Paxton [67] discusses in detail the use of PC Scheme's External Language Interface (XLI) to build hybrid,

multilingual, ESs using PC Scheme and Turbo Pascal from Borland International, Inc. PC Scheme is TI's Implementation of Massachusetts Institute of Technology's Scheme Lisp programming language dialect, on which TI has based its Personal Consultant series of ES shells. The XLI interface is a memory resident routine, written in the C programming language which handles execution of externally executable programs and inter-program communication. XLI is dormant until called from PC Scheme during normal processing or when PC Scheme terminates. XLI requires three structures to be constructed in the user's programming language: a file block containing general information regarding the properties of the external program, a parameter block used to store information being passed back and forth, and a command table identifying names of routines in the external program. Paxton discusses various complexities, such as the programmer's responsibility for the consistent use of program-variables' lengths (e.g., XLI-passed strings have no sentinel character), and the inconsistent use of interface conventions for near and far program memory models.

Integration By Porting. This attempt at the solution to the integration problem ports an ES shell or development environment from an AI machine to a conventional machine, typically by rewriting the system in a conventional

programming language such as C. The C programming language is often chosen in light of its portability to assure that software developed will not be limited to chosen hardware [68]. Three of the four leading toolkit vendors (Carnegie Group, Inference, and Teknowledge; the exception is market leader Intellicorp) have chosen the porting approach to make their products more relevant and available to end users (and to the environments in which they operate) [69,70]. For example, Inference Corporation now offers its ART Automated Reasoning Tool in both Lisp and C in order "to gain considerable performance on traditional mainstream computer equipment and to make it easier to fit an ES into an existing environment", according to the company's vice-president and chief technical officer Chuck Williams [71,72]. Additionally, the company is under contract to NASA to build an ADA based version of ART. Such more-easily-modifiable software is especially important to large corporate customers who rely heavily on their own data-processing departments or third-party software houses to tailor applications to their employees' needs [73]. Some shells running on conventional hardware are fully integrated within themselves, e.g., supporting access to built-in word processors, data communications, spreadsheets, and related utilities. Though, more common is the break-out to other programs or the running of batched operating system commands. Some ported shells can

import and export fairly popular file formats, such as Lotus' 1-2-3 spreadsheet files or Ashton-Tate's dBase files [74].

Embedded Integration. Embedded systems are written in a conventional language that can be fully integrated into the application as an integral part of the conventional software system. The Lockheed Expert System (LES) under study at the Research Laboratories of Lockheed Missiles & Space Company utilizes this approach and is being written in ADA, the computer programming language specified by the Department of Defense [75]. Distribution Management Systems, Inc. (DMS), is working on an ES shell for embedding ESs in existing Cobol applications. According to DMS's Chairman John B. Landry, "The language best suited for an ES is the one that allows the user to write a program that calls the inference engine." [76]. However, according to Frederick Hayes-Roth of Teknowledge, the disadvantage of approaching the integration issue by attempting to embed the technology of ESs within existing applications is that a high degree of customization will be necessary [77]. The shell will need to be accessed as subroutines, but data structures and inference schemes will be inflexible. Yet, Butler, et al., see the buried ES as the third phase in the evolution of ESs [78]. Phase one consisted of the early exploratory development of ESs

using AI languages. Phase two stressed the implementation using ES shells allowing users to develop knowledge bases in a high level language format. Now at the threshold of phase three, organizations are debating how to integrate the technology of ESs into operational environments. Many other researchers, authors and practitioners are making the same observation [79,80,81,82,83,84]. As the mysticism surrounding ESs is slowly stripped away, a greater understanding of the technology and techniques of ESs is resulting. Such understanding argues increasingly for implementing ESs with conventional hardware and software, thus allowing development of fully integrated applications. There are many advantages to this strategy. No specialized hardware is needed, no new languages or operating systems need to be learned, and no additional software has to be acquired. Additionally, the modular approach advocated, developed, and implemented by this thesis effort allows for as little or as much ES technology to be incorporated as necessary, without any unnecessary overhead, and at the speed of conventional computers. Moreover, this approach appeals to the non-AI programmer. The engineer responsible for the entire application can also handle the ES portion. Admittedly, some customization may be required. However, with a careful analysis and synthesis of modules this customization can be minimized. In the long term as ESs proliferate and the technology matures, knowledge

representation and inferencing techniques may develop into standard, compatible, and interchangeable modules of algorithms and data structures.

Methodology

A macro-to-micro systems analysis approach has been the general outline for the methodology used in this thesis work. However, a bottom-up approach to the development of the modules was also frequently employed. The problem was attacked by division into five, relatively sequential, steps. Additionally, because this thesis effort involved a large amount of software development, a schedule was used to expedite the timely completion of all required work.

The first step was to develop a small, general-purpose, prototype ES-shell in the C programming language on an IBM-compatible, personal computer (PC). The purpose of this step was to gain experience in the technology of ESs while gathering sources of information for the project. The C programming language was chosen (in addition to the decision to use a conventional programming language) due to its popularity and the availability of Microsoft Corporation's affordable and efficient PC implementation called Quick-C [85]. Quick-C provides the developer with an environment for rapid program development. The C programming language is a popular programming language and has been called the foundation of

today's software industry. New commercial software packages are largely coded in C [86]. The PC was chosen because of its popularity and success as a widely used computer platform in industry. Also, Microsoft's C programming language (MS-C) adheres to the new ANSI standard for C, and has been successfully ported to other popular hardware platforms, such as DEC's VAX series of computers [87]. MS-C has received excellent reviews as a language for software development, primarily due to the quality of the product and the Quick-C environment for rapid program development. MS-C is also highly interfaceable to MS-Fortran. Though C is gaining in popularity, Fortran still dominates in approximately 80 percent of the scientific and engineering community's applications [88]. Though no actual experiments for interfacing to Fortran were performed in the scope of this thesis, it is believed to be a valuable feature. A shareware product called MC-WINDOW was used to develop a user-friendly interface. MC-WINDOW consists of library calls in C programs which provide quick, multi-level windows and fast screen writing utilities.

The second step was to identify the various logical modules of ES technology that would be desirable for use in a conventional or existing application. This most time consuming step combined analysis of a large body of literature with synthesis of a design for ES technology to

be integrated into conventional computing applications in a modular fashion. After completion of analysis and synthesis, the third step involved implementing the resulting design for the modules in the C programming language. This implementation was performed using a bottom-up approach. A menu-driven driver to dispatch the modules was developed to aid in the design, development, and testing of individual modules and their interactions.

Step four involved the selection of a conventional application for integration of the modules. This step actually overlapped early stages of the project so that development of the conventional part of the application might allow for a timely completion, synchronized with the development and completion of the modules.

Integration, the fifth and final step, was the climax of this thesis, i.e., the implementation and testing of the modules in the conventional application chosen in step four.

CHAPTER 3

ANALYSIS AND SYNTHESIS

Components Of An Expert System

A rule-based ES consists of three major elements in which there is usually a clean separation, namely, a set of production rules, a global database, and a control system. The global database is also often referred to as "working memory", a term borrowed from its use in the OPS5 programming language. The term working memory also avoids confusion with the term "database" used with database management systems [89,90,91]. A look at these three, basic components in greater detail is now necessary for further analysis.

Production Rules

These rules of thumb or heuristics are also commonly known as an ES's knowledge base. The knowledge base embodies some expert's procedural knowledge for the solution of problems belonging to a specific problem domain, and is represented in the form of production rules. These rules are highly domain specific, i.e., they represent a great deal of knowledge within a narrow range, rather than something about everything. A production rule

consists of a premise or condition part and a conclusion part. If the condition part of the rule is satisfied by facts in the working memory, then the rule becomes applicable and may be selected by the control system for application. If the control system selects or fires the rule, facts obtained from the conclusion part of the rule are added to the working memory. The production rules can be seen as static knowledge on how to solve problems, but, of which only a small portion are used to solve a particular instance of a problem [92,93,94,95].

Working Memory

The working memory consists of knowledge or facts which have been established for a particular instance of a problem. Working memory reflects the current state of the system, a type of factual knowledge valid only for the current instance of a problem. Thus, this memory is dynamic and of short term, and different for each problem that the set of production rules is able to attack. The set of rules comprising the knowledge base contains more than the number of rules required to solve a particular instance of a problem. Therefore, during a particular application of the rules only a subset is used. Each particular application is referred to as an instantiation. The facts in working memory are only valid during the instantiation in which they were generated [96,97,98,99].

Control System

The control system is a computer program which has a task to complete: the proof of a goal. The control system is also often referred to as an inference engine or reasoning mechanism. There are two general classes of control systems, namely, forward-reasoning control systems (a bottom-up approach), and backward-reasoning control systems (a top-down approach). These are also commonly called forward and backward-chaining control systems. Systems that reason forward, start with facts in working memory and apply all rules that are applicable until a termination condition has been reached involving the firing of a rule proving the goal, or when a termination due to failure occurs, i.e., due to a lack of applicable rules. Systems that reason backward, start with a goal and retrieve all rules that could satisfy the goal. These are rules that prove the goal by concluding a value for the goal in their conclusion parts. If none of the rules retrieved are applicable, i.e., none of the retrieved rules can be fired to prove the goal, then the system will generate new goals from the retrieved rules. These new goals are extracted from the condition parts of the previously retrieved rules in such a way, that if a new goal extracted is satisfied by the working memory, then the rule will become applicable allowing the original goal to be proved. The system terminates when an applicable rule

has been fired involving the goal or when no more applicable rules can be found. Both systems require facts in working memory, otherwise the control system will be unable to prove the goal and terminate unsuccessfully. For this purpose, the working memory is usually initialized in forward-reasoning systems, while in backward-reasoning systems facts are usually added to working memory interactively. Nonetheless, forward-reasoning systems can also be designed to add facts to working memory incrementally through special decision and search control mechanisms designed into the knowledge base and control system [100,101]. The user interface for the interactive addition of facts to working memory is presented in the form of a question or prompt, and is typically constructed from attributes assigned to rule conditions. Forward and backward-reasoning can also possibly be combined into a single control system. An important issue in designing the control system is conflict resolution, selection of the next rule to apply when more than one rule may be applicable [102,103,104].

The control system is the guts of an ES, i.e., the actual procedural program that operates on the production rules in the knowledge base and the facts in working memory until a termination condition involving the goal has been satisfied. Thus, in effect the term "control system" is a conglomerate term for the rest of the system that must

synthesize something from the knowledge base and working memory. Therefore, the control system is where analysis must be concentrated. Neither the knowledge base nor the working memory lend themselves to much further analysis or reduction to smaller parts; they already are basic building blocks. But the control system, on the other hand, is decomposable into smaller pieces.

The control system must consist of at least two parts with differing tasks. The first part must consist of an algorithm determining the overall control strategy, i.e., top-down or bottom-up approach, combined with a strategy for selecting which rule to apply next, i.e., a conflict resolution strategy. The second part must consist of routines to perform manipulation of knowledge base and working memory.

The control strategy must employ at least one of the two classes of control strategies (forward-reasoning and backward-reasoning), but it may also use both simultaneously. Nilsson [105] describes the use of a bidirectional control strategy, and the dangers involved with a poor selection of conflict resolution heuristics.

There are many conflict resolution strategies or policies, both uninformed and based on heuristics. Selecting a conflict resolution strategy basically involves choosing an appropriate search technique and combining it with a means to evaluate rules for applicability. Nilsson

[106] discusses a large number of search strategies for use in production systems. Both Rich [107] and Pearl [108] describe these algorithms from a practical or more implementation-oriented point-of-view.

A functional control strategy requires access to both the knowledge base and working memory. The control system must continuously query the knowledge base and working memory, and occasionally it must add facts to the working memory. Therefore, routines to perform these queries and additions must exist. Additionally, prior to any instantiation, the working memory must be initialized and the knowledge base must have been created. Thus, routines to add to the knowledge base must also exist. In fact, a user interface is required to make any program useful, and additional routines must be accessible to such an interface for the construction of the knowledge base and for the initialization of working memory.

Analytically, the control system can be dissected into three parts, (1) the control system embodying a resolution strategy, (2) manipulation routines for both knowledge base and working memory, and (3) a general user interface to create the knowledge base, and to initialize the working memory prior to instantiation and/or incrementally add to the working memory during instantiation.

Layers As A Framework For Integration

The development of an ES technology consisting of modules suggests the need for a clean and fresh approach. The architecture must be open, i.e., working memory and knowledge base must be accessible from routines other than those calling from the control system. Both the knowledge base and working memory are types of databases, and are specifically handled as database structures within a computer. Moreover, working memory and knowledge base must be viewed as databases open for questioning, analysis, and manipulation by routines other than the control system. Thus, if viewed as databases, both working memory and knowledge base must be accompanied by a set of low-level routines that can be used to perform manipulation functions normally associated with database management systems. This then suggests a three-tier or three-layer framework for the design of the ES technology modules for integration. This framework is illustrated in Figure 1 and is discussed below.

