NASA Conference Publication 10085

2nd CLIPS Conference Proceedings Volume 2

(NASA-CP-10085-Vol-2) SECOND CLIPS CONFERENCE PROCEEDINGS, VOLUME 2 (NASA) 280 p CSCL 09B
N92-16590 --THRU-- N92-16615
Unclas
G3/61 0057703

Proceedings of a conference sponsored by the CLIPS Users Group and hosted by Lyndon B. Johnson Space Center Houston, Texas September 23-25, 1991

NASA Conference Publication 10085

2nd CLIPS Conference Proceedings Volume 2

Joseph Giarratano, Editor
University of Houston - Clear Lake
Houston, Texas

Christopher Culbert, Editor
NASA Lyndon B. Johnson Space Center
Houston, Texas

Proceedings of a conference sponsored by
the CLIPS Users Group and hosted by
Lyndon B. Johnson Space Center
Houston, Texas
September 23-25, 1991

NASA

National Aeronautics
and Space Administration

Scientific and Technical
Information Branch

1991

FOREWORD

I am very pleased to have helped bring about the compilation and presentation of the papers in these proceedings for the 2nd CLIPS conference. The papers provide a breadth of topics from teaching with CLIPS to extensions of the CLIPS code. The many applications of CLIPS attest to its worth as a focus for the conference and to the appropriateness of the CLIPS Users Group itself.

As president of the CLIPS Users Group, I was particularly happy to be able to see many of the people with whom I have had contact over the past year. It was also rewarding to observe the richness of the interchange between conference attenders. The efforts of the presenters, helpers, and organizers seemed to be genuinely appreciated.

In addition to the professional interaction associated with the presentations and demonstrations, the banquet and receptions provided an opportunity for social interaction as well. It was good to see that many spouses attended these activities. Without question, the excellent presentation by astronaut Dr. Linda Godwin provided an ideal setting for an evening of good will and enjoyment. All of us who attended the conference can be proud for helping to provide an enhancing experience for everyone.

The quality of the papers presented can be readily observed in these proceedings. However, we would also like for the reader to be aware of the significant benefits that come from attending the conference and actively participating in the forum of discussion that is focused through CLIPS.

The reader is also encouraged to consider joining the CLIPS Users Group. The new CLIPS Board and the new slate of officers are enthusiastically planning more benefits to membership and the next CLIPS Conference.

To Bob Savely, Chris Culbert, Gary Riley, Brian Donnell, Joe Giarratano, Carl Armstrong, Marty Buff, Linda Cook, Linda Martin, Matt Berry, and Pat Mortensen -- and to all of the members of the CLIPS family of users, I send as my last official act as president of the CLIPS Users Group,

Best Wishes!

Len Myers

PRECEDING PAGE BLANK NOT FILMED

1 October 1991
San Luis Obispo, California

Fellow CLIPS users,

It is my distinct pleasure to provide some remarks to accompany the Proceedings of the Second Annual CLIPS Conference. The conference was both enjoyable and productive, and a quick look through these papers will confirm the many successes being had using the CLIPS language.

A successful conference requires the hard work of many people. I would like to personally thank everyone who helped out with this conference and especially thank Dr. Joe Giarra tano, Gary Riley, Brian Donnell and Carla Armstrong for all their hard work. Last, but not least, I would like to recognize and thank my predecessor, Dr. Len Myers, whose tireless efforts on behalf of the Users Group and the Conference have not gone unnoticed. In recognition of his unwavering support the CLIPS Users Group Board of Directors has established the "Len Myers Best Student Paper Award" to be awarded at each subsequent CLIPS Conference. Thank you Len.

Finally, I would like to encourage you to join the CLIPS Users Group. Besides sponsorship of the Third Annual CLIPS Conference we have several other activities planned including the continuation of our newsletter, establishment of an online CLIPS applications library and continued liaison with the CLIPS developers. With your support we can make our plans a reality.

Looking forward to seeing you at the next CLIPS conference.

Cheers!

Rodney Doyle Raines III
rraines@polyslo.calpoly.edu
President
CLIPS Users Group. Inc.

PRECEDING PAGE BLANK NOT FILMED

NASA Johnson Space Center
SOFTWARE TECHNOLOGY BRANCH

COSMIC
382 E. Broad St.
Athens, GA 30602
(404) 542-3265

CLIPS

A Tool for the Development and Delivery of Expert Systems

CLIPS is a productive development and delivery expert system tool which provides a complete environment for the construction of rule and/or object based expert systems. CLIPS was created by the Software Technology Branch of the Information Systems Directorate at NASA/Johnson Space Center (JSC) with support from the USAF. CLIPS is the first language to provide a verification/validation utility for the development of expert systems. CLIPS enabled the use of expert system technology in the JSC Mission Control Center and is being used by over 3,000 users throughout the public and private community including: all NASA sites and branches of the military, numerous federal bureaus, government contractors, 160 universities, and many companies. The key features of CLIPS are:

- **Knowledge Representation:** CLIPS 5.0 provides a cohesive tool for handling a wide variety of knowledge with support for three different programming paradigms: rule-based, object-oriented and procedural. Rule-based programming allows knowledge to represented as heuristics, or "rules of thumb", which specify a set of actions to be performed for a given situation. Object-oriented programming allows complex systems to be modeled as modular components (which can be easily reused to model other systems or to create new components). The procedural programming capabilities provided by CLIPS are similar to capabilities found in languages such as C, Pascal, Ada, and LISP.
- **Portability:** CLIPS is written in C for portability and speed and has been installed on many different computers without code changes. Computers on which CLIPS has been tested include IBM PC compatibles, Macintosh, VAX 11/780, Sun 3/260, and HP9000-500. CLIPS can be ported to any system which has an ANSI compliant C compiler. CLIPS comes with *all* source code, approximately 40,000 lines of C, which can be modified or tailored to meet a user's specific needs.
- **Integration/Extensibility:** CLIPS can be embedded within procedural code, called as a subroutine, and integrated with languages such as C, FORTRAN and ADA. CLIPS can be easily extended by a user through the use of several well-defined protocols.
- **Interactive Development:** The standard version of CLIPS provides an interactive, text oriented development environment, including debugging aids, on-line help, and an integrated editor. An interface providing features such as pulldown menus, an integrated editor, and multiple windows has been developed for the Macintosh (and similar interfaces will soon be available for MS-DOS, Windows 3.0, and X-Windows environments).
- **Verification/Validation:** A unique utility called CRSV aids in verification and validation of rules by providing cross referencing of patterns, style checking, and semantic error checking.
- **Fully Documented:** CLIPS comes with extensive documentation including a full Reference Manual and a User's Guide. An Architecture Manual provides a guide to the source code and internal operations of CLIPS. A 650-page college textbook, *Expert Systems Principles and Programming*, using CLIPS is available from PWS Kent publishers.
- **ADA Version:** A version of CLIPS developed entirely in Ada and fully syntax compatible with the C version of CLIPS is currently available for VAX and UNIX workstations.
- **Availability:** CLIPS version 5.0 is currently available. CLIPS is free to NASA, USAF, and their contractors for use on NASA and USAF projects by calling the Software Technology Branch Help Desk between the hours of 9:00 AM to 4:00 PM (CST) Monday through Friday at (713) 280-2233. Government contractors should have their contract monitor call the Software Technology Branch Help desk to obtain CLIPS. Others may purchase CLIPS (including all documentation) from COSMIC at a nominal fee for unlimited copies with no royalties. An electronic bulletin board containing information regarding CLIPS can be reached 24 hours a day at (713) 280-3896 or (713) 280-3892. Communications information is 300, 1200, or 2400 baud, no parity, 8 data bits, and 1 stop bit.