Database Layer

The foundation or bottom layer is a database layer consisting of two data structures, one for working memory and one for knowledge base. These two structures are repositories for data in the same sense that databases are repositories for data in database management systems. They

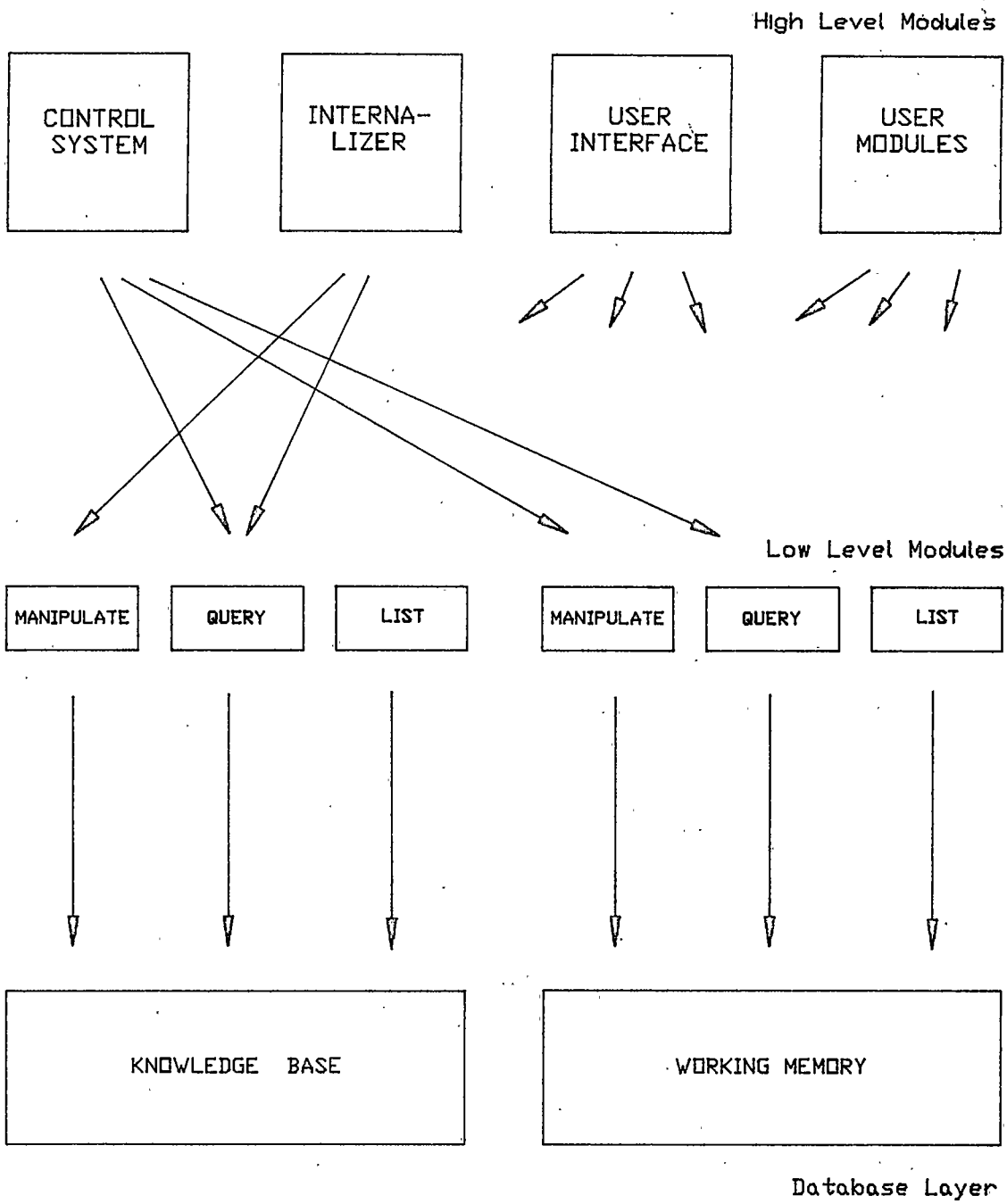


Figure 1. Three layer framework for the design of expert system technology modules for integration.

may be unconventional with respect to the type of data they store, however, their function is the same as in any database. These two structures are specialized systems for efficient data storage and retrieval.

Ideally, the database layer should be designed for maximum flexibility. Any control system should be able to operate on it without sacrificing efficiency or effectiveness. In the database-management world, there are currently three, dominant, database-architecture types, namely, hierarchical, network, and relational. It is likely that in the ES world of databases, eventually some architecture types for rule-based representation of knowledge will also become dominant.

Low-Level Modules

A second level consists of modular sets of low-level, database-maintenance routines, i.e., those routines that would normally be found in a formal, database-management system. These routines are classified as low-level modules because they perform basic, repetitive, and specialized functions. They consist of small and simple-to-implement algorithms. They can be constructed from well-established, proved, and tested technologies. Such routines typically include basic functions that will add, change, delete, query or list database information. These low-level modules maintain the knowledge base and working

memory as databases, and are the only connection between the bottom, foundation layer, and the top layer.

Of course, as in any database management system, database integrity is of concern. Although integrity can be made the user's responsibility, it is also possible to use the low-level modules as a screen to prevent database anomalies from occurring. For example, a global flag can be established to set an application's requirements to monotonic or non-monotonic reasoning. Low-level modules effecting changes to working memory can be required to update working memory based on a monotonic reasoning flag. If monotonic reasoning is required, then changes to established working memory facts are illegal.

High-Level Modules

A third or top layer, consists of high-level modules which embody logic of the particular reasoning methodology to be applied. These modules are categorized as high-level modules because the procedures that they employ are the actual algorithms from which an ES derives its expert capabilities. For example, these modules could contain various types of control system schemes combined with blind or informed search strategies. The key concept is that these modules only embody high-level algorithms, the actual work on the databases is performed by the routines in the appropriate, low-level modules.

At this same high-level, there may be any number of modules embodying other logic. In the case of an ES shell, such other modules may certainly include a user interface and a module to initialize or construct databases. But in case of an integrated application, some of these modules may consist of the conventional application. User modules of a conventional application can access the knowledge base and working memory in the foundation layer through the low-level modules.

Inter-Application Communication

The three-tier framework for the design of ES technology modules offers a solution to the problem of designing an efficient means of integration. But effectiveness must also be addressed. Effectiveness in this case refers to how well the design can accomplish the task of integrating ES technology with conventional computing. Thus, the emphasis of the design task is integration. Conventional applications embody knowledge in algorithms consisting of hierarchical procedural programs. ESs embody their procedural knowledge in independent and non-hierarchical rules. Integration from an effectiveness standpoint must refer to the communication between these two types of programming. Each approach to problem solving, i.e., rule-based versus conventional, must be assigned to the task element it is most suited to or

qualified to perform. In performing their respective tasks, there must be sharing of intermediate and final results. However, communication does not just refer to the sharing of data alone. Because execution on a conventional computer is sequential, control must pass from rule-based programming to hierarchical programming and vice-versa. This can only occur if there is a means for relinquishing and resuming control for these two approaches to problem solving. Therefore, inter-application communication refers to sharing of data and transfer of control.

The three-tier framework approach for the design of the modules offers a simple way for the sharing of data. Because of the open architecture, low-level routines are accessible by the conventional application allowing communication through this common access to data. However, if transfer of control is to occur at any time during the operation of an integrated application, then it must be designed specifically into a reasoning module. Otherwise, transfer of control can only occur before and after complete knowledge base instantiations, i.e., before and after a control system has been given control of execution. Thus, a reasoning module must have a specific feature that allows for the transfer of control to a conventional hierarchical program procedure and back. For example, in a backward-reasoning control system, the most likely means for implementing transfer of control is by

designing relinquishment of control into a ready-made mechanism, i.e., the user interface. To avoid backward-reasoning control systems from terminating unsuccessfully, the user interactively adds required facts to working memory through the user interface. However, this suggests an elegant but simple means for transferring control: instead of relying on the user interface, allow a procedural program to add the required fact to working memory through a function call.

CHAPTER 4

DEVELOPMENT AND IMPLEMENTATION

Expert System Technology Modules

Formal engineering and systems analysis procedures typically call for a stepwise approach to the design process. A developer will proceed sequentially through various stages of analysis followed by many steps of synthesis. The development and implementation process for the ES technology modules was at times formal and at times experimental, and at most times paralleled analysis and synthesis. The original prototype developed prior to the analysis stage was based on a number of simplified ES shells [109,110,111,112]. Features included a depth-first search algorithm incorporating backtracking and a linear search to select rules for application which were read into a simple array structure serving as knowledge base. Prototype development allowed for a familiarization with ES shell programming techniques and for the learning of the C programming language. Later, the prototype was enhanced extensively and served as the basis for four modules, namely, depth-first backward-reasoning, query working memory, query knowledge base, and manipulate working

memory. In the final implementation, many underlying design concepts were borrowed from the Mycin and Emycin architectures, due to their apparent popularity, success, and the relatively large amount of literature available on Mycin and Emycin [113].

Database Layer

Knowledge Base. The knowledge base is a database that is implemented as a data structure. This data structure must meet several requirements. It must store the production rules in such a way that the control mechanism can rapidly select individual rules for evaluation based on a search algorithm. If the data structure uses an internal representation of the production rules, then there must be a capability to reproduce the production rules for the purpose of explaining how a particular goal was satisfied. The data structure must also be suited for basic database manipulation activities. Additionally, the data structure must also be able to accommodate both small and large numbers of rules without sacrificing efficiency. There are probably many suitable data structures that can meet these requirements, but only one will likely be optimal for a specific application. In database management systems the relational architecture appears to be preferable over hierarchical and network designs, but, this was not established until after many years of practical experience.

Binary trees were chosen to be used as the basic data structure for the knowledge base as a compromise between space and time complexities, ease of implementation, and programming experience.

The binary tree data structure implementation of the knowledge base is depicted in Figure 2. The knowledge base was designed to accommodate knowledge bases divided into chunks called contexts, thereby increasing search efficiency for large knowledge bases. The division of the knowledge base into contexts also facilitates knowledge base content design and construction by permitting the knowledge engineer to use a top-down design approach. For this purpose, each context is a binary tree of production rules, and all contexts are tied together in a super binary tree called the context tree. The nodes of the context binary tree represent the goals for each context within the context tree. These context goals are also the search keys in the context tree. The use of a context tree to tie knowledge bases together allows for contexts to instantiate each other when a goal within a context cannot be satisfied, but occurs somewhere on the context tree as a search key, i.e., a goal to another context. The context tree was first implemented in Mycin; however, Mycin's authors are vague regarding all references to the actual data structure used for its implementation [114]. Each context within the context tree is a binary tree of

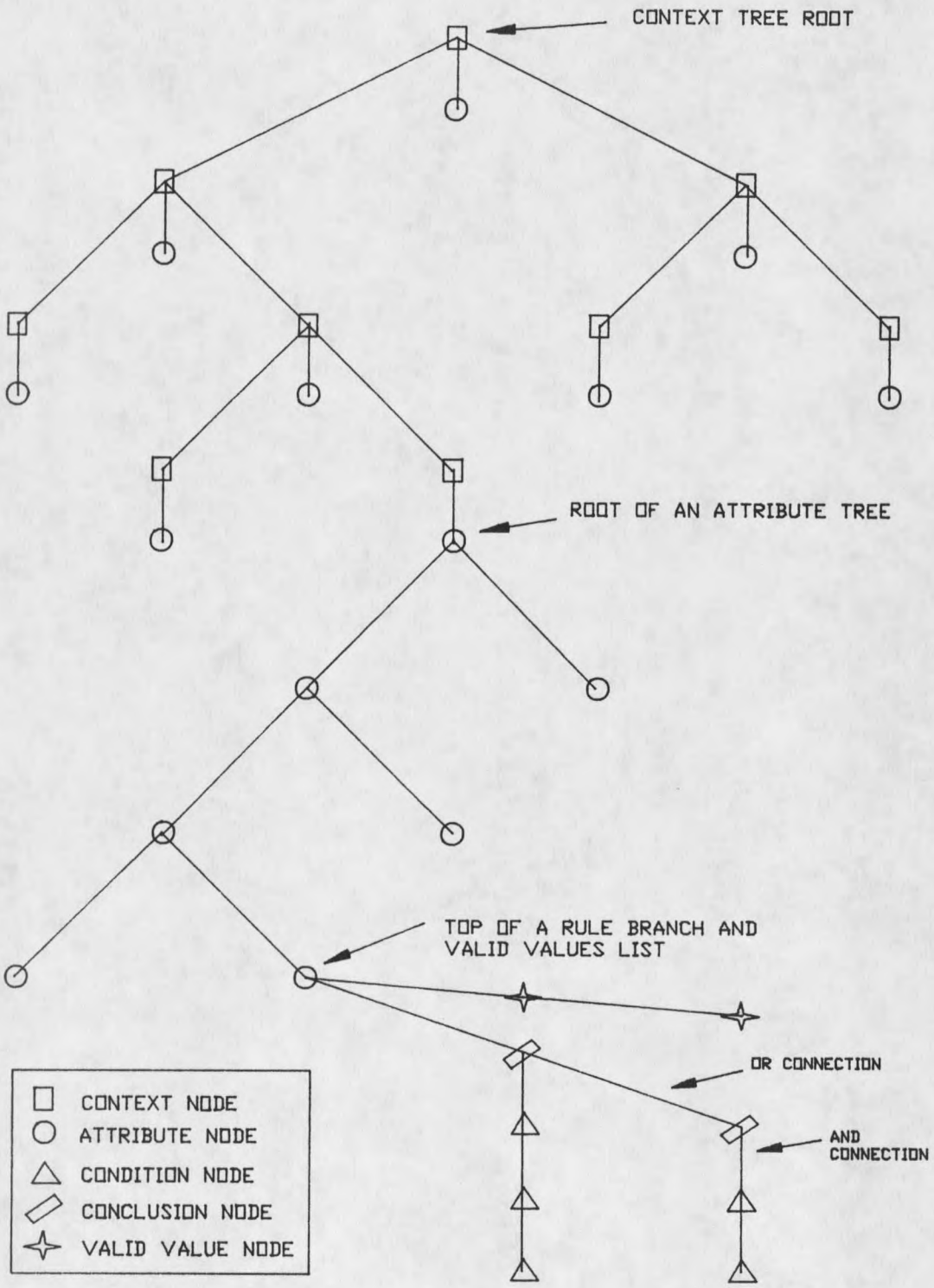


Figure 2. Knowledge base implemented as attribute binary trees linked together through a super context binary tree.

production rules. To optimize the production rule trees for backward-reasoning control systems, each node on these trees has a conclusion attribute of a production rule as a search key. The production-rule binary tree of a context is, therefore, called an attribute tree, and its design is unique to this implementation. By placing the conclusion attributes of production rules on the nodes, a backward-reasoning control mechanism can rapidly search an attribute tree for applicable rules. Several linear lists and other information are attached to each node of the attribute tree. All rule conditions concluding in a single attribute type form a rule-branch, linear list, with all "and" conditions concluding in a specific attribute value connected together as a linear list, where the head of the list is connected to sister linear lists of "and" conditions concluding in other values for the same attribute through "or" connectors. All values which an attribute can attain form another linear list, and parameters such as prompts and English-like translations are also attached to attribute tree nodes. English-like translations for attributes were first introduced in Mycin [115]. In Mycin the triplet of a context possessing many attributes, each having values, was called the object-attribute-value triplet. The data structures for the binary trees are based on implementations discussed by Robert L. Kruse [116]. The C language implementation of

the knowledge base data structure is listed in Appendix A as the header file NODES.H1.

Working Memory. Several criteria exist for the selection of a data structure to implement the working memory as a database. Just as with the knowledge base, the control mechanism must be able to search the working memory rapidly for facts. As with the knowledge base, the data structure must also be suited for basic database manipulation activities. Also, each fact in the working memory must have a recallable memory of how it was obtained. A binary tree data structure was also chosen to implement the database for the working memory. Each node of the working memory binary tree has an attribute as key. Other information is also stored on each working memory tree node, including the information to allow explanation of how working memory facts were obtained. The C language implementation of the working memory data structure is listed in Appendix A as the header file NODES.H1.

Low-Level Modules

Knowledge Base Maintenance. The modules to maintain the knowledge base consist of routines for binary tree traversal, search, and insertion. Kruse [117] discusses the implementation of binary tree algorithms in detail. The implementation of these routines is based directly on

these discussions. Routines for change and deletion would be required in a full implementation of modules, but they were not needed in any of the implementations developed for this thesis. Also, the knowledge base maintenance modules access the internalized data structure of the knowledge base used at run time; actual knowledge base development, including changes and deletions, occurs externally to an intermediate knowledge base file, an ASCII text file, using an editor. This approach was the most attractive with respect to programming effort and user interface. Full screen editors provide a user-friendly means to develop a knowledge base and are readily available in the public domain. Thus, rather than developing the change and deletion routines, which would at most provide a poor user interface, a parser was developed to reap the benefits of a full screen editor for knowledge base development. The parser is discussed below as the knowledge base internalizer. A sample low-level knowledge base maintenance module, i.e., Q_KB.C (Query Knowledge Base), is listed in Appendix B.

Working Memory Maintenance. Because the working memory database is also based on a binary tree, routines for working memory maintenance use the same principles as those developed for the knowledge base. Just as with the knowledge base, routines for changes and deletion were also

omitted as they were not required by any of the implementations. A global flag was implemented to indicate whether or not monotonic reasoning was desired. If monotonic reasoning is desired, changes are illegal; once values are assigned to attributes in working memory they become established as non-modifiable facts. Otherwise, attribute values can be altered as desired or required by the application. Two, sample-C-language implementations of low-level working memory maintenance modules have been included in Appendix B, i.e., M_WM.C (Manipulate Working Memory), and Q_WM.C (Query Working Memory). These two modules, together with the module Q_KB.C referred to above, are required by the backward, depth-first, control system (DEPTH.C) C-language implementation discussed below and listed in Appendix C.

High-Level Modules

Internalize Knowledge Base. To be able to access the production rules of the knowledge base efficiently, a specialized data structure was developed called the context binary tree. This structure is optimized for the computer and not humans. For this reason, an additional intermediate form of the knowledge base was adopted to allow for the knowledge base to be developed in a user-friendly manner. An ASCII-type text file was chosen so that any full screen editor could be used to develop the

knowledge base. A language parser was chosen as the means to internalize this intermediate knowledge base into a computer efficient, internal, tree structure. A predictive, recursive descent parser was implemented, based on a LL(1) grammar using First and Follow functions. Aho, et al., [118] describe the implementation of such parsers in great detail. Though, not always very time and space efficient, recursive descent parsers provide a relatively straightforward and flexible top-down approach to designing and implementing parsers, and have been in use since the early 1960s [119,120,121]. Ironically, all recursions in the implementation of this parser were tail recursions and were, therefore, removed and replaced with looping constructs. The LL(1) grammar developed to describe the language of the parser is illustrated using BNF notation in Table 1.

Control Systems. Two exhaustive variants of one control system type were implemented, namely, backward-reasoning using a depth-first search with backtracking technique for conflict resolution, and a backward-reasoning mechanism with a breadth-first-search, conflict-resolution strategy. Judea Pearl describes the implementation of depth-first and breadth-first search algorithms in [122]. A source code listing of the C language implementation for the backward depth-first control system is included in

Table 1. BNF notation for the grammar of the parser.

```
⟨kb⟩          ::= GOAL IS ⟨attr⟩ ⟨context⟩ EOF | GOAL IS ⟨attr⟩ ⟨context⟩ ⟨kb⟩
⟨context⟩     ::= ⟨statement⟩ | ⟨statement⟩ ⟨context⟩
⟨statement⟩   ::= ⟨comment⟩ | ⟨rule⟩ | ⟨title⟩ | ⟨epilog⟩ | ⟨prompt⟩ | ⟨trans⟩ |
                  ⟨valid_val⟩
⟨comment⟩     ::= / [ascii chars except /] /
⟨title⟩       ::= TITLE IS ⟨text⟩
⟨epilog⟩      ::= EPILOG IS ⟨text⟩
⟨prompt⟩      ::= PROMPT ⟨attr⟩ ⟨text⟩
⟨trans⟩       ::= TRANS ⟨attr⟩ ⟨sentence⟩
⟨valid_val⟩   ::= VALID_VAL ⟨attr⟩ ⟨val⟩
⟨rule⟩        ::= ⟨condition⟩ ⟨conclusion⟩
⟨condition⟩   ::= IF ⟨attr⟩ ⟨oper⟩ ⟨val⟩ ⟨more_cond⟩
⟨more_cond⟩   ::= AND ⟨attr⟩ ⟨oper⟩ ⟨val⟩ ⟨more_cond⟩ |
                  OR ⟨attr⟩ ⟨oper⟩ ⟨val⟩ ⟨more_cond⟩ | NULL
⟨conclusion⟩  ::= THEN ⟨attr⟩ IS ⟨val⟩ | THEN ⟨attr⟩ EQ ⟨math⟩
⟨attr⟩        ::= ⟨word⟩ | ⟨sentence⟩
⟨oper⟩        ::= EQ | GE | GT | LE | LT | NE | IS | IS NOT
⟨val⟩         ::= ⟨word⟩ | ⟨sentence⟩
⟨math⟩        ::= " ⟨word⟩ ⟨more_math⟩ "
⟨more_math⟩   ::= ⟨math_oper⟩ ⟨word⟩ ⟨more_math⟩ | NULL
⟨math_oper⟩   ::= + | - | * | /
⟨word⟩        ::= Any combination of ascii characters except /, " and spaces.
⟨sentence⟩    ::= " [ascii chars except "] "
⟨text⟩        ::= " [ascii chars except " where ^ is a forced end of line] "
```

Appendix C. The three, low-level modules that it uses for database access are listed in Appendix B. Two global flags were established for the depth-first search variant. The first controls the type of backtracking to be employed, i.e., prompt or leaf backtracking. Prompt backtracking specifies that if no value for an attribute can be established and there does not exist a prompt for the user to update the working memory with a fact, then backtrack until a new path can be found to explore. Leaf backtracking specifies that backtracking is only to occur after every leaf has been explored; if no user interface

exists at a leaf, then construct a prompt. The second global flag controls the order by which individual "and" conditions within a rule are expanded for exploration. Both variants have the ability to relinquish and resume control of execution to and from a routine of a conventional application, as required. This was implemented by recognizing null values assigned to prompts as indicating a desire of a rule to transfer control to a conventional procedure for the purpose of establishing a fact. Both variants also have the ability to search the context tree for unresolved goals, based on a global flag called the cascading context instantiation flag.

Implementation Of Modules As An Expert System Shell

The driver originally used to design, develop, and test the individual ES technology modules was later expanded with a complete user interface and developed into a stand-alone ES shell. This extension had two purposes, namely, to allow the development of stand-alone ES applications, and, also for development and debugging of knowledge bases to be used in integrated applications. The shell is called ESTD for Expert System Technology Driver. It is important to note that more than half of the code comprising this full-fledged ES shell consists of the user interface, despite the use of prepackaged menuing and windowing utilities in its construction. This demonstrates

the magnitude of unwanted overhead inherited from shells providing break-out type of interfaces when they are used in integration schemes. Figure 3 illustrates this implementation with modules represented by blocks; the sizes of the blocks roughly correspond to the module sizes in the actual ESTD shell implementation.

ESTD's user interface consists of a horizontal main menu with four headings, each of which can be selected using a cursor. Each main menu option has its own vertical pull-down menu for the selection of an actual activity. The first main menu option corresponds to knowledge base activities, such as the loading of a knowledge base from a disk file into the internal context data structure, listing the internal data structure to a disk file, and instantiation of a context. The second main menu option corresponds to working memory activities, such as the explanation of concluded context goals and facts. The third main menu option gives the user the flexibility to select from a number of conflict resolution strategies offered by the implementation of the two backward-reasoning control systems. Aside from the access to all the ES technology modules, with the fourth menu option the user can direct the driver to execute a user defined editor and a read-only file browser as child processes to allow knowledge base development and the viewing of context listings to occur as part of an integrated environment.

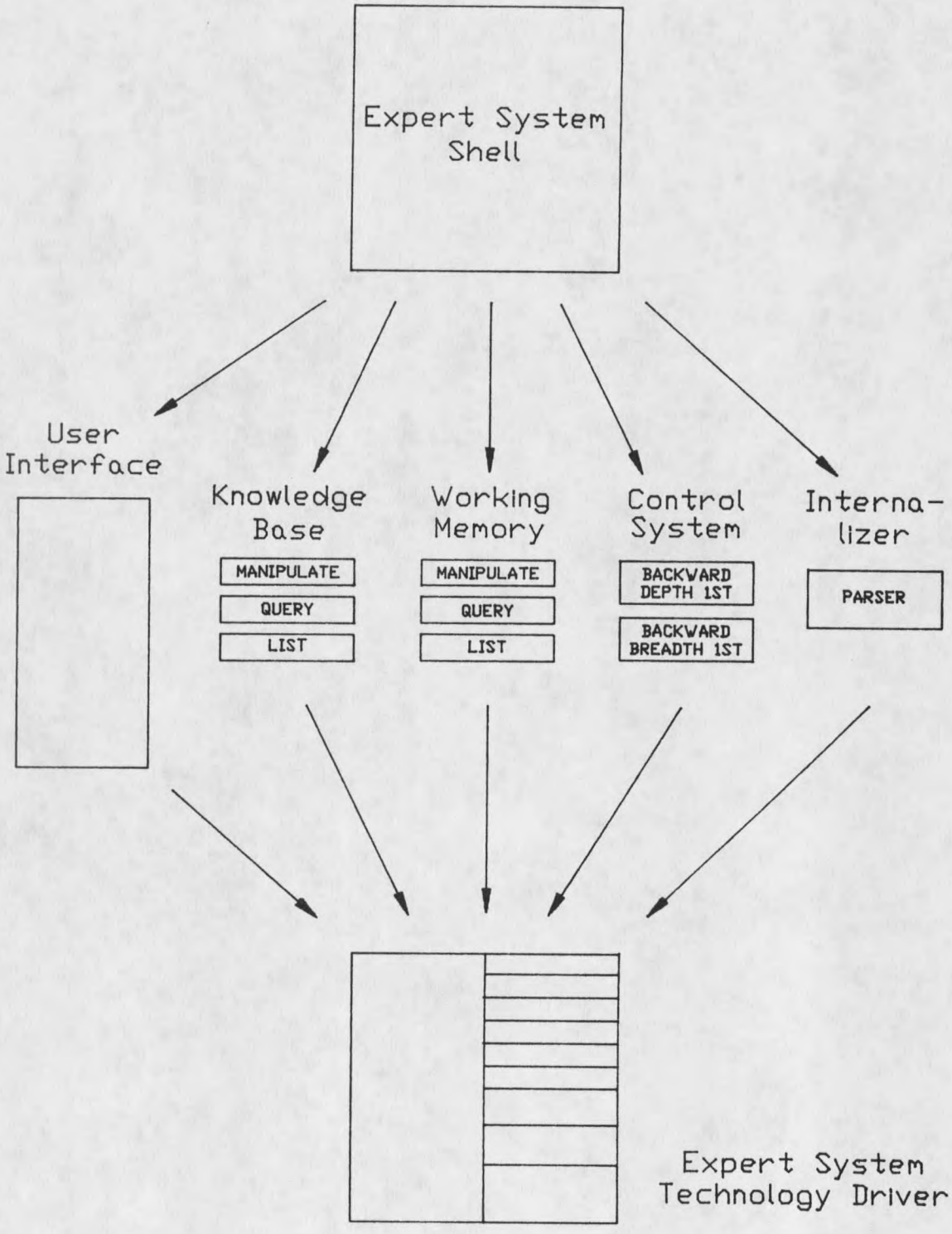


Figure 3. Expert system technology modules implemented as a skeletal shell.

The names of the editor and browser executables are fed to ESTD as command line parameters. The driver also has the ability to suppress all development menus in order to simulate the delivery of a completed ES application. This is triggered by the "/Auto" command line switch. Three other switches were also implemented. One, the "/Help" switch, gives command line usage. Another, the "/Grammar" switch, lists the parser's grammar, as shown in Table 1. Finally, the "/Keywords" switch, causes ESTD to list valid keywords the parser can match, together with keyword-type codes corresponding to those specified in the grammar's BNF notation. This listing is the formatted output of the actual array the parser uses in a binary search to match keywords and is, therefore, always current. The listing is shown in Table 2.

Integration Of Modules Into A Modeling Tool

The test ground for integration of the modules into a conventional application consisted of a modeling tool which is an outgrowth of two modeling tools originally developed by Dr. Donald W. Boyd of the Montana State University Department of Industrial and Management Engineering. The original tools were two Fortran programs called Trainer and Modeler, the outgrowth of the two combined with the ES technology modules is an experimental C program now called 501 Modeler. The modeling approach these tools use is one

Table 2. Keywords and codes corresponding to the grammar's BNF notation.

Keyword	Code	Keyword	Code	Keyword	Code
!=	NE	<	LT	<=	LE
<>	NE	=	EQ	>	GT
>=	GE	and	AND	are	IS
aren't	NOT	can	IS	can't	NOT
cannot	NOT	did	IS	didn't	NOT
do	IS	does	IS	doesn't	NOT
don't	NOT	epilog	EPILOG	equal	IS
equals	IS	goal	GOAL	has	IS
hasn't	NOT	have	IS	haven't	NOT
if	IF	implies	IS	imply	IS
indicate	IS	indicates	IS	infer	IS
infers	IS	intro	INTRO	is	IS
isn't	NOT	mean	IS	means	IS
not	NOT	or	OR	prompt	PROMPT
then	THEN	title	INTRO	trans	TRANS
valid_val	VALID_VAL	was	IS	wasn't	NOT
were	IS	weren't	NOT		

developed by Dr. Boyd which began with his Ph.D. research and dissertation in 1967 and 1968, and which created the basis for developing the first, operational, state water planning model of all 50 states. Following this success, techniques were generalized into a macro-to-micro methodology applicable to modeling any dynamic system. A graduate course employing this methodology was introduced by Dr. Boyd at Montana State University in 1981, I&ME 501 Development of Mathematical Planning Models [123].

The underlying mechanism on which this modeling approach is based is the ability to represent a non-linear system as a set of linear equations that can be solved simultaneously to produce exact values over incremental

periods of time. Each solution of the set of simultaneous equations represents the state of the system for the corresponding increment of time. The obstacle to this modeling approach lies in development of the model, i.e., the set of linear equations that represents the system to be modeled. Though this modeling technique achieves impressive results when compared to other approaches, the technique is so esoteric that it currently requires a minimum of a graduate level course in instruction prior to usage in order to become acquainted with the techniques involved in constructing a model. However, Boyd produced an excellent comprehensive summary of this approach to modeling [124].