2/7/91

PRECEDING PAGE BLANK NOT FILMED

NASA Johnson Space Center
SOFTWARE TECHNOLOGY BRANCH

COSMIC
382 E. Broad St.
Athens, GA 30602
(404) 542-3265

CLIPS

Version 5.0 Announcement

The Software Technology Branch of the Information Technology Division at NASA/Johnson Space Center announces the upcoming release of version 5.0 of CLIPS. CLIPS is a powerful development and delivery expert system tool which provides a complete environment for the construction of rule-based expert systems. CLIPS is free to NASA, USAF, and their contractors for use on NASA and USAF projects by calling the CLIPS Help Desk between the hours of 9:00 AM to 4:00 PM (CST) Monday through Friday at (713) 280-2233. Government contractors should have their contract monitor call the CLIPS Help desk to obtain CLIPS. Others may purchase CLIPS from COSMIC. The key new features of CLIPS 5.0 are:

- **Object Oriented Programming:** The primary addition to version 5.0 of CLIPS is the CLIPS Object-Oriented Language (COOL). COOL supports many of the features found in commercial object-oriented languages such as Common Lisp Object System (CLOS) and SmallTalk. The features of COOL include: classes with multiple inheritance, single and multi-valued slots (with slot daemons), instances with encapsulation, and message-passing (with before, after, primary, and around message-handlers). COOL is not tightly integrated with the rule system (i.e. you cannot pattern match against instances on the LHS of a rule, and rules are not instances of the rule class). However, an extensive query system is provided for finding and/or performing actions on sets of instances which meet arbitrary user-defined restrictions. Coordination between rules and objects can be achieved easily by explicit programmer control. (A future release of CLIPS will support direct pattern-matching on objects).
- **Deffunctions:** Procedural code (called deffunctions) can be defined directly in CLIPS. Deffunctions may be used to add new capabilities to CLIPS without having to write new code in C and recompile CLIPS.
- **Generic-Functions:** Procedural code (called generic-functions) can be defined directly in CLIPS. Generic-functions can be used to overload functions in a manner similar to, but much more powerful than, languages such as Ada and C++. A user may also use generic-functions to add new capabilities to CLIPS without having to write new code in C and recompile CLIPS.
- **Global Variables:** Variables that are global in scope may be defined in a manner similar to procedural languages. These variables can then be accessed or set from within rules, message-handlers, generic-function methods, etc.
- **Integer Data Type Support:** An integer data type is supported which is represented internally as a C long integer. Floating point values are now stored internally as C double precision numbers for greater accuracy.
- **Conflict Resolution Strategies and Salience Extensions:** Seven conflict resolution strategies are provided including depth, breadth, lex, mea, simplicity, complexity, and random. Salience values can be expressions and contain global variables. Salience values can also be dynamically evaluated each time a new activation is added to the agenda or every cycle of execution.
- **Deftemplate Field Checking:** Type, value, and range checking for deftemplate field values are now supported both statically (when rules are loaded) and dynamically (when new facts are asserted) .
- **Incremental Reset:** Newly added rules are automatically initialized when defined, and any new activations are placed on the agenda. Rules may also be "refreshed" which adds previously executed activations which are still valid for a rule to the agenda.
- **Truth Maintenance:** Facts can be made logically dependent upon the existence or non-existence of other facts.

CONTENTS

Volume 1

Page

AGENDA 3

SESSION 1

Rule Groupings: An Approach Towards Verification of Expert Systems 21
Enhanced Use of CLIPS at the Los Alamos National Laboratory 25

SESSION 2A

Using a CLIPS Expert System to Automatically Manage TCP/IP Networks and Their Components 41
NMESys: An Expert System for Network Fault Detection 52
A Mission Executor for an Autonomous Underwater Vehicle 58

SESSION 2B

The Automated Army ROTC Questionnaire (ARQ) 71
Decision Blocks: A Tool for Automating Decision Making in CLIPS 76
Automated Predictive Diagnosis (APD): A Three Tiered Shell for Building Expert Systems for Automated Predictions and Decision Making 89
ICADS: A Cooperative Decision Making Model With CLIPS Experts 102

SESSION 3A

A CLIPS/X-Window Interface 115
Application of Machine Learning and Expert Systems to Statistical Process Control (SPC) Chart Interpretation 123
Application of Software Technology to Automatic Test Data Analysis 139

SESSION 3B

Acquisition, Representation and Rule Generation for Procedural Knowledge 149
Projects in an Expert System Class 162
Using CLIPS as the Cornerstone of a Graduate Expert Systems Course 166

Page

SESSION 4A

CRN5EXP - Expert System for Statistical Quality Control 173
Distributed Semantic Networks and CLIPS 177
Object-Oriented Knowledge Representation for Expert Systems 186

SESSION 4B

Linkfinder: An Expert System that Constructs Phylogenic Trees 199
Generating Target System Specifications from a Domain Model Using CLIPS 209
The Management and Security Expert (MASE) 227

Volume 2

SESSION 5A

Adding Run History to CLIPS 237
CLIPS Application User Interface for the PC 253
Expert Networks in CLIPS 267
ECLIPS: An Extended CLIPS for Backward Chaining and Goal-Directed Reasoning 273

SESSION 5B

Extensions to the Parallel Real-Time Artificial Intelligence System (PRAIS) for Fault-Tolerant Heterogeneous Cycle-Stealing Reasoning 287
PCLIPS - Parallel CLIPS 294
Separating Domain and Control Knowledge Using Agenda 307
Integrating CLIPS Applications into Heterogeneous Distributed Systems 308

SESSION 6A

Data-Driven Backward Chaining 325
Automated Information Retrieval Using CLIPS 332
Proposal for a CLIPS Software Library 344

Page

SESSION 6B

Improving NAVFAC's Total Quality Management of Construction Drawings with CLIPS 357
Validation of an Expert System Intended for Research in Distributed Artificial Intelligence 365
Testing Validation Tools on CLIPS-Based Expert Systems 382

SESSION 7A

Design Concepts for Integrating the IMKA Technology with CLIPS 395
A CLIPS-Based Tool for Aircraft Pilot-Vehicle Interface Design 407
On the Generation of Graphical Objects and Images from within CLIPS Using XView 417

SESSION 7B

Passive Acquisition of CLIPS Rules 423
YUCSA: A CLIPS Expert Database System to Monitor Academic Performance 436
A CLIPS Based Personal Computer Hardware Diagnostic System 445

SESSION 8A

PVEX - An Expert System for Producibility/Value Engineering 455
Rule Groupings in Expert Systems Using Nearest Neighbour Decision Rules and Convex Hulls 464
Debugging Expert Systems Using a Dynamically Created Hypertext Network 475

SESSION 8B

Implementing a Frame Representation in CLIPS/COOL 497
Application of a Rule-Based Knowledge System Using CLIPS for the Taxonomy of Selected *Opuntia* Species 505
The Nutrition Advisor Expert System 511

SESSION 5 A

PRECEDING PAGE BLANK NOT FILME

N92-16591

ADDING RUN HISTORY TO CLIPS

Sharon M. Tuttle and Christoph F. Eick

Department of Computer Science
University of Houston, Houston, Texas

Abstract. To debug a CLIPS program, certain 'historical' information about a run is needed. It would be convenient for system builders to be able to ask questions requesting such information. For example, system builders might want to ask why a particular rule did not fire at a certain time, especially if they think that it should have fired then, or they might want to know at what periods during a run a particular fact was in working memory. It would be less tedious to have such questions directly answered, instead of having to rerun the program one step at a time or having to examine a long trace file.