The Trainer and Modeler programs were designed to develop and run models once constructed; they do not guide a user through the model development process. The purpose of integrating ES technology into this conventional application is to bring part of the external model development process, known as problem domain system analysis, into the modeling tool. Model development can be seen as a knowledge acquisition (KA) process, consisting of many independent and interdependent steps greatly necessitating the guiding control of an ES. This KA process is not the type associated with the development of ESs. Its goal is not to develop a knowledge base, rather, its goal is to develop a model. There are two reasons why

the integration of ES technology is beneficial to this application. For the user of the modeling tool the modeling process is simplified because he/she no longer needs to be concerned with which is the appropriate next systems analysis step; the ES assumes this responsibility. For the developer of the modeling tool, ES technology offers a new dimension in programming flexibility and ease of expandability. Because there are so many small activities involved in the KA process, the application is well suited for the integration of a rule-based control system to guide the application of a large number of small procedures. Figure 4 illustrates the integration of the ES technology modules, using the three-level framework developed in this thesis, with the user interface and algorithms required for Boyd's modeling methodology. Note how only a subset of the ES technology modules are used, and how the ES shell's user interface is replaced by high-level modules of Boyd's conventional application and the knowledge-acquisition user interface to create the 501 Modeler program.

501 Modeler consists of roughly three parts, i.e., the original algorithms used in the Trainer and Modeler programs to develop and run models, a set of ES technology modules, and a large number of relatively independent routines for KA. The KA routines are tied together through global variables and data structures which are common to

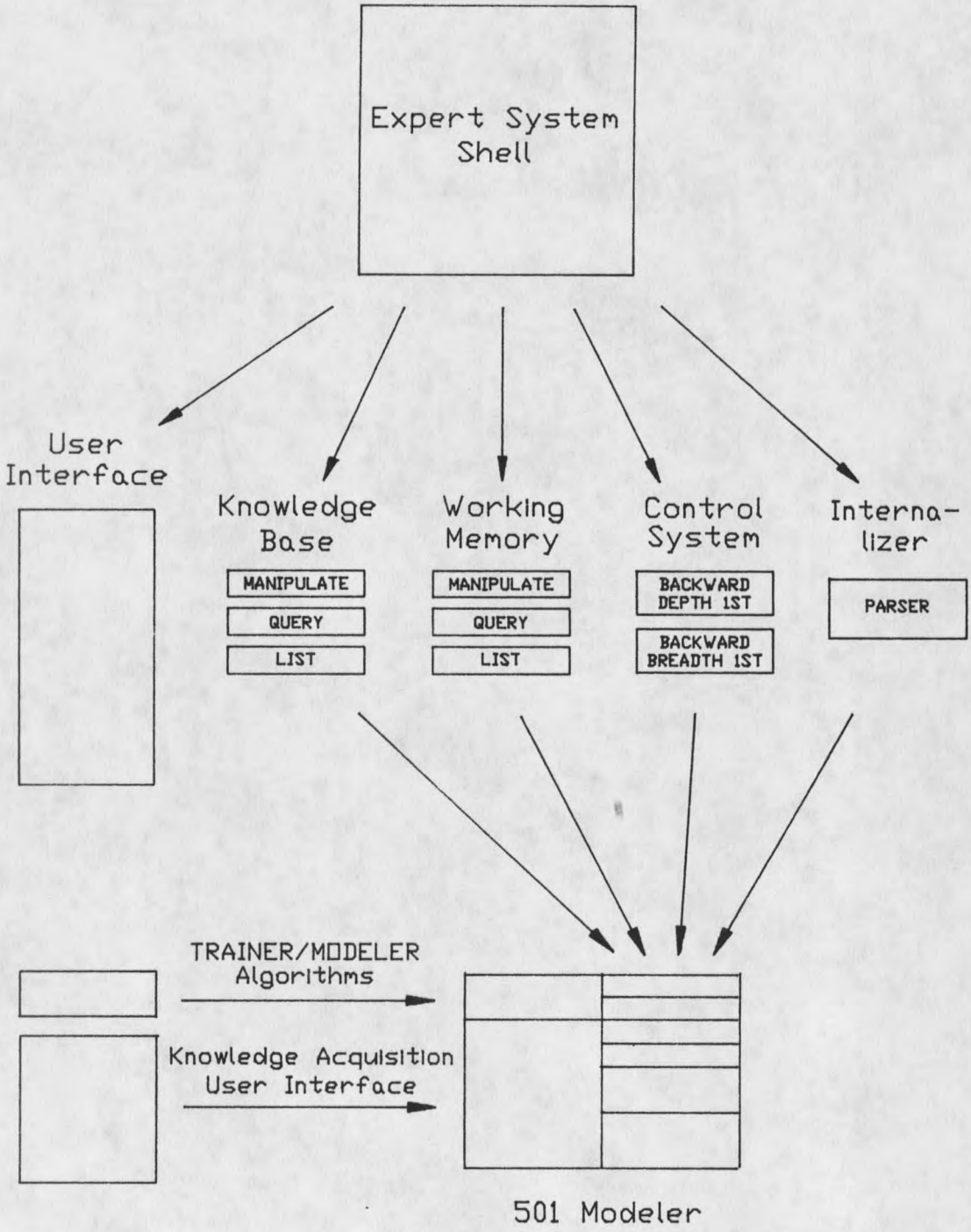


Figure 4. Expert system technology modules integrated into a conventional application.

all the routines. These global data structures act as a type of virtual database for the construction of a model. The goal of the system is to produce a valid model that can be run by the original algorithms. The ES modules work together with the KA routines to achieve this goal. Each KA routine corresponds to an attribute in a knowledge base which has the construction of a valid, runable model as goal. Under backward-reasoning control, rules in that knowledge base that could satisfy the goal, but, which are not applicable, are used to create new goals. Once an attribute corresponding to a KA routine becomes a goal, control of execution is transferred from the ES modules to the appropriate KA routine through a uniquely identified routine, executed by the control system that maps goal attributes to callable conventional routines. The knowledge base and mapping routine are listed in Appendix D. Appendix C lists the backward depth-first module, the control system employed in 501 Modeler, which calls the mapping routine for goal attributes identified by prompts with a null value.

Note that this approach to solving the KA problem requires that there are a large number of KA routines involved, otherwise, the solution to the problem would be trivial, a set of logical transfer-of-control statements in a conventional hierarchical program would be able to accomplish the task.

CHAPTER 5

SUMMARY AND CONCLUSIONS

This thesis work has resulted in a new design method for the integration of ES technology into conventional applications. The design method belongs to the category of embedded approaches to integration; its uniqueness is derived from the three-level framework for the design of the integration, which treats ES technology as data structures and algorithms, i.e., a pair of databases accompanied by two levels of modules.

The three-tier design is efficient. Integration on the part of the application developer consists of selecting the appropriate modules and creating an interface routine. There is minimal overhead on the part of the ES technology integrated, since only the modules required are selected for inclusion. By implementing the three-tier design in the traditional language of the conventional application, a match of performance is almost guaranteed.

The three-layer design is effective. The open architecture provided by the low-level modules lets a conventional application access knowledge base and working memory if required, allowing for the sharing of data. Transfer of control initiated from conventional application

to ES occurs through instantiation of a context. The attribute-routine matching function callable from the control system allows for transfer of control to be initiated by the ES.

The ESTD implementation of the modules as an ES shell illustrates the completeness, flexibility, and transparency of a set of modules designed for integration.

The 501 Modeler implementation of the C language modules illustrates the efficiency and effectiveness of the three-tier framework, and most importantly, it allows for a problem to be tackled not previously attempted.

The three-layer framework is very suitable. Compared to other integration alternatives, the three-layer framework offers many advantages. An investment in new hardware and software is not required. No new programming languages or operating systems need to be learned. Personnel do not have to be retrained. The technology is at the disposal of any application developer. The approach is flexible and easy to implement due to a hands-on treatment of ES technology. It is a potentially low programming effort approach, considering that, once the modules are developed, they will need only to be maintained.

GLOSSARY

Algorithm: A process that will solve a given problem; a formal procedure guaranteed to produce correct or optimal solutions.

And/Or Graph: An important aid useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems. This decomposition, or reduction, produces And arcs and Or arcs to subproblems. And arcs point to any number of successor problems, all of which must be solved in order for the arc to point to a solution. Or arcs point to single successors problems, each of which solved for will solve the parent solution.

Anomaly: Inconsistencies within a database conflicting with the rules governing the database brought about by an add, change or delete operation.

Artificial Intelligence: A cross disciplinary science, relying heavily on computer science, which is concerned with developing intelligent computer systems. These include systems that can solve problems, learn from experience, understand language, interpret visual scenes, and, in general, behave in a way that would be considered intelligent if observed in a human.

ASCII text file: A file containing textual information that represents characters according to the American Standard Code for Information Interchange. The term text file alone is confusing, as it could refer to an EBCDIC text file, the other most popular code used on IBM computers (except the IBM-PC, which uses ASCII).

Backward Chaining Or Reasoning: A control method that attempts to achieve goals recursively, first by enumerating premises that would be sufficient for goal attainment and second by attempting to achieve or establish the premises themselves as goals.

Backtracking: A search procedure that makes guesses at various points during problem solving, and which returns to a previous point to make another choice when a guess leads to an unacceptable result. It is a programming technique that allows a system to remove the effects of incorrect assumptions during its search for a solution to a problem.

As the system infers new information, it keeps dependency records of all its deductions and assumptions, showing how they were derived. When the system finds that an assumption was incorrect, it backtracks through the chains of inferences, removing conclusions based on the faulty assumption.

Binary Tree: A special case of a rooted, ordered tree. A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root.

BNF (Backus-Naur Form): A widely used notation for specifying the syntax of a language, specifically that of a context free grammar.

Breadth First Search: An exhaustive tree (or graph) search algorithm which searches its most shallow vertices first to find a goal.

C Programming Language: A compact, efficient, portable, readable, small, low-level and high-level, and expressive programming language. C was originally used for system programming (the UNIX operating system was written in C) but is now the top language for serious software development. The strength of C lies in its abscence of restrictions, i.e., its philosophy that programmers know what they are doing. However, this is also its weakness, creating a longer than normal learning curve and reserving the language for advanced programmers.

Child Process: If a process can create other processes, then these other processes are referred to as child processes.

Conflict Resolution: The technique of resolving the problem of multiple matches in a rule-based system. When more than one rule's antecedent matches the working memory, a conflict arises since every matched rule could appropriately be executed next, but, only one rule can actually be executed next. A common conflict resolution method is priority ordering, where each rule has an assigned priority and the highest priority rule that currently matches the working memory is executed next. Conflict resolution is an important component of the control strategy.

Control Or Inference Method: The overall technique used by the control system to access and apply the domain knowledge, e.g., forward-chaining and backward-chaining.

Control System Or Inference Engine: Any procedure, explicit or implicit, that determines the overall order of problem-solving activities. It is that part of a expert system that contains the general problem solving approach. The control system processes the domain knowledge (located in the knowledge base) to reach new conclusions.

Database: A set of systematized data relevant to some topic (theme) collected into a data storage system.

Database Integrity: The accuracy or validity of data and interdependent data in a database.

Depth-First Search: An exhaustive tree (or graph) search algorithm which searches through the deepest (or longest) paths to vertices first to find a goal. Depth-first search is always combined with backtracking.

Domain: A sphere or field of activity or influence of some commonality.

Domain Expert: A person who, through years of training and experience, has become extremely proficient at problem solving in a particular domain.

Domain Knowledge: Knowledge about the problem domain.

Exhaustive, Uninformed Or Blind Search Technique: A search for a goal involving some systematic but arbitrary scheme. This type of scheme will systematically search every possible path until a goal is found. Depending on the type of exhaustive search technique used, the outcome of the search may or may not be optimal.

Expert System: A computer system that achieves high levels of performance in narrow problem areas that, for human beings, require years of special education and training. These programs typically can explain their reasoning processes.

Expert System Shell Or Skeletal System: The programming language and support package used to build an expert system. In such a tool the control structure for the system already exists and all the user has to do is add the knowledge base. The price paid for this convenience is the loss of freedom and flexibility imposed by standard data structures and control schemes.

Expertise: The set of capabilities that underlies the high performance of human experts, including extensive domain knowledge, heuristic rules that simplify and improve

approaches to problem-solving, metaknowledge and metacognition, and compiled forms of behavior that afford great economy in skilled performance.

Explanation Facility: That part of an expert system that explains how solutions were reached and justifies the steps used to reach them.

Firing: The application of an applicable production rule on the working memory by the control structure.

Forward Chaining: An control method where the condition or premise portion of rules are matched against facts to establish new facts in order to eventually prove a goal.

Garbage Collection: The reposession process of memory once dynamically allocated but later no longer required.

Grammar: A grammar is a scheme for specifying the sentences allowed in a language, indicating the syntactic rules for combining words into well formed phrases and clauses. The syntax of a grammar expresses the rules of the grammar.

Header or Include File: A file which contents the C compiler preprocessor adds to another file when specified by a preprocessor directive. Useful for incorporating declarations of external variables, complex data types, and function prototypes, as once these are declared in a header file this header file can be included into a number of source files. Updates need only occur in one place.

Heuristic Or Heuristic Rule: A procedural tip or incomplete method for performing some task. A rule of thumb or simplification that limits the search for solutions in domains that are difficult and poorly understood.

Heuristic, Intelligent Or Informed Search: A search for a goal which is guided by search task dependent information. The purpose for the use of heuristic information is to reduce the effort required to perform the search for the goal.

Interpreter: A system that analyzes the next instruction to decide what actions to take next, and each instruction is executed as soon as it is analyzed.

Key: A unique identifier for some grouped together data.

Knowledge Acquisition: The process of extracting, structuring, and organizing knowledge from some source,

usually human experts, so it can be used in a program.

Knowledge Base: The repository of knowledge in a computer system. The portion of an expert system that contains the domain knowledge. It includes domain specific facts and heuristics useful for solving problems in the domain. It is a type of database containing soft data in addition to hard data, using some form of knowledge representation.

Knowledge Based System: Another term for an expert system, but, often more appealing as it less specific.

Knowledge Engineer: The person who designs and builds the expert system. This person is usually a computer scientist experienced in applied artificial intelligence methods, but, could also be the domain expert herself.

Knowledge Engineering: The discipline that addresses the task of building expert systems; the tools and methods that support the development of an expert system.

Knowledge Representation: The process of structuring knowledge about a problem in a way that makes the problem easier to solve. Production rules are a means to represent knowledge.

Language: The set of all properly constructed sentences that can be produced using a grammar corresponding to the language.

Lisp: The principal programming language of AI in the United States, which provides an elegant, recursive, untyped, and applicative framework for symbolic computing; it is actually a family of variants or dialects.

LL(1) Grammar: A grammar specifically suited for predictive recursive descent parsing. The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions without backtracking. It employs a First function to define the beginning of sentences, and a Follow function to define the lookahead symbols.

Macro-To-Micro Approach: A sequential systems analysis approach starting at a high level of abstraction and slowly incorporating more detail. A governing rule for the successful application of this approach is the principle of parsimony.

Monotonic Reasoning: A reasoning in which beliefs are consistent and not subject to change. Established facts are not modifiable.

Near/Far Memory Models: The segmented architecture of the 8086 family of microprocessors (IBM-PC and compatibles) requires memory to be addressed using near and far addresses. In near memory models program coding is contained within a single segment allowing for efficient near addressing, while in far memory models program coding is spread over more than one segment requiring far addressing. The data of a program can be spread respectively in a similar fashion.

Non-Monotonic Reasoning: A reasoning technique that supports multiple lines of reasoning (multiple ways to reach the same conclusion) and the retraction of facts or conclusions, given new information. It is useful for processing unreliable knowledge and data.

Ordered Tree: An rooted tree in which the children of each vertex are assigned an order.

Parse: To break a sentence down into its component parts in order to decipher it. The process of splitting a text or expression into pieces to determine its syntax.

Predicate Calculus: A formal language of classical logic that uses functions and predicates to describe relations between individual entities.

Predictive Recursive Descent Parser: a top down method of syntax analysis in which a set of recursive procedures process input. A predictive parser is one that avoids backtracking by using a lookahead function to identify symbols defined by a special Follow function.

Production Rule: The type of rule used in a production system to operate on a global database. A production rule consists of two parts, i.e., a precondition (antecedent, premise or condition) part, and an action (conclusion) part.

Production System: An underlying type of AI system having a control system that uses productions to operate on a global data base.

Prune: To reduce or narrow the alternatives, normally used in the context of reducing possibilities in a branching tree structure, such as the search through a problem space.

Recursion: A procedure that calls itself, either directly or indirectly.

Rooted Tree: A free tree having one particular vertex singled out as the root.

Search: The process of looking through the set of possible solutions to a problem in order to find an acceptable solution.

Search Space: The set of all possible solutions to a problem.

Semantics: The specification of the meaning of sentences in a language.

Space Complexity: A measure of how much storage space is required by a program as the size of the input changes.

Syntax: The specification of the proper construction of sentences from words for a language.

Tail Recursion: A procedure in which the last action is to make a recursive call to itself. This type of recursion is undesirable since upon resumption of the recursive procedure the procedure itself terminates, after having wasted computer resources. Tail recursion is easy to remove.

Time Complexity: A measure of how a program's execution time is affected as the size of the input changes.

Tool: A shorthand notation for expert system building tool or shell.

Tree Or Free Tree: Any set of points (called vertices) and any set of pairs of distinct vertices (called edges or branches) such that (1) there is a sequence of edges (a path) from any vertex to any other vertex, but, traversable in only one direction, and (2) there are no circuits, that is, no paths starting from a vertex to and returning to the same vertex.

REFERENCES CITED

1. Altman, Laurence, Publisher's Letter, Electronics, Vol. 59, No. 28, August 7, 1986, p. 3

2. Rose, Frederick, Thinking Machine, The Wall Street Journal, Vol. 119, No. 30, August 12, 1988, p. 1, p. 11

3. Carlyle, Ralph Emmett, Midrange Shootout: Mini/Micro Survey, Datamation, November 15, 1987, Vol. 33, No. 22, p. 60

4. Schussel, George, Application Development In The 5th Generation, Datamation, November 15, 1987, Vol. 33, No. 22, pp. 94-100

5. Hill, Thomas, Expert System Shells May Be Key To Artificial Intelligence, PC Week, Vol. 4, No. 30, July 28, 1987, pp. 47-54.

6. Taylor, W. A., Artificial Intelligence: Potentials And Limitations, Design News, Vol. 42, No. 5, March 3, 1986, pp. 75-82

7. Kidd, Alison L., Knowledge Acquisition For Expert Systems, A Practical Handbook, Plenum Press, 1987, p. 5

8. Firebaugh, Morris W., Artificial Intelligence, A Knowledge Based Approach, Boyd & Fraser, 1988, p. 401

9. Potter, Andrew, Direct Manipulation Interfaces, AI Expert, October 1988, Vol. 3, No. 10, pp. 28-35.

10. Schwartz, Tom, PC Perspectives, Evolution Comes To AI Artifacts, IEEE Expert, Fall 1987, Vol. 2, No. 3, pp. 80-85.

11. Jacobson, Alexander D., Lisp Is Not Needed For Expert Systems, Electronics, Vol. 59, No. 28, August 7, 1986, p. 64

12. Turpin, Bill, Artificial Intelligence: Project Needs, Design News, Vol. 42, No. 5, March 3, 1986, pp. 75-82

13. Firebaugh, p. 415

14. Davis, B. Dwight, Artificial Intelligence Enters The Mainstream, High Technology, Vol. 6, No. 7, July 1985, pp. 16-21

15. Manuel, Tom, The Pell-Mell Rush Into Expert Systems Forces Integration Issue, Electronics, Vol. 58, No. 26, July 1, 1985, pp. 54-59

16. Freedman, David H., Expert Systems Vendors Aim For Simpler, Lower-Cost Packages, High-Technology, Vol. 7, No. 4, April 1987, p. 26

17. Ten Dyke, Richard P., Artificial Intelligence: Integrating Expert Systems, Design News, Vol. 42, No. 5, March 1986, pp. 131-133

18. Schussel, pp. 94-100

19. Hill, pp. 47-54

20. Hayes-Roth, Frederick, Waterman, Donald A., and Lenat, Douglas B., Building Expert Systems, Reading, MA: Addison-Wesley, 1983, p. 6

21. Feigenbaum, Edward A., and McCorduck, Pamela, The Fifth Generation, Artificial Intelligence And Japan's Computer Challenge To The World, Reading, MA: Addison-Wesley, 1983

22. Japan threatens America's final stronghold, author unknown, San Jose Mercury News, April 23, 1986, p. 1a & p. 12a

23. Hayes-Roth, p. 7

24. Waterman, Donald A., A Guide To Expert Systems, Addison-Wesley, 1986, p. 214

25. Buchanan, Bruce G., and Shortliffe, Edward H., Rule Based Expert Systems, Addison-Wesley Publishing Company, 1985, p. 8

26. Forsyth, Richard, Ed, Expert Systems, Principles And Case Studies, Chapman and Hall, 1984, p. 6

27. Firebaugh, p. 375

28. Buchanan, p. 108

29. Waterman, p. 204

30. Firebaugh, p. 356

31. Waterman, p. 185

32. van de Brug, Arnold, Bachant, Judith, and McDermott, John, The Taming Of R1, IEEE Expert, Vol. 1 no. 3, Fall 1986, pp. 33-39.

33. Rich, Elaine, Artificial Intelligence, McGraw-Hill, 1983, p. 287

34. Kidd, p. 6

35. Nilsson, Nils J., Principles Of Artificial Intelligence, Tioga Publishing Company, 1980.

36. Hayes-Roth, p. 60, 65, 66, 69, 94, 293, 377

37. Sharit, Joseph, and Salvendy, Gavriel, A Real-Time Interactive Model of a Flexible Manufacturing System, IIE Transactions, June 1987, Vol. 19, No. 2, pp. 167-177

38. Charniak, Eugene, and McDermott, Drew, Introduction To Artificial Intelligence, Addison-Wesley, 1985, p.

39. Firebaugh, p. 20, 91, 96, 187

40. Rich, p. 20, 49, 78, 82, 91, 254, 267, 271, 277

41. Kidd, p. 15

42. Pearl, Judea, Heuristics, Intelligent Search Strategies For Computer Problem Solving, Addison-Wesley, 1984.

43. Addis, T. R., Designing Knowledge-Based Systems, Prentice-Hall, 1987.

44. Buchanan, p. 304

45. Nilsson, p. 17

46. Nilsson, p. 18

47. Nilsson, pp. 193-267

48. Townsend, Carl, and Feucht, Dennis, Designing And Programming Personal Expert Systems, Tab Books, 1986

49. Grigonis, Richard W., Mycin-Like Expert Systems, Dr. Dobb's Journal, April 1987, Vol. 112, No. 4, pp. 42-82

50. Thompson, A. Beverly, and Thompson, A. William, Inside An Expert System, From Index Cards To Pascal Program, Byte, April 1985, Vol. 10, No. 4, pp. 315-329

51. Morgeson, Darrell, Artificial Intelligence Using Modula-2, A Simple Expert System, Journal Of Pascal, Ada & Modula-2, January/February 1985, pp. 29-38

52. Charniak, p. 440

53. Gabriel, Richard P., Lisp Expert Systems Are More Useful, Electronics, Vol. 59, No. 28, August 7, 1986, p. 65

54. Texas Instruments, An AI Productivity Roundtable; The Third Artificial Intelligence Satellite Symposium, televised world-wide on April 8, 1987

55. A New Way To Move AI Into The Mainstream, unsigned article, Electronics, Vol. 60, No. 2, January 22, 1987, pp. 90-92

56. Davis, pp. 16-22

57. Davis, B. Dwight, Artificial Intelligence Goes To Work, High Technology, Vol. 7, No. 4, April 1987, pp. 16-27

58. Leinwerber, David, Knowledge-Based Systems For Financial Applications, IEEE Expert, Fall 1988, Vol. 3, No. 3, pp. 18-31

59. Kalb, Bill, Artificial Intelligence Moves Into Reality, With a return of investment of 10:1, AI systems are exploding into real-time manufacturing applications, Chilton's Automotive Industries, Vol. 166, November, 1986, pp. 96-98

60. Davis, pp. 16-22

61. Davis, pp. 16-22

62. Kalb, pp. 96-98

63. Davis, pp. 16-22

64. Wolfe, Alexander, TI Puts Its LISP Chip Into A System For Military AI, Electronics, Vol. 60, No. 6, March 19, 1987, pp. 95-96

65. Brown, A. Windsor, Ed., Expert Products, Ti weds Explorer to Macintosh II, IEEE Expert, Vol. 3, No. 2, Summer 1988, p. 77.

66. Bruno, Giorgio, Elia, Antonio, and Laface, Pietro, A Rule-Based System to Schedule Production, Computer, Vol. 19, No. 7, July 1986, pp. 32-39

67. Paxton, C. Mark, Integrating Pascal And Scheme, AI Expert, October, 1988, Vol. 3, No. 10, pp. 55-61

68. Dietz, P. W., Artificial Intelligence: Building Rule-Based Expert Systems, Design News, Vol. 42, No. 5, March 3, 1986, pp. 137-144

69. Livingston, Dennis, Expert Systems And AI Hardware Reach Commercial Markets, High Technology, Vol. 6, No. 7, July 1986, p. 22

70. Manuel, Tom, What's Holding Back Expert Systems?, Electronics, Vol. 59, No. 28, August 7 , 1986, pp. 59-63.

71. Inference's Strategy To Speed Things Up, unsigned article, Electronics, Vol. 59, No. 28, pp. 66-69

72. Manuel, Tom, Update: Major Glitches Hit Non-Lisp Ai Tool, Electronics, Vol. 60, No. 17, August 20, 1987, p. 87.

73. Davis, pp. 16-22

74. Eliot, Lance B., Expert Focus, The Expert System Business, IEEE Expert, Vol. 3, No. 3, Fall 1988, pp. 5-6.

75. Stambler, Irwin, Researchers build flexibility into Lockheed expert system, Research & Development, Vol. 28, No. 10, October 1986, pp. 44-45

76. Babcock, Charles, Cobol-based AI shell bows, Tool lets expert systems run with mainframe data, Computerworld, Vol. 20, No. 35, September 1, 1986, p. 1 & p. 12

77. A New Way To Move AI Into The Mainstream, pp. 90-92

78. Butler, C. W., Hodil, E. D., and Richardson, G. L., Building Knowledge-Based Systems with Procedural Languages, IEEE Expert, Vol. 3, No. 2, Summer 1988, pp. 47-59.

79. Manuel, What's Holding Expert Systems Back?, pp. 59-63

80. Inference's Strategy To Speed Things Up, pp. 66-69

81. Manuel, The Pell-Mell Rush Into Expert Systems Forces Integration Issue, pp. 54-59

82. Ten Dyke, pp. 131-133

83. Schussel, pp. 94-100

84. Jacobson, p. 64

85. Perrone, Giovanni, Revised C Compiler Beats Competition In Speed, Features and Microsoft's Quick-C Is State Of The Art C Language Programming Environment, PC Week, December 1, 1987, pp. 107-119

86. Langa, Fred, Editorial, Touching All The Bases, Byte, August 1988, Vol. 13, No. 8, p. 6

87. Howard III, Harvey L., and Simon, Michael A., Microsoft Venerable C Compiler Gets Even Better, IEEE Software, September 1987, pp. 92-93

88. Schatz, Willie, Opponents Of Fortran 8X Are Facing An Uphill Battle, Datamation, October 15, 1987, Vol. 33, No. 20, pp. 22-28

89. Hayes-Roth, p. 302

90. Firebaugh, p. 309

91. Charniak, p. 438

92. Charniak, p. 438

93. Firebaugh, pp. 336-309

94. Nilsson, p. 195

95. Forsyth, p. 96

96. Charniak, p. 438

97. Firebaugh, p. 309

98. Nilsson, p. 195

99. Forsyth, p. 96

100. Nagy, Tom, Gault, Dick, and Nagy, Monica, Building Your First Expert System, Ashton-Tate Publishing Group, 1985

101. Laird, John E., Soar's User's Manual, Xerox Corporation, Palo Alto Research Center, 1986

102. Davis, Randall, Knowledge Based Systems, Science, Vol. 231, February 28, 1986, pp. 957-963

103. Firebaugh, p. 309

104. Charniak, p. 438

105. Nilsson, p. 89

106. Nilsson, pp. 53-112

107. Rich, pp. 72-94

108. Pearl, pp. 33-69

109. Townsend, pp. 34-64

110. Levine, Robert I., Drang, Diane E., and Edelson, Barry, A Comprehensive Guide To AI And Expert Systems, McGraw-Hill Book Company, 1986

111. Thompson, pp. 315-329

112. Morgeson, pp. 29-38

113. Grigonis, p. 42

114. Buchanan, p. 82

115. Buchanan, p. 102

116. Kruse, Robert L., Data Structures And Program Design, Prentice-Hall, Inc., 1987, p. 318

117. Kruse, p. 318

118. Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey, Compilers, Principles, Techniques, And Tools, Addison Wesley Publishing Company, 1986, p. 181