This paper advocates extending the Rete network used in implementing CLIPS by a temporal dimension, allowing it to store 'historical' information about a run of a CLIPS program. We call this extended network a *historical Rete network*. To each fact and instantiation are appended *time-tags*, which encode the period(s) of time that the fact or instantiation was in effect. In addition, each Rete network memory node is partitioned into two sets: a *current* partition, containing the instantiations currently in effect, and a *past* partition, containing the instantiations which are not in effect now, but which were earlier in the current run. These partitions allow the basic Rete network operation to be surprisingly unchanged by the addition of time-tags and the resulting effect that no-longer-true instantiations now do not leave the Rete network.

We will discuss how historical Rete networks can be used for answering questions that can help a system builder detect the cause of an error in a CLIPS program. Moreover, the cost of maintaining a historical Rete network is compared with that for a classical Rete network. We will demonstrate that the cost for assertions is only slightly higher for a historical Rete network. The cost for handling retractions could be significantly higher; however, we will show that by using special data structures that rely on hashing, it is also possible to implement retractions efficiently.

I. INTRODUCTION

One of the activities of a system builder developing any kind of software is *debugging*, "the process of locating, analyzing, and correcting suspected faults" ((IEEE 1989), p. 15). So, first, the system builders notice, one way or another, that there is a manifestation of an error in the program. Then, they debug by finding, and then correcting, the cause(s) of that particular error. In a forward-chaining rule-based language such as CLIPS ((Giarratano 1989), (COSMIC 1989)), a program's data-driven execution affects the process of debugging.

In a CLIPS program, data changes determine what happens next. One of the rules whose left-hand-side conditions are all satisfied will be chosen to have its right-hand-side actions executed. Those actions may change working memory, causing some previously-unsatisfied rules to now be satisfied, and vice versa. So, the choice of which rule to fire determines which rules will have a chance to fire next, and so affects what can happen next. To debug such a program, the system builders need detailed information about what happened during its run, including the *order* in which rules were executed, and *when* certain data were (or were not) in working memory. This *historical* information about a run will be

 PRECEDING PAGE BLANK NOT FILMED

necessary to discover why the program executed as it did.

Particularly for large CLIPS program, system builders armed only with those tools currently provided have a tedious job ahead. CLIPS allows one to run a program one rule-firing at a time (or, it allows one to *single-step* through a program), and to check what is in memory or in the *agenda*, the ordered list of rules currently eligible to fire. CLIPS can also be directed to display a trace of rule-firings and/or of working memory changes for system builder use. So, to find out, for example, when a fact was in working memory, we can carefully examine a possibly-long trace of assertions to and retractions from working memory, or we can single-step through the program, and ask to see the current list of facts when we reach various points at which we suspect that the fact is true. To find out why a rule did not fire at a particular time, we can single-step through the program up to that time, and then try to determine, from the agenda and working memory contents, why this rule did not fire then. We can find out such information using the current tools, but we may also easily overlook details in the sheer volume and monotony of the data.

CLIPS' current debugging tools are not very different from those found for other, similar forward-chaining rule-based languages. Evidence that these tools are not sufficient can be found in the current research trying to ease the debugging of large forward-chaining programs. (Domingue and Eisenstadt 1989) presents a graphics-based debugger, while (Barker and O'Connor 1989), (Jacob and Froscher 1990), (Eick et al. 1989), and (Eick 1991) suggest changes to rule-based languages, such as the addition of rule-sets, that might, among other goals, ease debugging, changing, and maintaining rule-based programs.

We would like to explore a slightly different approach to debugging: how explanation can be used in debugging CLIPS programs. The explanation subsystem envisioned will allow system builders to ask questions on the top level of CLIPS about the latest run of a program, which the system then answers, instead of requiring the system builders to run the program again, single-stepping through it, or to pore over the system trace. This explanation, designed with the problems of forward-chaining rule-based program debugging in mind, would be a useful addition to the current debugging facilities provided by CLIPS.

Many questions useful for debugging deal with the aforementioned historical details of a run. For example, system builders might ask what fired rule's right-hand-side actions contributed a particular fact to working memory, allowing it to trigger some other rule. They might ask which fired rules needed a particular fact to be true for their left-hand-side conditions to be satisfied; this gives them an idea of that fact's impact. They might ask why a particular rule did not fire at a certain time, especially if they think that it should have fired then. These questions can be answered using the current CLIPS facilities, by single-stepping through a run or by studying a trace, but the tedium would be reduced if the questions could be directly answered instead. However, one of the first hurdles to answering such questions is determining how to store and maintain a run's historical information.

CLIPS uses an inference network to efficiently match left-hand-side (or LHS) rule conditions to facts; in particular, it uses the Rete algorithm ((Forgy 1982), (Scales 1986)) for this matching. Basically, in the Rete algorithm, a network of all the LHS conditions from all the rules is built, which includes tests both for each LHS condition appearing in any rule, and for certain combinations of conditions within rules. When a fact matches a LHS condition, an *instantiation* — indicating that this condition is satisfied by this fact — is stored in the network; instantiations are also stored for combinations of LHS conditions satisfied by sets of facts. A rule instantiation represents a collection of facts that satisfies all of the rule's LHS conditions. Then, as each new fact is asserted, it is sent through the network. As a result of this propagation, rules may become eligible to be fired, if this fact's assertion causes instantiations of those rules to be created. Likewise, rule instantiations may be removed because of this fact's assertion, if the rule contains a LHS condition requiring that this fact not be true. When facts are retracted from working memory, that is also propagated through the network, causing the removal of instantiations including the now-retracted fact.

In this paper, a generalization of the Rete network, called a *historical Rete network*, is proposed that allows the storing and maintenance of historical information for a single

run of a CLIPS program. We have two main objectives in modifying Rete for this purpose: CLIPS programs using the modified network must still run almost as efficiently as before the modifications, so that run-time operation during program development is not overly impeded, and the modified network should allow reasonable maintenance and retrieval of a program run's historical information. Since the Rete network implements rule instantiations, it is feasible to 'tag' condition and rule instantiations with when they occurred during a run. Including this information within the network will allow us to design top-level explanation facilities that can more easily and efficiently answer 'historical' questions about a run, and thus ease the task of debugging for system builders. Storing and maintaining this information is just one of several components in providing such explanation, but it is a necessary and important aspect.

The rest of the paper will be organized as follows. Section 2 briefly introduces the Rete algorithm, then discusses our use of rule-firings as the basis for time, and then describes how time-tags, along with current and past partitions, can be used to to store CLIPS program run history. Since some questions that a system builder might ask would involve knowing what the agenda looked like at a certain time, section 3 covers how an agenda copy may be reconstructed on demand. Section 4 then briefly describes how historical information about a program run may be retrieved in the context of gathering data for answering several different kinds of questions useful for debugging. Finally, section 5 concludes the paper.

II. MODIFICATIONS TO THE RETE NETWORK

A. Introduction to Rete Networks

Before discussing the necessary structural changes, we will briefly review 'normal' Rete networks. (For a fuller description, see (Forgy 1982), (Scales 1986), and (Gupta 1987).) In a Rete network, there are three kinds of memory nodes: alpha nodes, beta nodes, and production nodes. (For simplicity, 'node' will stand for test nodes along with their corresponding memory.) There is an alpha node for each LHS condition; the alpha node stores an instantiation for each fact matching this condition. A beta node contains instantiations representing two or more consistently-satisfied LHS conditions from a particular rule (or rules), and a production node holds instantiations that satisfy all of the LHS conditions of a rule.