119. Aho, p. 82

120. Waite, William M., and Goos, Gerhard, Compiler Construction, Springer-Verlag New York Inc., 1984, p. 161

121. Tremblay, Jean-Paul, and Sorenson, Paul G., An Implementation Guide To Compiler Writing, McGraw-Hill, Inc., 1982, pp. 27-99

122. Pearl, pp. 36-43

123. Boyd, Donald W., Integrating Artificial Intelligence Into Planning Model Development, Unpublished Research Proposal, Department Of Industrial And Management Engineering, Montana State University, Bozeman, Montana, September 16, 1987

124. Boyd, Donald W., River Basin Modeling Via Systems Analysis And Artificial Intelligence, Hydrosoft, April 1988, Vol. 1, No. 2, pp. 80-87

APPENDICES

APPENDIX A

Data Structure Definitions: Databases

CROSS REFERENCE

Page

Figure 5. NODES.H1: Data Structures Header File.

```
/*  NODES.H1
 *
 *  CREATED 02-23-88, BEN GROENEVELD
 *  IHE 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  DEFINITIONS FOR MAXIMUM (X_s) STRING LENGTHS (_LENs), ARRAY LENGTHS
 *  (_LINs), BOOLEANS, RETURN STATUS VALUES, ALL NODAL STRUCTURES (_Ns, TREES
 *  AND LISTS), AND MEMORY ALLOCATION MACRO FOR CHARACTER STRINGS.
 */

#define ALLOC      999    /* ERROR CODE FOR INSUFFICIENT DYNAMIC MEMORY */

                          /* EACH LEN UNIQUE BECAUSE OF USE IN SWITCHES */

#define ATTR_LEN   313    /* 3 * 78, ACTUAL LEN = LEN - 1 = 312 */
#define ERR_LEN    511    /* MAX + ".....***%s***", I.E., ERROR TEXT  */
#define LIN_LEN     79    /* ACTUAL LEN = LEN - 1 = 78 */
#define OPER_LEN    15    /* INCLUDES SPACE FOR CODE */
#define TRANS_LEN  312
#define X_LEN      512    /* MAX ALLOWABLE BY MS-C COMPILER IS 512 */

#define EPILOG_LIN  22    /* ACTUAL LINES = LIN - 1 = 21; END NULL SENTINEL */
#define INTRO_LIN   22    /* ACTUAL LINES = LIN - 1 = 21 */
#define PROMPT_LIN  20    /* AT LEAST 1 LINE FOR ANS */
#define RULE_LIN    17    /* AT LEAST 1 LINE FOR ANS AND TWO TITLES */

#define OCCUPIED  3       /* STACKS */
#define EMPTY     2       /* STACKS */
#define FAIL      1       /* RULES */
#define OK        0       /* RULES AND FUNCTION SUCCESSFUL RETURN CODE */
#define UNKNOWN  -1       /* RULES */
#define EXTERNAL -2       /* RULES: ATTRIBUTE VALUE OBTAINED EXTERNALLY */

#define C_UNKN    "_Unknown"   /* DEFAULT UNKNOWN ATTR VAL INSERTED INTO WM */
#define C_NO_Q    "_Value of " /* DEFAULT RULE PROMPT */
#define NO_Q_LEN  10           /* LENGTH OF DEFAULT RULE PROMPT */

#define RESET_WM  "_rswm"
#define DO_RESET  "do_reset"

#define TRUE      1
#define FALSE     0

/*  MACRO TO ALLOCATE MEMORY FOR A CHARACTER STRING. */

#define MEM(str,len) ((char *) malloc ((strlen (str) + len) * sizeof (char)))
```

Figure 5 (continued).

```
/*  MACRO TO REMOVE OPERATOR CODE VALUE FROM STORED OPERATOR STRING */

#define KEY_LEN 9 /*  012345678  = KEY LEN OF *keywords ELEMENTS, parse.c */
                  /* "valid_val" = KEY LEN OF *keywords ELEMENTS, parse.c */
#define ENCODE(str) (((str)[KEY_LEN] - 48) * 10 + (str)[KEY_LEN + 1] - 48)

/*  CONTEXT BINARY TREE NODAL STRUCTURE */

struct context_n {char            *key;
                  char            *intro[INTRO_LIN];
                  char            *epilog[EPILOG_LIN];
                  int             exist;              /* DEL CONT FLAG */
                  struct attr_n   *attr_rt;           /* ATTR SUB TREE */
                  struct context_n *left;
                  struct context_n *right;};

/*  ATTRIBUTE BINARY TREE NODAL STRUCTURE */

struct attr_n {char          *key;            /* CONCLUSION ATTR */
               char          *prompt[PROMPT_LIN];
               char          *trans;
               char          oper[OPER_LEN];  /* OPERATOR STRING AND CODE */
               struct list_n *valid_val;      /* LINEAR ATTR VAL SUB-LIST */
               struct val_n  *top_branch;     /* LINEAR RULE SUB-LIST */
               struct attr_n *left;
               struct attr_n *right;};

/*  RULE BRANCH LINEAR LIST NODAL STRUCTURE, I.E., "OR" NODES AND
 *  CONNECTORS TO HEADS OF "AND" LISTS
 */

struct val_n {char          *val;          /* CONCLUSION ATTR VALUE */
              int           rule;          /* RULE: TRUE, FALSE, UNKNOWN */
              int           rule_num;      /* RULE NUMBER, PARSE ORDER */
              struct cond_n *p_cond;       /* CONDITIONS LINEAR SUB-LIST */
              struct val_n  *next;};       /* OR CONDITIONS SAME ATTR/RULE */

/*  RULE CONDITION CLAUSES FOR A SINGLE CONCLUSION ATTRIBUTE VALUE LINEAR
 *  LIST NODAL STRUCTURE, I.E., "AND" NODES IN RULE BRANCH.
 */

struct cond_n {char          *attr;          /* CONDITION ATTR */
               char          oper[OPER_LEN]; /* OPERATOR STRING AND CODE */
               char          *val;           /* CONDITION ATTR VAL */
               struct cond_n *next;};        /* AND CONDITIONS SAME RULE */
```

Figure 5 (continued).

```
/*  WORKING MEMORY BINARY TREE NODAL STRUCTURE */

struct wm_n {char            *attr;
             char            *val;
             int             rule_num;
             char            *goal;        /* CONTEXT WHICH PRODUCED FACT */
             struct cond_n   *trig_nodes;  /* CONDITION CLAUSE    "       */
             struct wm_n     *left;
             struct wm_n     *right;};

/*  LINEAR LIST NODAL STRUCTURE: GOAL STACKS AND LISTS (I.E., *valid_val) */

struct list_n {char         *elem;
               struct val_n  *from;
               struct list_n *next;};

/*  BREADTH FIRST SEARCH SPECIALIZED QUEUE NODAL STRUCTURE */

struct q_n {char          *elem;
            struct list_n *trace_s;   /* STACK OF GOALS (POINTERS) TO ROOT */
            struct q_n    *next;};

typedef struct context_n CONT_N;
typedef struct attr_n    ATTR_N;
typedef struct val_n     VAL_N;
typedef struct cond_n    COND_N;
typedef struct wm_n      WM_N;
typedef struct list_n    LIST_N;
typedef struct q_n       Q_N;

/* CODES FOR ALL OPERATORS, SAME CODE FOUND IN *oper NODAL ELEMENTS */

#define AND        10
#define EQ         11
#define GE         12
#define GOAL       13
#define GT         14
#define IDENTIFIER 35   /* BECAUSE OPER_LEN IS 35, PRODUCES DUP CASE */
#define IF         16
#define IS         17
#define LE         18
#define LT         19
#define NE         20
#define NOT        21
#define OR         22
#define PROMPT     23
#define INTRO      24
#define THEN       25
#define TRANS      26
```

Figure 5 (continued).

```
#define EPILOG      27
#define VALID_VAL   28
#define TXT_VAL     30
```

APPENDIX B

Sample Low-Level Database Maintenance Modules

CROSS REFERENCE

Page

Figure 6. Q_KB.H1: Query Knowledge Base Header File.

```
/*  Q_KB.H1
 *
 *  CREATED 07-19-88, BEN GROENEVELD
 *  IHE 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  FUNCTION PROTOTYPES.
 */

extern ATTR_N *a_tr_srch (ATTR_N *rt, char *targ);
extern CONT_N *c_tr_srch (CONT_N *rt, char *targ);
```

Figure 7. Q_KB.C: Query Knowledge Base Source File.

```
/*  Q_KB.C - LOW-LEVEL EST MODULE (KNOWLEDGE BASE MAINTENANCE)
 *  Q_KB.HI HEADER FILE HAS LIST OF THIS MODULE'S GLOBAL FUNCTIONS.
 *
 *  CREATED 02-29-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  QUERY KNOWLEDGE BASE BINARY TREE ROUTINES.
 *
 *  ENDING NODES FOR ALL TREE ROUTINES MUST BE PROPERLY INITIALIZED TO NULL,
 *  INCLUDING THE ROOTS AND ANY STRUCTURE ELEMENTS.
 *
 *  NOTE: ALL PREPROCESSOR DIRECTIVES (INCLUDING MACROS) CAPITALIZED.
 *        ALL GLOBAL VARIABLES BEGIN WITH A CAPITAL.
 */

#define LINT_ARGS          /* enforce lib function type checking */

#include <math.h>          /* atof */
#include <stdio.h>         /* NULL, sprintf */
#include <stdlib.h>        /* free, malloc, MEM */
#include <string.h>        /* strcmp, strcpy, strlen */

#include <nodes.hi>        /* EST DEFINITIONS */

/*  PROGRAM GLOBAL DECLARATIONS (VISIBLE IN ALL MODULES).
 */

#include <q_kb.hi>         /* EST MODULE */

/*  MODULE (SOURCE FILE) GLOBAL DECLARATIONS.
 */

/*  SEARCH ATTRIBUTE TREE FOR ATTRIBUTE.
 *  RETURN VALUE: NODE, NULL.
 */

ATTR_N *a_tr_srch (ATTR_N *rt, char *targ)

{  int result;

   while (rt != NULL)
   {  if ((result = strcmp (rt -> key, targ)) == OK) return (rt);

      if (result > 0)
        rt = rt -> left;
      else
        rt = rt -> right;
   }
```

Figure 7 (continued).

```
   return (rt);
}

/*  SEARCH CONTEXT TREE FOR CONTEXT.
 *  RETURN VALUE: NODE, NULL.
 */

CONT_N *c_tr_srch (CONT_N *rt, char *targ)

{  int result;

   while (rt != NULL)
   {  if ((result = strcmp (rt -> key, targ)) == OK) return (rt);

      if (result > 0)
        rt = rt -> left;
      else
        rt = rt -> right;
   }

   return (rt);
}
```

Figure 8. M_WM.H1: Manipulate Working Memory Header File.

```
/*  H_WM.H1
 *  CREATED 07-19-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *  FUNCTION PROTOTYPES, ERROR NUMBERS.
 */

extern int  del_wm_tr (WM_N **rt),
            ins_wm_tr (WM_N **rt, char *attr, char *val, int rule_num,
              COND_N *trig_nodes, char *cont_goal, ATTR_N *attr_rt,
              char *err);

#define M_NON_MONO 301
#define M_MATH     302
```

Figure 9. M_WM.C: Manipulate Working Memory Source File.

```
/*  M_WM.C - LOW-LEVEL EST MODULE (WORKING MEMORY MAINTENANCE)
 *  M_WM.H1 HEADER FILE HAS LIST OF THIS MODULE'S GLOBAL FUNCTIONS.
 *
 *  CREATED 02-29-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  MANIPULATE WORKING MEMORY BINARY TREE ROUTINES.
 *
 *  ENDING NODES FOR ALL TREE ROUTINES MUST BE PROPERLY INITIALIZED TO NULL,
 *  INCLUDING THE ROOTS AND ANY STRUCTURE ELEMENTS.
 *
 *  GLOBAL FLAGS:  MONOTONIC REASONING ON/OFF
 *
 *  NOTE: ALL PREPROCESSOR DIRECTIVES (INCLUDING MACROS) CAPITALIZED.
 *        ALL GLOBAL VARIABLES BEGIN WITH A CAPITAL.
 */

#define LINT_ARGS          /* enforce lib function type checking */

#include <math.h>          /* atof */
#include <stdio.h>         /* NULL, sprintf */
#include <stdlib.h>        /* free, malloc, MEM */
#include <string.h>        /* strcmp, strcpy, strlen */

#include <nodes.h1>        /* EST DEFINITIONS */
#include <q_kb.h1>         /* EST MODULE */
#include <q_wm.h1>         /* EST MODULE */

/*  PROGRAM GLOBAL DECLARATIONS (VISIBLE IN ALL MODULES).
 */

#include <m_wm.h1>         /* EST MODULE */

extern int Mono_flag;

/*  MODULE (SOURCE FILE) GLOBAL DECLARATIONS.
 */

#define A_WID 10   /* MAX WID FOR DOUBLE CONVERTED TO ASCII */

static int  do_math (WM_N *wm, char *val, char *err),
            dtoa (char *ascii, double real_val),
            man_err (int err_num, char *err_str, char *err);
static WM_N *mk_wm_n (char *attr, char *val, int rule_num, COND_N *trig_nodes,
              char *cont_goal);
```

Figure 9 (continued).

```
/*  RECURSIVELY TRAVERSE WM TREE IN ORDER TO DELETE AND FREE IT.
 *  RETURN VALUE: OK.
 */

int del_wm_tr (WM_N **rt)

{  if (*rt != NULL)
   {  del_wm_tr (&((*rt) -> left));
      del_wm_tr (&((*rt) -> right));
      free ((*rt) -> attr);
      free ((*rt) -> val);
      free ((*rt) -> goal);
      free (*rt);
      *rt = NULL;
   }

   return (OK);
}

/*  INSERT AN INFERRED ATTRIBUTE AND ITS VALUES INTO WORKING MEMORY TREE.
 *  RETURN VALUE: OK, ERROR CODE.
 */

int ins_wm_tr (WM_N **rt, char *attr, char *val, int rule_num, COND_N
      *trig_nodes, char *goal, ATTR_N *attr_rt, char *err)

{  ATTR_N *curr_attr = NULL;
   char   val_cpy[ATTR_LEN];
   int    result, code;
   WM_N   *p = *rt;

   strcpy (val_cpy, val);
   curr_attr = a_tr_srch (attr_rt, attr);
   if (ENCODE (curr_attr -> oper) == EQ)
      if ((code = do_math (*rt, val_cpy, err)) != OK) return (code);

   if (*rt == NULL)
   {  if ((*rt = mk_wm_n (attr, val_cpy, rule_num, trig_nodes, goal)) != NULL)
        return (OK);
      return (man_err (ALLOC, attr, err));
   }

   while (p != NULL)
   {
      if ((result = strcmp (attr, p -> attr)) < 0)
      {
        if (p -> left == NULL)
        {  if ((p -> left = mk_wm_n (attr, val_cpy, rule_num, trig_nodes,
              goal)) == NULL) return (man_err (ALLOC, attr, err));
```

Figure 9 (continued).

```
            p = NULL;
          }
          else
            p = p -> left;
        }
        else if (result > 0)
        {
          if (p -> right == NULL)
          {  if ((p -> right = mk_wm_n (attr, val_cpy, rule_num, trig_nodes,
                goal)) == NULL) return (man_err (ALLOC, attr, err));
            p = NULL;
          }
          else
            p = p -> right;
        }
        else   /* CHECK FOR NON-MONOTONIC REASONING */
        {
          if (strcmp (val_cpy, p -> val) != OK && Mono_flag)
            return (man_err (M_NON_MONO, attr, err));
          else
          {  strcpy (p -> val, val_cpy);
            strcpy (p -> goal, goal);
            p -> rule_num = rule_num;
            p -> trig_nodes = trig_nodes;
          }

          return (OCCUPIED);
        }
      }

      return (OK);
}

/*  PERFORM MATHEMATICAL OPERATIONS IN VALUE TO CALCULATE IT AND RETURN IT.
 *  RETURN VALUE: OK, FAIL.
 */

int do_math (WM_N *wm, char *val, char *err)

{  char   *asc_operand, *operator, cat_str[ATTR_LEN];
   double result, operand;
   WM_N   *wm_n = NULL;

   strcpy (cat_str, val);
   if ((asc_operand = strtok (cat_str, " ")) == NULL)
      return (man_err (M_MATH, val, err));
   if ((operator = strtok (NULL, " ")) == NULL) return (OK); /* CONDITION */
   if ((wm_n = in_wm_tr (wm, asc_operand)) == NULL)
      return (man_err (M_MATH, val, err));
```

Figure 9 (continued).

```
    result = atof (wm_n -> val);

    do
    {  if ((asc_operand = strtok (NULL, " ")) == NULL)
         return (man_err (M_MATH, val, err));
       if ((wm_n = in_wm_tr (wm, asc_operand)) == NULL)
         return (man_err (M_MATH, val, err));
       operand = atof (wm_n -> val);

       switch (*operator)
       {  case '+':
            result += operand;
            break;
          case '*':
            result *= operand;
            break;
          case '-':
            result -= operand;
            break;
          case '/':
            result /= operand;
            break;
          default:
            return (man_err (M_MATH, val, err));
       }
    }
    while ((operator = strtok (NULL, " ")) != NULL);

    dtoa (val, result);
    return (OK);
}

/*  CONVERT DOUBLE VALUE TO ASCII, INSERT SIGN AND DECIMAL FOR IN DISPLAY.
 *  RETURN VALUE: OK.
 */

int dtoa (char *str, double real_val)

{  char ascii[A_MID + 1], a_str[A_MID + 1], *p;
   int  sign, dec_loc, index = 0;

   strcpy (ascii, ecvt (real_val, A_MID - 1, &dec_loc, &sign));
   if (sign) a_str[index++] = '-';

   if (dec_loc > 0)   /* DOUBLE IS LARGER THAN 1 */
   {  for (p = ascii; p < ascii + dec_loc && index < A_MID; p++)
        a_str[index++] = *p;
      a_str[index++] = '.';
   }
```

Figure 9 (continued).

```
   else   /* DOUBLE < 1; DECIMAL IS TO THE LEFT OF FIRST DIGIT */
   {  a_str[index++] = '.';
      while (dec_loc++ < 0 && index < A_MID) a_str[index++] = '0';
      p = ascii;
   }

   while (*p != '\0' && index < A_MID) a_str[index++] = *(p++);
   a_str[index] = '\0';
   for (p = a_str + index - 1; *p == '0' && *p != '.'; p--)
      a_str[--index] = '\0';   /* DEL TRAILING 0s */

   if (strcmp (a_str, ".") == OK)
      strcpy (str, "0");
   else
   {  if (*p == '.') *p = '\0';  /* MAKE IT AN INTEGER */
      strcpy (str, a_str);
   }

   return (OK);
}

/*  ALLOCATE WORKING MEMORY NODE FOR USE IN WORKING MEMORY TREE.
 *  RETURN VALUE: NODE, NULL.
 */

WM_N *mk_wm_n (char *attr, char *val, int rule_num, COND_N *trig_nodes,
       char *goal)

{  WM_N *new_n = NULL;

   if ((new_n = (WM_N *) malloc (sizeof (WM_N))) == NULL) return (NULL);
   new_n -> left = new_n -> right = NULL;
   if ((new_n -> attr = MEM (attr, 1)) == NULL) return (NULL);
   strcpy (new_n -> attr, attr);
   if ((new_n -> val = MEM (val, 1)) == NULL) return (NULL);
   strcpy (new_n -> val, val);
   new_n -> rule_num = rule_num;
   if ((new_n -> goal = MEM (goal, 1)) == NULL) return (NULL);
   strcpy (new_n -> goal, goal);
   new_n -> trig_nodes = trig_nodes;
   return (new_n);
}

/*  PUT ERROR MESSAGE CORRESPONDING TO CODE INTO err.
 *  RETURN VALUE: ERROR CODE.
 */
```

Figure 9 (continued).

```
int man_err (int err_num, char *err_str, char *err)

{  switch (err_num)
   {  case ALLOC:
        sprintf (err, "Hanip wm err %d; insufficient memory for dynamic "
          "allocation:  ***%s***", err_num, err_str);
        break;
      case H_NON_MONO:
        sprintf (err, "Hanip wm err %d; non-monotonic reasoning invalid "
          "in this configuration:  ***%s***", err_num, err_str);
        break;
      case H_MATH:
        sprintf (err, "Hanip wm err %d; unable to perform mathematical "
          "operations specified in conclusion attribute value:  ***%s***",
          err_num, err_str);
        break;
   }
   return (err_num);
}
```

Figure 10. Q_WM.H1: Query Working Memory Header File.

```
/*  Q_WM.H1
 *  CREATED 07-19-88, BEN GROENEVELD
 *  IHE 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *  FUNCTION PROTOTYPES.
 */

extern int  wm_tr_srch (WM_N *rt, COND_N *targ_n, int oper_code);
extern WM_N *in_wm_tr (WM_N *rt, char *targ);
```

Figure 11. Q_WM.C: Query Working Memory Source File.

```
/*  Q_WM.C - LOW-LEVEL EST MODULE (WORKING MEMORY MAINTENANCE)
 *  Q_WM.HI HEADER FILE HAS LIST OF THIS MODULE'S GLOBAL FUNCTIONS.
 *
 *  CREATED 02-29-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  QUERY WORKING MEMORY BINARY TREE ROUTINES.
 *
 *  ENDING NODES FOR ALL TREE ROUTINES MUST BE PROPERLY INITIALIZED TO NULL,
 *  INCLUDING THE ROOTS AND ANY STRUCTURE ELEMENTS.
 *
 *  NOTE: ALL PREPROCESSOR DIRECTIVES (INCLUDING MACROS) CAPITALIZED.
 *        ALL GLOBAL VARIABLES BEGIN WITH A CAPITAL.
 */

#define LINT_ARGS          /* enforce lib function type checking */

#include <math.h>          /* atof */
#include <stdio.h>         /* NULL */
#include <string.h>        /* strcmp */

#include <nodes.hi>        /* EST DEFINITIONS */

/*  PROGRAM GLOBAL DECLARATIONS (VISIBLE IN ALL MODULES).
 */

#include <q_wm.hi>         /* EST MODULE */

/*  MODULE (SOURCE FILE) GLOBAL DECLARATIONS.
 */

/*  SEARCH WORKING MEMORY TREE TO FIND NODE MATCHING BOTH ATTRIBUTE AND VALUE.
 *  RETURN VALUE: OK, FAIL.
 */

int wm_tr_srch (WM_N *rt, COND_N *targ_n, int oper)

{  int result;

   while (rt != NULL)
   {
      if ((result = strcmp (rt -> attr, targ_n -> attr)) == OK)
      {
        switch (oper)
        {  case IS:
             if (strcmp (rt -> val, targ_n -> val) == OK) return (OK);
             return (FAIL);
           case EQ:
             if (atof (rt -> val) == atof (targ_n -> val)) return (OK);
```

Figure 11 (continued).

```
            return (FAIL);
         case GE:
            if (atof (rt -> val) >= atof (targ_n -> val)) return (OK);
            return (FAIL);
         case GT:
            if (atof (rt -> val) >  atof (targ_n -> val)) return (OK);
            return (FAIL);
         case LE:
            if (atof (rt -> val) <= atof (targ_n -> val)) return (OK);
            return (FAIL);
         case LT:
            if (atof (rt -> val) <  atof (targ_n -> val)) return (OK);
            return (FAIL);
         case NE:
            if (atof (rt -> val) != atof (targ_n -> val)) return (OK);
            return (FAIL);
         case NOT:
            if (strcmp (rt -> val, targ_n -> val) != OK) return (OK);
            return (FAIL);
       }
     }

     if (result > 0)
       rt = rt -> left;
     else
       rt = rt -> right;
   }

   return (UNKNOWN);
}

/*  DO SIMPLE CHECK TO SEE IF ATTRIBUTE EXISTS IN WORKING MEMORY.
 *  RETURN VALUE: NODE, NULL.
 */

WM_N *in_wm_tr (WM_N *rt, char *targ)

{  int result;

   while (rt != NULL)
   {
     if ((result = strcmp (rt -> attr, targ)) == OK) return (rt);

     if (result > 0)
       rt = rt -> left;
     else
       rt = rt -> right;
   }
```

Figure 11 (continued).

```
    return (rt);
}
```

APPENDIX C

Sample High-Level Expert System Technology Modules

CROSS REFERENCE

Figure 12. DEPTH.H1: Backward Depth-First Control System Header File.

```
/*  DEPTH.H1
 *
 *  CREATED 07-19-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  FUNCTION PROTOTYPES, ERROR NUMBERS.
 *
 *  NOTE: rd_fn, rd_val, wr_intro and wr_epilog ARE USER WRITTEN ROUTINES.
 */

extern int depth (CONT_N **kb, WM_N **wm, char *goal, char *goal_val,
            char *err),
         rd_fn (LIST_N *valid_val, char *prompt[], int oper_code,
            char *goal, char *goal_val, LIST_N **goal_s, char *err),
         rd_val (LIST_N *valid_val, char *prompt[], int oper_code,
            char *goal, char *goal_val, char *err),
         wr_intro (char *intro[], char *err),
         wr_epilog (char *epilog[], char *cont_goal, char *val, char *err);

#define DEPTH_ERR  401
#define D_NOT_ATTR 402
#define D_NOT_CONT 403
```

Figure 13. DEPTH.C: Backward Depth-First Control System Source File.

```
/*  DEPTH.C - HIGH-LEVEL EST MODULE (CONTROL SYSTEM)
 *  DEPTH.HI HEADER FILE HAS LIST OF THIS MODULE'S GLOBAL FUNCTIONS.
 *
 *  CREATED 10-9-87, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  INFERENCE ENGINE: BACKWARD CHAINING AND DEPTH FIRST SEARCH.
 *
 *  FLAG SETTINGS:  - BACKTRACKING ON/OFF
 *                  - CASCADING CONTEXT INSTANTIATION ON/OFF
 *                  - EXPANSION OF RULES FIRST/LAST CONDITION ATTRIBUTE FIRST
 *
 *  NOTE: ALL PREPROCESSOR DIRECTIVES (INCLUDING MACROS) CAPITALIZED.
 *        ALL GLOBAL VARIABLES BEGIN WITH A CAPITAL.
 */

#define LINT_ARGS          /* enforce lib function type checking */

#include <stdio.h>         /* NULL */
#include <string.h>        /* strcpy, strcmp, strlen */

#include <nodes.hi>        /* EST DEFINITIONS */
#include <q_kb.hi>         /* EST MODULE */
#include <q_wm.hi>         /* EST MODULE */
#include <m_wm.hi>         /* EST MODULE */
#include <stack.hi>        /* EST MODULE */

/*  PROGRAM GLOBAL DECLARATIONS (VISIBLE IN ALL MODULES).
 *  REASONING FLAGS.
 */

#include <depth.hi>        /* EST MODULE */

extern int Track_flag, Casc_flag, Expand_first_cond;

/*  MODULE (SOURCE FILE) GLOBAL DECLARATIONS.
 */

static int depth_err (int err_num, char *err_str, char *err),
           eval_rule (WM_N *wm, COND_N *cond, char *new_goal);

/*  INFERENCE ENGINE: DEPTH FIRST SEARCH WITH BACKTRACKING.
 *  RETURN VALUE: OK, ERROR CODE.
 */
```

Figure 13 (continued).

```
int depth (CONT_N **kb, WM_N **wm, char *goal, char *goal_val, char *err)

{  ATTR_N *curr_attr = NULL;
   char   new_goal[ATTR_LEN], cont_goal[ATTR_LEN];
   CONT_N *cont = NULL;
   int    code, goal_found = FALSE;
   LIST_N *goal_s = NULL;
   VAL_N  *from = NULL;

   /*  FIND THE CONTEXT, DISPLAY INTRODUCTORY TITLE TEXT, AND FIND THE GOAL'S
    *  RULE BRANCH TOP (from).
    */

   strcpy (cont_goal, goal);
   if ((cont = c_tr_srch (*kb, goal)) == NULL)
      return (depth_err (D_NOT_CONT, goal, err));
   if ((code = wr_intro (cont -> intro, err)) != OK) return (code);
   if ((curr_attr = a_tr_srch (cont -> attr_rt, goal)) == NULL)
      return (depth_err (D_NOT_ATTR, goal, err));

   /*  DO META RULE OPERATION HERE (RE-ORDER RULE BRANCH). */

   from = curr_attr -> top_branch;

   /*  RUN DOWN RULE BRANCHES IN DEPTH FIRST, BACKTRACKING MANNER */

   do
   {
      switch (eval_rule (*wm, from -> p_cond, new_goal))
      {  case OK:
           from -> rule = TRUE;
           if ((code = ins_wm_tr (wm, goal, from -> val, from -> rule_num,
              from -> p_cond, cont_goal, cont -> attr_rt, err)) != OK &&
              code != OCCUPIED) return (code);
           if (pop (&goal_s, goal, &from) == EMPTY) goal_found = TRUE;
           break;
        case FAIL:
           from -> rule = FALSE;

           if ((from = from -> next) == NULL)   /* CANNOT DETERMINE ATTR */
           {  if ((code = ins_wm_tr (wm, goal, C_UNKN, UNKNOWN, NULL,
                 cont_goal, cont -> attr_rt, err)) != OK && code != OCCUPIED)
                 return (code);
              if (pop (&goal_s, goal, &from) == EMPTY)
                 return (depth_err (DEPTH_ERR, cont_goal, err));
           }

           break;
```