In a Rete network, two alpha nodes representing rule conditions are joined into a beta node, containing instantiations that consistently represent both conditions being true. Then that beta node is (typically) joined with another alpha node into another beta node, containing instantiations that consistently represent these three conditions being true, and so on until all of a rule's LHS conditions have been represented, at which point, instead of leading to a beta node, a beta node and alpha node are joined into a production node, containing 'complete' rule instantiations that are eligible to fire. And, when it is being built, as conditions are added to the network, each condition appears in the network only once; if it is used in several rules, then that alpha node has a number of successors, and likewise, if a set of conditions appears in several rules, the section of the Rete network leading to a beta node representing that set of conditions may also be shared among several rules.

Figure 1 shows a simplified Rete network for a single CLIPS rule, rule-13, given in Table 1. (We will use facts, instead of fact-ids, in instantiations in most of the figures, for greater clarity.) Each of the three LHS conditions in rule-13 has a corresponding alpha node that tests for matches and stores an instantiation of each matching fact. So, we see in Figure 1 that working memory fact (p 1 3) matches rule-13's LHS condition (p ?X ?Y), that fact (q 3 5) matches (q ?Y ?Z), and that fact (r 1 7) matches (r ?X ?Q). The first two conditions are then joined into a beta node, which tests if any of the facts matching those two conditions are compatible, and then stores any combinations passing the test. (p 1 3) and (q 3 5) both match their respective conditions with ?Y = 3, so an instantiation for that pair is stored in the beta node. Then, that beta node is joined with the remaining condition, and since

this is the last condition, any instantiations resulting from these compatibility tests will be instantiations for rule-13, stored in a production node. The variable ?X is 1 in both the beta node's only instantiation and in (r ?X ?Q)'s only instantiation, so they can be combined into a compatible instantiation for the entire rule, and so an instantiation is stored in rule-13's production node.

```
(defrule rule-13
   (p ?X ?Y)
   (q ?Y ?Z)
   (r ?X ?Q)
=>
   ({ rule-13 actions }))
```

Table 1. A CLIPS rule

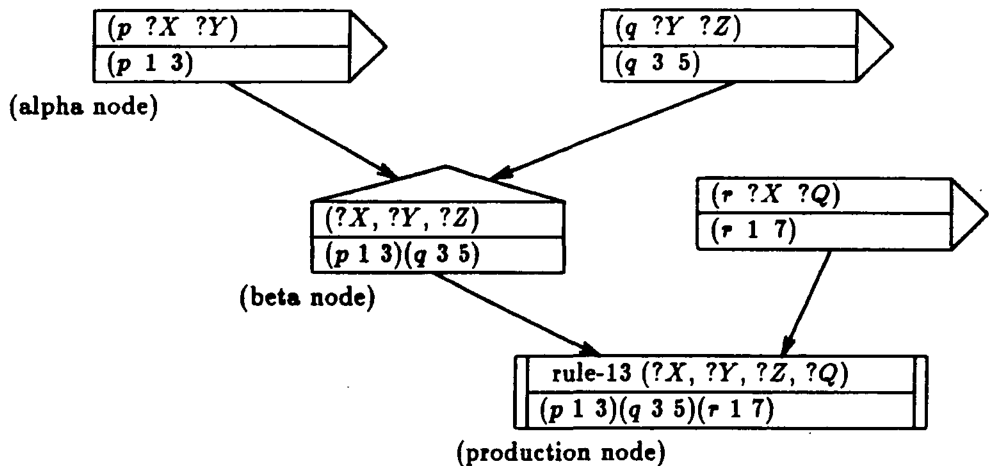


Figure 1. A 'regular' Rete network

In general, when a fact is asserted, it is added to the working memory element (or wme) hash table, which stores all the facts currently in working memory. Each wme hash table entry includes pointers to all of the LHS conditions (or, alpha nodes) in the Rete network that match this fact; these pointers allow us to avoid another search of all the conditions if the fact is later retracted. The newly-asserted fact is then compared to every alpha node, and, if the fact matches that LHS condition, an instantiation is stored in the alpha node, and a pointer to this alpha node is stored in the fact's wme hash table entry. We then visit all of that alpha node's successors, seeing if the new instantiations in a node result in new consistent instantiations at the succeeding node. Any new rule instantiations added to production nodes are added to the agenda.

CLIPS uses an *agenda*, a priority queue containing the currently-eligible rule instantiations in the order that they should be fired (if they stay on the agenda long enough); the one on top of the agenda will be the next chosen to fire. An agenda of size n does not necessarily display the next n rules that will be fired, however, because each rule-firing has the potential to change working memory, causing rule instantiations to join and to be deleted from the agenda. CLIPS uses the following conflict resolution strategy (COSMIC 1989):

(1) The instantiations are ordered by the *salience*, or priority, of the rules involved; the instantiation with the highest salience is on top of the agenda.
(2) If instantiations have the same salience, then the one which became true most

recently is preferred over earlier ones by being placed nearer to the top of the agenda.

B. Time-Related Considerations

Before we can modify Rete to store historical information, we need to decide on some time scale, so that we can store the data needed to determine when and in what order rules fired, and when facts were (and were not) in working memory. To further facilitate debugging, we would like a time-unit that is central to CLIPS program operation. Conceptually, rule-firings are the major units of action in a CLIPS program. Computations are done when a rule fires, and the computations performed are those of the selected rule.

The Transparent Rule Interpreter (TRI), a graphical debugger for forward-chaining rule based languages described in (Domingue and Eisenstadt 1989), uses rule-firings as the 'time' scale in its 'musical score' framework for graphically representing forward-chaining execution. Also, note that the existing debugging aids within CLIPS are rule-firing based. As previously mentioned, CLIPS allows system builders to run a program one rule-firing at a time, in order to more closely examine its execution while debugging; the single step in this single-stepping process is one rule-firing. And, the CLIPS (*run*) command concludes by printing out how many rules have fired during the program execution, and if one chooses to display rule-firing information during program execution, then each is numbered by its order of occurrence within the run. The rule-firings are considered a measure of how much or how little has occurred. Therefore, it is quite natural to use rule-firings as a time basis. A counter starts at zero at the beginning of each run and is incremented with each rule-firing. This also has the useful feature of being comparable between runs; for example, running the same program with the same data twice, the fifth rule to fire does so at the same 'time' in both runs — at time counter value five — which can make it easier to compare and contrast, for example, two runs of the same program using slightly different sets of facts.

C. Historical Rete Networks

Historical Rete networks, proposed by this paper, differ from classical Rete networks in two major respects: each instantiation stored within the network has a *time-tag*, which gives the period(s) of time that the instantiation was in effect, and each memory node has its contents partitioned into two sets: a *current* partition, containing all instantiations currently in effect, and a *past* partition, containing all instantiations in effect earlier in this run.

A *time-tag* is a set of one or more intervals stored with a fact or instantiation, which gives the time period(s) during a run that the fact or instantiation was in effect. For brevity, we will use 'true' to describe a fact in working memory, an instantiation representing a condition or conditions satisfied by working memory, and an eligible rule instantiation. (Note that, because of *refraction* ((Brownston et al. 1985), pp. 62-63), a rule instantiation that fires becomes ineligible, even if its RHS actions do not cause any of its LHS conditions to become unsatisfied, until at least one of its facts is retracted and asserted again.)

This time-tag is different from the time tag mentioned in (Brownston et al. 1985), p. 43, because that time tag is associated just with facts, and not also with instantiations as ours is, and it consists of only one integer, representing when that fact joined working memory or was last modified. The time-tags we use store more information, about both facts and instantiations. An *interval* is a component $(x\ y)$ in a time-tag, in which x was the time when that fact or instantiation became true and y was the time when it became no longer true — when the fact was retracted from working memory, when one or more conditions represented by an instantiation were no longer satisfied, or, for a rule instantiation, when it left the agenda. An *open interval* indicates that the fact or instantiation is still true; we write such an interval as $(x\ *)$.

Time-tags are found in the wme hash table and the historical Rete network. Each wme hash table entry now also includes the time-tag for that fact. When a fact is retracted from working memory, its entry is not removed from the table; instead, the open interval

in its time-tag is closed with the current time. So, the wme hash table stores all the facts that are *or have been* in working memory during this program run. We can tell if a fact is currently true by simply seeing if the last interval of its time-tag is open. (The wme hash table entry should probably also store all of the fact-ids that a fact has had during a run.)

The time-tags give the time period(s) during a run that a fact or instantiation was true, and so they are part of the run's historical information. The partitions, on the other hand, serve a very different purpose: they allow the basic Rete network *operation* to be surprisingly unchanged by the addition of time-tags and the resulting effect that instantiations do not leave the historical Rete network. (Instantiations that no longer hold have their time-tags' intervals closed, but those instantiations are not actually removed from the network.) If we keep a memory node's no-longer-true instantiations in a past partition, then the instantiations in each memory node's current partition are exactly those that would appear in the corresponding 'normal' Rete network memory node. This, then, allows most historical Rete network operations to take place as in a 'normal' Rete network: the actions that involve *all* instantiations in normal Rete now involve all instantiations *in current partitions only* in historical Rete.

```
(defrule rule-1
   ?p_addr <- (p ?X ?Y)
   (q ?Y ?Z)
   (r ?X ?W)
=>
   (assert (r ?X ?Z))
   (retract ?p_addr))

(defrule rule-2
   (r ?X ?W)
   (s ?Z ?X)
=>
   (assert (q =(*?W ?X) ?Z)))
```

Table 2. Rules for Historical Rete Example

Figure 2 shows a historical Rete network as it would be right before time 4 for the initial facts given and for the rules in Table 2. Following the chronology shown in Figure 2, one can see how the facts propagate through the historical Rete network, how the time-tags are set, and how instantiations come and go (and move from current to past partitions). As shown, the instantiation of rule-2 matching facts (r 4 6) and (s 2 4) will be the next to fire, at time 4. The action is basically the same as a classical Rete network, but now one can see such historical details as, for example, why rule-1 could not fire at time 3: because it had no true instantiations then.

Conceptually, a historical Rete network will look like figure 2; however, for performance reasons, we will likely implement it slightly differently; for example, we will very likely incorporate hashing into it. Hashing has been proposed for Rete networks to improve performance (for example, in (Gupta et al. 1988)); it will be useful for historical Rete networks as well. In particular, past partitions of nodes should be hashed, so that a particular past partition entry can be found in constant time (in the average case).

When a fact is asserted into a historical Rete network at time counter value t, it has the time-tag (t $*$) added to its wme hash table entry and also to any new instantiations resulting from its propagation through the historical Rete network. The propagation through the historical Rete network is essentially the same as for a "classical" Rete network, except that

(a) each new instantiation that results is placed in the corresponding memory node's current partition (instead of in its 'only' partition, in the normal case),
(b) beta tests are performed for instantiations in current partitions (but these current partitions contain the same instantiations as the 'only' partitions in the normal case), and
(c) (as already mentioned) each instantiation that results from asserting this fact has the time-tag interval $(t\ *)$ appended to it.

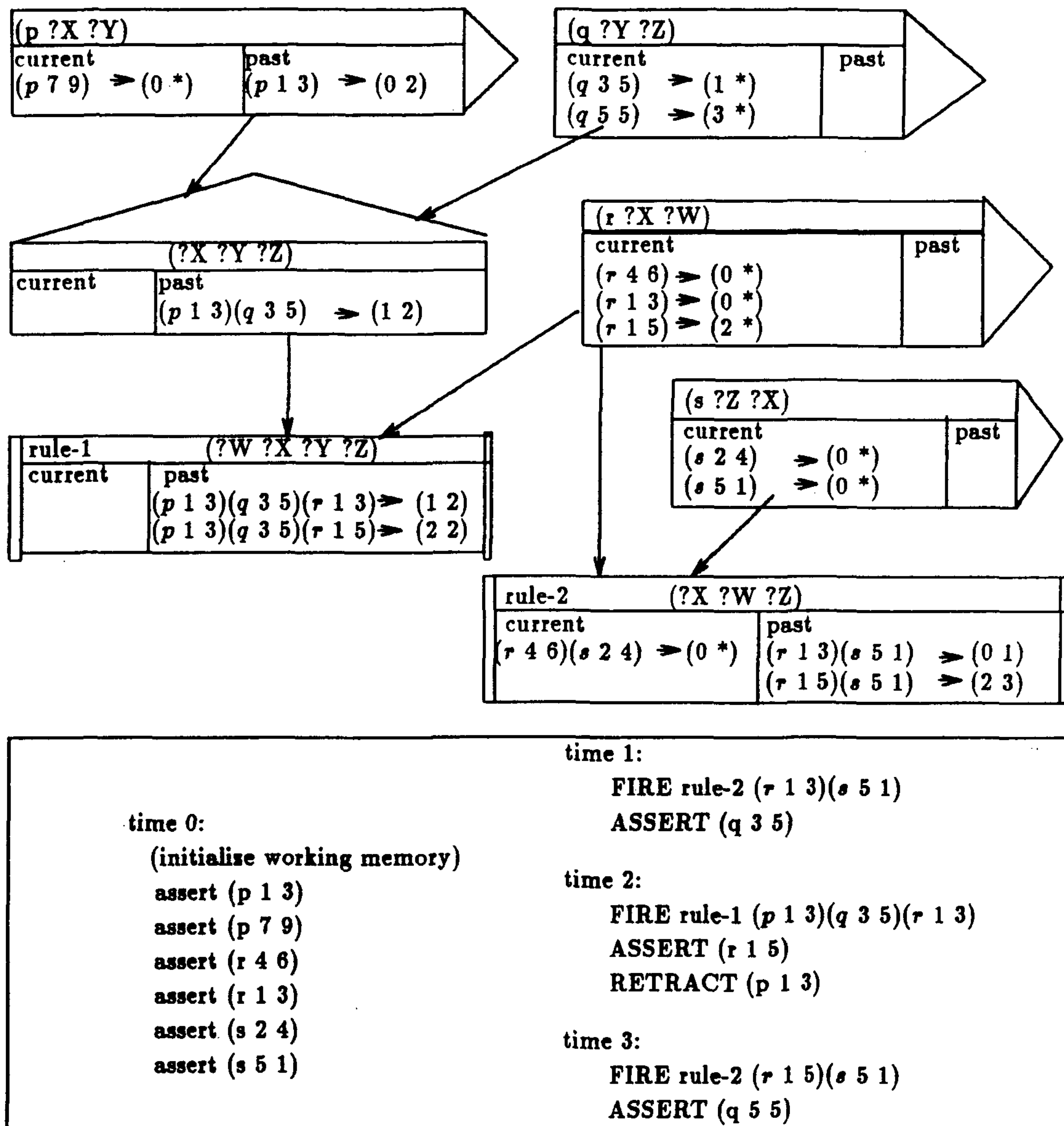


Figure 2. A Historical Rete Network

When asserting a fact, the computational cost of keeping historical information is quite low. The computations for deciding if an instantiation has to be propagated are still the same as in the original network. The only additional computational overhead comes from adding the time-tag intervals to each fact and instantiation, and from updating any node hash tables being used as necessary.