Figure 13 (continued).

```
case UNKNOWN:
   if (push (&goal_s, goal, from) != OK)
      return (depth_err (ALLOC, cont_goal, err));
   strcpy (goal, new_goal);
   if ((curr_attr = a_tr_srch (cont -> attr_rt, goal)) == NULL)
      return (depth_err (D_NOT_ATTR, goal, err));

   /*  PERFORM FUTURE META RULE OPERATION HERE, ONCE PER BRANCH. */

   if ((from = curr_attr -> top_branch) == NULL)
   {
     /*  CANNOT FIND VALUE OF GOAL WITH RULES, OTHER OPTIONS:
      *  1)  GET EXTERNALLY IF PROMPT EXISTS (PROMPT BACKTRACKING).
      */

      if (strncmp ((curr_attr -> prompt)[0], C_NO_Q, NO_Q_LEN) != OK)
      {
         if ((curr_attr -> prompt)[0][0] == '\0')
         /*  XFER CONTROL TO EXTERNAL FUNCTION */
         {  if ((code = rd_fn (curr_attr -> valid_val,
               curr_attr -> prompt, ENCODE (curr_attr -> oper), goal,
               goal_val, &goal_s, err)) != OK) return (code);
         }
         /*  ADD GOAL TO WM USING PROMPT FOR VALUE */
         else
         {  if ((code = rd_val (curr_attr -> valid_val,
               curr_attr -> prompt, ENCODE (curr_attr -> oper), goal,
               goal_val, err)) != OK) return (code);
         }

         if (strcmp (goal_val, RESET_WM) != OK)
           if ((code = ins_wm_tr (wm, goal, goal_val, EXTERNAL,
               NULL, cont_goal, cont -> attr_rt, err)) != OK && code
               != OCCUPIED) return (code);
      }

     /*  2)  GET BY INSTANTIATING OTHER CONTEXT IF IT'S CONTEXT GOAL.
      */

      else if (Casc_flag && strcmp (goal, cont_goal) != OK &&
         (code = depth (kb, wm, goal, goal_val, err)) != D_NOT_CONT)
      {  if (code != OK) return (code);
      }

     /*  3)  GET EXTERNALLY IF NO PROMPT EXISTS BUT LEAF BACKTRACKING.
      */
```

Figure 13 (continued).

```
            else if (!Track_flag)
            {
               if ((code = rd_val (curr_attr -> valid_val, curr_attr -> prompt,
                  ENCODE (curr_attr -> oper), goal, goal_val, err)) != OK)
                  return (code);

               if ((code = ins_wm_tr (wm, goal, goal_val, EXTERNAL, NULL,
                  cont_goal, cont -> attr_rt, err)) != OK && code !=
                  OCCUPIED) return (code);
            }

           /*   4)  THIS GOAL'S A DEAD END (THE PREVIOUS "OTHER" OPTIONS
            *       DIDN'T WORK): BACKTRACK TO PREVIOUS GOAL.
            */

            else
            {  if ((code = ins_wm_tr (wm, goal, C_UNKN, UNKNOWN, NULL,
                  cont_goal, cont -> attr_rt, err)) != OK && code !=
                  OCCUPIED) return (code);
            }

           /*  AFTER 1), 2), 3) OR 4), CONTINUE WITH NEXT GOAL.
            */

            if (pop (&goal_s, goal, &from) == EMPTY) return (depth_err
               (DEPTH_ERR, cont_goal, err));
         }
      }

   }
   while (!goal_found);

   strcpy (goal_val, in_wm_tr (*wm, cont_goal) -> val);
   if ((code = wr_epilog (cont -> epilog, cont_goal, goal_val, err)) != OK)
      return (code);
   return (OK);
}

/*  EVALUATE RULE: Ok.......CONDITION CLAUSES MATCH WORKING MEMORY;
 *                 Fail.....CONDITION CLAUSES CONTRADICT WM FACTS;
 *                 Unknown..CONDITION CLAUSE VALUES UNKNOWN.
 *  RETURN VALUE: OK, FAIL, UNKNOWN.
 */

int eval_rule (WM_N *wm, COND_N *q, char *new_goal)

{  int code, unknown_attr = FALSE, last_first = TRUE;
```

Figure 13 (continued).

```
    do
    {  if ((code = wm_tr_srch (wm, q, ENCODE (q -> oper))) == FAIL)
          return (FAIL);

       if (code == UNKNOWN && last_first)   /* EXPAND 1ST/LAST NODE */
       {  unknown_attr = TRUE;
          if (Expand_first_cond) last_first = FALSE;
          strcpy (new_goal, q -> attr);
       }
    }
    while ((q = q -> next) != NULL);

    if (unknown_attr) return (UNKNOWN);
    return (OK);
}

/*  PUT ERROR MESSAGE CORRESPONDING TO CODE INTO err.
 *  RETURN VALUE: ERROR CODE.
 */

int depth_err (int err_num, char *err_str, char *err)

{  switch (err_num)
   {  case ALLOC:
        sprintf (err, "Reason err %d; insufficient memory for dynamic "
           "allocation:  ***%s***", err_num, err_str);
        break;
      case D_NOT_ATTR:
      case D_NOT_CONT:
        sprintf (err, "Reason err %d; cannot find target in tree:  "
           "****%s****", err_num, err_str);
        break;
      case DEPTH_ERR:
        sprintf (err, "Reason err %d; productions and working memory "
           "cannot conclude value for:  ***%s****", err_num, err_str);
        break;
   }

   return (err_num);
}
```

Figure 14. STACK.H1: Stack Auxiliary Routines Header File.

```
/*  STACK.H1
 *
 *  CREATED 07-19-88, BEN GROENEVELD
 *  IHE 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  FUNCTION PROTOTYPES.
 */

extern int pop (LIST_N **stack, char *elem, VAL_N **from),
           push (LIST_N **stack, char *elem, VAL_N *from);
```

Figure 15. STACK.C: Stack Auxiliary Routines Source File.

```
/*  STACK.C - HIGH-LEVEL EST MODULE AUXILIARY FILE (CONTROL SYSTEM)
 *  STACK.HI HEADER FILE HAS LIST OF THIS MODULE'S GLOBAL FUNCTIONS.
 *
 *  CREATED 07-13-88, BEN GROENEVELD
 *  IME 590, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 *
 *  SPECIALIZED STACK MANIPULATION ROUTINES USED BY CONTROL SYSTEMS.
 *
 *  NOTE: ALL PREPROCESSOR DIRECTIVES (INCLUDING MACROS) CAPITALIZED.
 *        ALL GLOBAL VARIABLES BEGIN WITH A CAPITAL.
 */

#define LINT_ARGS          /* enforce lib function type checking */

#include <stdio.h>         /* NULL */
#include <stdlib.h>        /* free, malloc, MEM */
#include <string.h>        /* strcpy, strlen */

#include <nodes.hi>        /* EST DEFINITIONS */

/*  PROGRAM GLOBAL DECLARATIONS (VISIBLE IN ALL MODULES).
 */

#include <stack.hi>        /* EST MODULE */

/*  MODULE (SOURCE FILE) GLOBAL DECLARATIONS.
 */

/*  POP ELEMENT FROM TOP OF STACK.
 *  RETURN VALUE: EMPTY, OCCUPIED.
 */

int pop (LIST_N **stack, char *elem, VAL_N **from)

{  LIST_N *top_n;

   if ((top_n = *stack) == NULL) return (EMPTY);
   strcpy (elem, top_n -> elem);
   *from = top_n -> from;
   *stack = top_n -> next;
   free (top_n -> elem);
   free (top_n);
   return (OCCUPIED);
}

/*  PUSH ELEMENT ONTO TOP OF STACK.
 *  RETURN VALUE: OK, FAIL.
 */
```

Figure 15 (continued).

```
int push (LIST_N **stack, char *elem, VAL_N *from)

{  LIST_N *new_n;

   if ((new_n = (LIST_N *) malloc (sizeof (LIST_N))) == NULL) return (FAIL);
   if ((new_n -> elem = MEM (elem, 1)) == NULL) return (FAIL);
   strcpy (new_n -> elem, elem);
   new_n -> from = from;
   new_n -> next = *stack;
   *stack = new_n;
   return (OK);
}
```

APPENDIX D

501 Modeler Integration Implementation

CROSS REFERENCE

Figure 16. RD_FN (): Attribute-Routine Mapping Function.

```
/*  INTER-APPLICATION TRANSFER-OF-CONTROL ROUTINE.  USER ROUTINE CALLED FROM
 *  Depth.c MODULE TO OBTAIN VALUE OF AN ATTRIBUTE EXTERNALLY THROUGH A
 *  FUNCTION CALL.
 *  RETURN VALUE: OK.
 */

int rd_fn (LIST_N *valid_val, char *prompt[], int oper_code, char *goal,
      char *goal_val, LIST_N **goal_s, char *err)

{  if (strcmp (goal, "do_rd_mod") == OK)
   {  rd_mod (Rd_mod_t);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, "do_wr_mod") == OK)
   {  wr_mod (Wr_mod_t);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, DO_RESET) == OK)
   {  reset (goal_s);
      strcpy (goal_val, RESET_WH);
   }
   else if (strcmp (goal, "do_decl_vars") == OK)
   {  decl_vars (Decl_vars_t);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, "do_init_vars") == OK)
   {  init_vars (Init_vars_t);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, "do_sh_eq") == OK)
   {  if (sh_eq (Hod_eq_t, TRUE) == TRUE)
        strcpy (goal_val, "successful");
      else
        strcpy (goal_val, "refine");
   }
   else if (strcmp (goal, "do_sh_regr_eq") == OK)
   {  sh_regr_eq (Regr_t);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, "do_sh_cal") == OK)
   {  sh_eq (Cal_t, FALSE);
      strcpy (goal_val, "successful");
   }
   else if (strcmp (goal, "do_br") == OK)
   {  do_br ();
      strcpy (goal_val, "successful");
   }
```

Figure 16 (continued).

```
    else if (strcmp (goal, "do_simul") == OK)
    {  simul (atoi (in_wm_tr (Wm, "num_periods") -> val));
       strcpy (goal_val, "successful");
    }
    else if (strcmp (goal, "do_exo_vars") == OK)
    {  if (exo_vars ())
         strcpy (goal_val, "initialized");
       else
         strcpy (goal_val, "not initialized");
    }

    return (OK);
}
```

Figure 17. 501_KA.KB: 501 Modeler Knowledge Acquisition Knowledge Base.

```
/===  501_KA.KB
 ===
 ===  CREATED 09-23-88, BEN GROENEVELD
 ===  EST KNOWLEDGE BASE FILE.
 ===  IME590/MONTS 501 PROJECT, MONTANA STATE UNIVERSITY, BOZEMAN, MONTANA
 ===
 ===  PROTOTYPE KNOWLEDGE BASE FOR 501 MODELER KNOWLEDGE ACQUISITION PROCESS.
 ===/

GOAL is model

TITLE is
"
^              501 MODELER: A PROTOTYPE INTEGRATION OF ARTIFICIAL^
                 INTELLIGENCE INTO PLANNING MODEL DEVELOPMENT^^^
This is a prototype implementation of expert system (ES) technology integrated
into a conventional computer programming application.  In this demo, an ES
will use a set of conventional routines combined with a knowledge base to
guide a knowledge acquisition process for the development of mathematical
planning models.
"

EPILOG is "Optional epilog message: successful integration; model was"

/======== RUN THE MODEL ================================================/

PROMPT run
"
Do you want to perform a simulation run with this model?
"

PROMPT num_periods
"
Enter an integer for the number of time periods you want to run this model?
"

PROMPT do_simul ""

IF cal_model is true and
   do_exo_vars are initialized and
   run is yes and
   num_periods > 0 and
   do_simul was successful and
   output was checked and
   modification was available
   then model was run
```

Figure 17 (continued).

```
IF run is no and
   modification was available
   then model was "runable but not run"

PROMPT do_reset ""
PROMPT do_init_vars ""
PROMPT do_exo_vars ""
PROMPT continue
"
The model is calibrated but cannot be run until initial values or data files
have been entered for the exogenous variables.  Do you wish to continue by
providing this data?
"

IF do_exo_vars wasn't initialized and
   continue is yes and
   do_init_vars was successful and
   do_reset was done
   then model is developing

IF continue is no and
   modification was available
   then model was "in development"

PROMPT do_br ""
PROMPT view_output
"
Would you like to review the output of the simulation run?
"

IF view_output is yes and
   do_br was successful or
   view_output is no
   then output was checked

/======== EFFECT MANUAL CHANGES ==========================================/

PROMPT modify
"
Would you like to make any changes to the current model?^^
    Select an area where you wish to effect changes:^
"

If modify was "501 Modeler Exit"
   then modification was available
```

Figure 17 (continued).

```
 IF modify is "Variable Declarations" and
    do_decl_vars is successful and
    do_reset was done
    then modification is developing

 IF modify is "Variable Initializations" and
    do_init_vars is successful and
    do_reset was done
    then modification is developing

 IF modify is "Development of Regression Equations" and
    do_sh_regr_eq is successful and
    do_reset was done
    then modification is developing

 PROMPT do_sh_cal ""

 IF modify is "Development of Calibration Equations" and
    do_sh_cal is successful and
    do_reset was done
    then modification is developing

 /======== CHECK FOR CALIBRATED MODEL ======================================/

 PROMPT do_sh_eq ""

 IF vars are minimal and
    do_sh_eq is successful          /=== do_sh_eq SEES IF #EQS = #ENDO VARS ===/
    then cal_model is true

 PROMPT do_sh_regr_eq ""

 IF do_sh_eq was refine and
    do_sh_regr_eq is successful and
    do_reset was done
    then cal_model is developing

 /======== DECLARE AND INITIALIZE VARIABLES ================================/

 IF num_l <= 0 or                        /=== ATTRIBS num_xx ADDED TO HH BY  ===/
    num_sr <= 0 and num_dr <= 0          /=== decl_vars AND init_vars KA FNS ===/
    then add_vars is true

 IF num_l > 0 and
    num_sr > 0 or
    num_l > 0 and
    num_dr > 0
    then add_vars is false
```

Figure 17 (continued).

```
IF add_vars is false
   then vars are minimal

PROMPT do_decl_vars ""

IF add_vars is true and
   do_decl_vars is successful and
   do_init_vars is successful and
   do_reset was done
   then vars is developing
```