Retracting a fact from the historical Rete network at time counter value t has a few

additional differences compared to its "classical" Rete counterpart. In a "classical" Rete network, the wme hash table entry for the fact to be retracted is found, and the pointers within this entry to every alpha node matching this fact are traversed in turn. From each alpha node matching this fact, we remove any instantiations making use of the retracted fact, and then continue to all the nodes reachable from this alpha node, searching for, and removing if found, any instantiations using the fact being retracted. When done with all of that, the fact's wme hash table entry is deleted.

In a historical Rete network, the process is basically the same — we find the fact's wme hash table entry, follow its pointers to the alpha nodes matching this fact, and search from each of these alpha nodes for all of the instantiations using this fact. Where we searched all the instantiations in the "only" partition of each encountered memory node in the "classical" case, now we search only the instantiations in the current partition (which has the same collection of instantiations as the classical Rete network's only partition, for each memory node). Instead of deleting those instantiations that make use of the fact being retracted, we move them from the current partition to the past partition of their memory nodes, and close the open interval in their time-tags with the current time-counter value. (For example, if the open interval was $(a\ *)$, and the current time counter value is t, then the time-tag interval becomes $(a\ t)$.) Finally, when done with all that, instead of removing the wme hash table entry, we merely close the open interval in its time-tag with t.

Comparing retraction in the historical and regular cases, most of the differences are minor: the entry is removed from the wme hash table in the regular case, but just has its open time-tag interval closed in the historical case, and each instantiation using the retracted fact is deleted from the network in the regular case, but is moved from the current to the past partition of its memory node in the historical case, with its time-tag also closed with the current time counter value. However, there is a bit more work than might be apparent in moving an instantiation from the current to the past partition. To keep things most straightforward if an instantiation is true for more than one period of time (for example, if a rule contains a negated condition that is true, then false, then true again), we would like to keep no more than one entry per instantiation in a memory node's past partition; this will require a search of the past partition. If a previous instance of this instantiation is found, we append the newly closed time-tag to it, and if not, then we add a new entry with the instantiation and its newly closed time-tag. Fortunately, if we hash the past partitions, then this search will take constant time, in the average case.

Here is a simple example, to demonstrate the use of partitions. Let i be an instantiation that was true between times a and b and between times c and d. When it became true at time a, its time-tag was $(a\ *)$, and it clearly belonged in the current partition of the appropriate memory node. When it became no longer true at time b, its interval was closed, resulting in the time-tag $(a\ b)$, and i was removed from its memory node's current partition and moved into its past partition. At time c, it became true again.

There are a number of possibilities about how to proceed; we will leave the *past* instance of this instantiation in the past partition and create a current instantiation in the current partition. Keeping two copies of an instantiation when it is true and has been false in the past, one in each partition, simplifies run-time operation of the historical Rete network. It allows us to assume that instantiations in current partitions have exactly one interval, which is known to be open, in their time-tags. It also lets us avoid accessing the past partition when adding an instantiation. However, as mentioned, the past partitions will be accessed when instantiations are removed from current partitions, so that the newly-closed time-tag can be appended to an existing entry, if one exists. That way, each instantiation has at most one entry in its memory node's past partition, whose time-tag includes all of the time periods that this instantiation was true, instead of having a past partition entry for each interval that the instantiation was true.

So, assuming that i is the only instantiation in its memory node, then after time b and before time c the memory node's partitions are as shown in Table 3:

current: —
past: i, $(a\ b)$

Table 3. After time b, before time c

Now, when i becomes true again at time c, the past instance of the instantiation will be in the past partition, and the current instance will be in the current partition, as shown in Table 4:

current: i, $(c\ *)$
past: i, $(a\ b)$

Table 4. After time c, before time d

Finally, when i becomes false again at time d, then the current instance is removed from the current partition, and the now-closed time interval is added to the existing past partition entry's time-tag, as shown in Table 5:

current: —
past: i, $(a\ b)(c\ d)$

Table 5. After time d

These preliminary intuitions suggest that the addition of time-tags and current and past partitions does not fatally increase the overhead of asserting facts to and retracting facts from the historical Rete network. They also suggest that we will meet our goal of keeping the run-time operation of the historical Rete network reasonably close to that of the original version, while still allowing historical information to be maintained within. The major cost will be the storage of the historical information.

III. AGENDA RECONSTRUCTION

Since the agenda determines which rule fires next, its changing contents and their order are part of a run's historical information. And, for answering certain debugging-related questions, the agenda's state at a particular time will, indeed, be needed. For example, consider the question "Why did rule X not fire at time T?". Using the historical information in the historical Rete network, we can find out, from rule X's production node, if any instantiations of rule X were true — and thus eligible to fire — at time T (Any rule instantiation whose time-tag has an interval containing T was eligible at time T.) However, once we find that it was true then, we need the agenda from that time to obtain further details about why rule X did not fire. For example, with the agenda, we can see how many other rule instantiations were above rule X's highest instantiation. Such details may make it easier to determine what would be needed for rule X to fire at time T.

Since past states of the agenda may be useful in debugging a CLIPS program, we need to determine how to handle agenda history. We would like to avoid storing a copy of the agenda for every value of the time counter, because the potentially large number of instantiations in common between 'consecutive' agenda copies makes this seem like a poor use of space. It seems preferable to store enough information to reconstruct an agenda copy when desired. This reconstructed copy could be used by an explanation system to answer

questions, could be printed for direct system builder use, or could be modified to answer follow-up questions. Furthermore, the information used to reconstruct the agenda may also be useful for other purposes, perhaps more conveniently than if it were in the form of literal agenda copies.

Our current plan is to use information stored with only moderate redundancy to reconstruct the agenda at a particular time reasonably quickly. The needed information will be stored in an *agenda-changes list*, containing a chronological list of all changes made to the agenda during a program run. Three kinds of changes are possible: a rule instantiation can be added (an ADD), a rule instantiation can be removed to be fired (a DEL/FIRE), and a rule instantiation can be removed because at least one of the rule's LHS conditions is no longer satisfied by working memory (a DEL/REMOVE). Each change also includes the time of that change, even though the list is ordered, for easy searching for changes from a particular time period. And, finally, each agenda-changes entry also contains some representation of the instantiation being added, deleted/fired, or deleted/removed. Notice that, for any single time counter value, there will be exactly one DEL/FIRE entry, and zero or more DEL/REMOVE's and ADD's.

To construct a copy of the agenda as it was at time counter value T, we basically search the agenda-changes list entries from time 0 to time T. Then, for each ADD, we see if it was still on the agenda at time T, and if so, we add it to the agenda copy being constructed. We must start at the beginning of the agenda-changes list each time because a very-low-priority rule may be instantiated from the very beginning of a run, but not fired for a very long time because other rules always take priority. However, we can, of course, safely ignore all changes made to the agenda after time T.

To see if an ADD'ed instantiation from the agenda-changes list was removed from the agenda before time T, we think the best approach will be to search for the instantiation's entry in the past partition of its rule's production node. With hashed past partitions, this should normally take constant time. If found, then we find the time-tag interval beginning with the time of this ADD (after all, this rule instantiation was added to the agenda at this time because it had become eligible). If this interval was closed before time T, then this instantiation was not on the agenda at time T, and we do not need to add it to our agenda copy. Otherwise, it was still eligible then, and we should add it. (If the instantiation is not found in the past partition, then it is in the current partition, in which case, being still true currently, it also was true at time T, and should be added to the agenda copy. Notice, however, that this can only occur if an agenda copy is being constructed during a run, for example while single-stepping through it.)

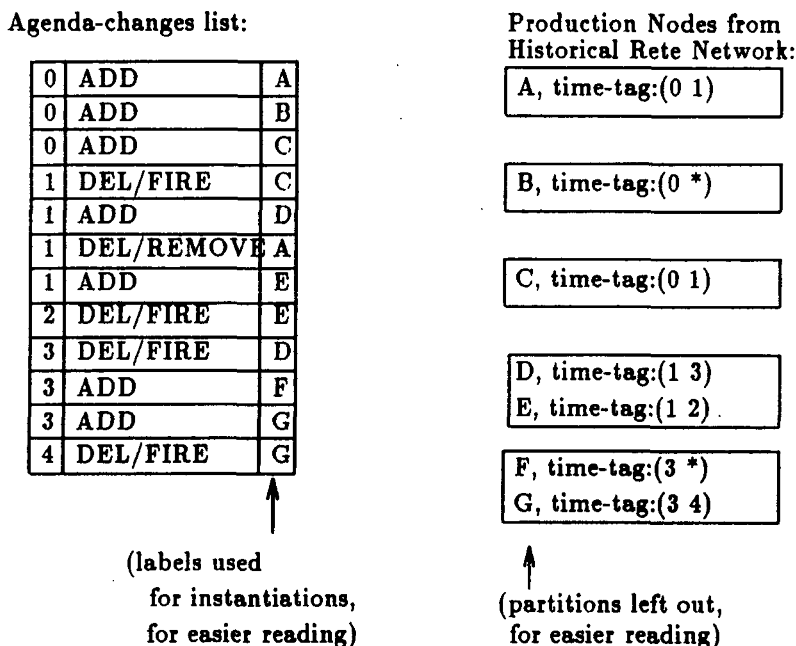
Using the production node in this way should take less time than other alternatives, which involve possibly time-consuming searches of other, non-hashed data structures. For example, we could add every ADD'ed instantiation encountered to the agenda copy, and then could remove any that left before time T as we come to their DEL/FIRE's or DEL/REMOVE's. However, that would involve a search of the agenda copy every time that an instantiation had to be removed. Similarly, for each ADD'ed instantiation, we could search down the agenda-changes list to see if it has a DEL/FIRE or DEL/REMOVE before time T; but, this would involve searching down the agenda-changes list for each ADD'ed instantiation.

A pleasant advantage to handling the ADD's in chronological order is that they can be added to the agenda copy using the same means that the system adds instantiations to the agenda during run-time, using CLIPS' conflict resolution strategy. For each ADD'ed instantiation that was still on the agenda at time T, we start at the top of the agenda copy, and compare the salience of the 'new' instantiation to that of each of the instantiations on the copy in turn, until reaching one whose salience is the same as or less than the 'new' instantiation's salience. The 'new' instantiation will be placed directly above that instantiation. We can safely stop searching down the copy after reaching one with the same salience because, since we are handling the ADD's in chronological order, recency dictates that this new instantiation, joining the agenda later than any of those already in the copy,

should go on top of those with the same salience.

Here is a simple example of agenda reconstruction. We have an agenda-changes list as shown on the left in Figure 3, in which we use letters to represent rule instantiations. To make the figure easier to read, current and past partitions are not indicated. Assume that the saliences of the instantiated rules labelled by A, B, C, F, and G are all zero, and that those of D and E are 1.

Figure 3. Data for Agenda Reconstruction Example



To reconstruct the agenda from right before time 2, before E fired, we start at the top of the agenda-changes list; the first entry is the addition of A. We check its entry in its rule's production node, and see that it was removed from the agenda at time 1, and so does not belong in the copy. The next entry is the addition of B at time 0. B does not have a past partition instance at this time, as it is still on the agenda, and so it was also on the agenda at time 2; B becomes the first instantiation in the copy.

C is also added at time 0, but is removed — by being fired — at time 1, so it is not put on the copy. Since the next entry is a DEL/FIRE, we go on to the entry after that, the addition of D. D is not removed until time 3, so it belongs on the copy. D's rule's salience of 1 is greater than B's, which is 0, so D is placed on top of B, as shown in Table 6:

Agenda copy
D — salience: 1
B — salience: 0

Table 6. D added to agenda copy

The next entry is ignored, since it is a DEL/REMOVE. For the next, E should be in the copy, since it is still on the agenda at time 2; its salience is the same as D's, but E is more recent, and so E goes on top of D, as shown in Table 7. The next entry occurred at time 2, and so, since we want the copy to be as the agenda was right *before* time 2, we stop now. The final agenda copy is the one shown in Table 7. If one writes out the agenda from time 0 onward, adding and deleting as specified, one sees that this is, indeed, the state of

the agenda as it was right before time 2.

E — salience: 1
D — salience: 1
B — salience: 0

Table 7. *E* added to agenda copy

One might question why the DEL/FIRE's and DEL/REMOVE's are kept in the agenda-changes list at all, since they are ignored during agenda reconstruction. They are worth keeping for when the states of the agenda *over a period of time* are desired. To observe the agenda between times T and U, having DEL/FIRE's and DEL/REMOVE's in the agenda-changes list keeps the system from having to reconstruct an agenda copy for time T, to then completely reconstruct another for time $(T+1)$, and so on through time U, starting over at the beginning of the agenda-changes list each time. With the DEL/FIRE's and DEL/REMOVE's, the system can instead reconstruct the agenda for time T, then apply each of time $(T+1)$'s agenda-changes list entries to the copy, resulting in time $(T+1)$'s agenda, and so on to time U, updating the previous time value's copy instead of completely rebuilding it each time. Similarly, the DEL/FIRE's and DEL/REMOVE's allow us to show what happened to the agenda, and in what order, during a single value of the time counter, if that is desired by the system builder.

IV. USING THE HISTORICAL RETE NETWORK TO ANSWER QUESTIONS FOR DEBUGGING

We will now consider how a CLIPS program run's historical Rete network and associated data structures can be used by an explanation subsystem in answering several types of questions useful for debugging. We will not discuss all the aspects involved in answering these questions, but will instead concentrate on which historical data should be collected for use in eventually answering them, and how to obtain this information.

A. When was fact F in working memory?

This question is very simple to answer using the stored historical information, and is also very useful for debugging. For example, if F is a control fact, asserted to indicate that the program has identified a particular sub-situation, then knowing when F was in working memory lets the system builders know when that sub-situation was considered during the run, if ever. If a discovered fault involves F, then knowing when F was in working memory lets the system builders concentrate on those time periods in their subsequent efforts. If F was *not* in working memory during the run, that may also be important, especially if the system builders think it should have been; its not being in working memory may explain why certain actions were not done. In addition, knowing when F joined working memory makes it easy to find out which rule asserted it (if it was not asserted directly by the system builder); we just find out what rule fired at that time, and double-check that its RHS actions could, indeed, assert F. (We may have a separate rule-firings list, or we can find out what rule fired at time T by looking for the DEL/FIRE entry with that time in the agenda-changes list.)

To find out when fact F was in working memory is even easier than finding out what rule (probably) asserted it, since we plan to include time-tags in the wme hash table. All we need to do is access fact F's wme hash table entry, and copy the time-tag from that entry (and probably the corresponding fact-id(s) as well). For example, if F's time-tag is $(a\ b)(c\ d)$, then F was in working memory from the ath rule firing to the bth rule firing, and then again from the cth rule firing to the dth rule firing. Since CLIPS associates a different fact-id with

each new assertion of a fact (following a retraction), it would also be potentially useful to the system builders to include with each time period the fact-id assigned to the fact during that period.

There is also the possibility of giving the system builders the time periods in a form other than the relative rule-firings; for example, the time-periods could be expressed as the actual rule instantiations that fired, or as the actions that were done. However, whether this would be better for debugging, how such an answer can be expressed clearly, and how the system builders can perhaps be allowed to specify dynamically which they would prefer, are still open questions.

B. What facts matched LHS condition L, and when?

This question could be useful for debugging when the system builder is interested in strange program behavior related to a particular LHS condition. If the system builder considers this condition to be key to one or more rules' firing, then knowing what satisfied it, and when, may shed light on why rules containing this LHS condition did or did not fire. The facts which satisfied L, and when each was in working memory, can be used in subsequent questions; and, if L was never satisfied, then that in itself may reveal to the system builders why certain actions were not performed by the program.

Given the historical Rete network, answering this question will be easy, as long as L is indeed a LHS condition in one of the rules. We access this LHS condition's alpha node within the historical Rete network; each instantiation stored in that alpha memory, whether in the present or past partition, corresponds to one fact that matched L. Moreover, each instantiation contains the time-tag giving when that fact matched L. So, this alpha node's contents constitute the data for answering this question.

C. What fired rule instantiations' LHS's included fact F?

This question allows system builders to easily find out which rule firings actually made use of a particular fact. For example, this could reveal, if F should not have been in working memory, just how much "damage" it caused, in terms of instantiations firing that should not have fired. The answer to this question can also make visible some of the effects of a previous instantiation firing: if one fired instantiation asserts fact F, and the system builders want to see if that action directly contributed to other actions, then the answer to this question gives them that information. And, if F is a control fact, inserted specifically to control what rule is to fire in a particular scenario, then if the answer to this question is that *no* rule fired using this fact, that could indicate to the system builders (1) that the fact was never in working memory, or (2) that the rules had an error (for example, a typo, or a missing field) in the condition that was supposed to correspond to that fact, or even (3) that the rules that were supposed to assert F did not do so, or asserted it incorrectly.

Although not directly requested, the answer should include the time periods that this fact was in working memory. This might be useful, because the system builders might notice, for example, that during one period the fact contributed to several rules' firing, and in another it did not. We should also include the time of firing of each rule instantiation that used this fact: this lets the system builders know when in the run this particular fact played a role, and it gives them the time information in case they want to follow up this question with one about why a certain rule instantiation did or did not fire at one of those time values. And, if the system builders choose to single-step through the run again, they know which rule-firings to pay particular attention to.

To collect the information for answering this question, we start at the wme hash table entry for fact F, and copy the time periods that F was in working memory. Then, using the hash table entry's pointers to all alpha nodes matching this fact, we travel in turn to each such alpha node.

From each alpha node that matches F, we travel to all of the production nodes

reachable from that alpha node. (An alpha node may lead to more than one production node because, as stated earlier, if a LHS condition is used in more than one rule, its alpha node is shared in the historical Rete network.) At each of these production nodes, we search its past partition for instantiations containing F. Each found is a rule instantiation that used F, and is no longer eligible — now, we need to see if it actually fired. After all, it may have left the agenda because one of its LHS conditions became unsatisfied before it could fire. We check every time value that it left the agenda — for example, if its time-tag is $(s\ t)(w\ x)$ then it left the agenda at times t and x — and see if the rule instantiation that fired at that time is, indeed, this instantiation. If so, then we have found an instantiation that fired, and that used F, and so it should be added to the list-in-progress of instantiations to be included in the answer. We continue in this way until we have checked all the past partition instantiations in production nodes reachable from alpha nodes for LHS conditions matching F. At that point, we have collected all of the fired rule instantiations that included fact F, and we can present them to the system builder in some reasonable form.

D. Why did rule X not fire at time T?

This type of question has the potential to be particularly useful for the purpose of debugging. Once system builders notice that a rule that they thought should fire at a particular time did not, having this question available — even with only low-level suggestions for what caused the "error" — could save them much tedium in terms of single-stepping through a run to the time of interest, poring over a long trace of rule firings, and/or adding print statements to particular rules. Knowing why a rule did not fire may lead directly to an error that kept a rule from firing, or it may indicate gaps in the system's rules (or data). If a particular unsatisfied LHS condition kept it from firing, then the system builders might immediately notice any obvious typos as soon as they are shown that condition. Or, the condition might cause the system builders to think of a fact that they thought was true, and that should have satisfied this condition; that might suggest what question they should ask next. (For example, they might ask what rules have RHS actions that could have asserted that particular fact. This follow-up question does not involve historical information about the run, but it is useful in this particular scenario. Historical information would come into play again with the likely-follow up to that question: why the rules that could have asserted that fact did not themselves fire.) If, on the other hand, the rule was eligible to fire at time T, but another fired instead, then knowing the relative saliences of the fired rule and rule X might give the system builders clues that salience-adjusting is needed, or might point out a rule (or rules) that should not have been eligible at time T, but were.

Again focusing on what historical data should be collected and how it can be obtained, we first reiterate the two basic reasons for a rule not firing: either its LHS was not satisfied, or it was eligible to fire, but was not on top of the agenda. To find out which of these is the case, we start by going to rule X's production node in the historical Rete network. We check the time-tags of all of the instantiations in this production node, in both the current and past partitions, and see which, if any, contain intervals including T. Each such instantiation was eligible to fire at time T, and we should add it to a list of instantiations of rule X that were eligible to fire at that time.

When we are done checking all of rule X's production node's instantiations, the list of eligible instantiations built will determine what we do next. If this list is empty, then rule X did not fire because it was not eligible to fire at that time; we should next determine what LHS conditions were not satisfied then. If this list is not empty, then rule X did not fire because none of its instantiations were on top of the agenda right before time T; we should next determine, in this case, why other instantiation(s) preceded rule X's instantiations on the agenda.

If the rule was not eligible to fire at time T, we will traverse the historical Rete network backwards from rule X's production node, searching the nodes corresponding to rule X's LHS conditions. These alpha and beta memory nodes will be searched for instantiations

current at time T: both current and past partitions of each node will be searched, and each instantiation's time-tag will be checked to see if T lies within any of its intervals.

If a beta node is found to have no instantiations that were true at time T, then the corresponding inter-condition test was not satisfied then. Likewise, if an alpha node is found to have no instantiations that were true at that time, then no fact matched this LHS condition then. Eventually, in this way, we build a list of unsatisfied LHS conditions and inter-condition tests, and this list will be used in constructing the answer, so that the system builders will know which conditions of rule X were not satisfied at time T; these need to be satisfied, if rule X is to even be eligible to fire then.

In the case that the rule was eligible to fire at time T, we could simply report what rule instantiation did fire then; however, that would not give such potentially-useful information as, for example, just where in the agenda rule X's instantiations were at time T. We can gather additional details in the following way. First, we reconstruct a copy of the agenda from right before time T, using the already-discussed agenda reconstruction algorithm. Then, we search down the agenda copy from the top to find out how far down the agenda the highest instantiation of rule X was, and also how many of the instantiations above rule X's highest one had higher saliences; such instantiations will always be chosen to fire before rule X's, if all are on the agenda concurrently. The remainder of the instantiations above rule X's highest one are there because they joined the agenda more recently — for rule X to fire before these, those rules will have to be instantiated earlier, or rule X will have to be instantiated later. If there are too many instantiations above rule X's to readably present them all to the system builders, then we can still at least tell them how many were above it, and how many of those had higher salience; and we should include the instantiation on the agenda top, and its salience, in any case.

In these examples, the historical information in the historical Rete network and associated data structures makes obtaining specific details about a run straightforward and reasonable. This will help a great deal in developing a practical system for answering questions such as the above. Storing historical information within the network makes it easy to update as the run proceeds, and leaves it where it can be easily obtained for different kinds of debugging purposes — such as different kinds of questions — after the run concludes. With the system able to answer such questions, the system builder will be able to easily obtain historical information about a run.

V. CONCLUSIONS AND FUTURE WORK

We have proposed that historical information about a run of a CLIPS program be stored within a historical Rete network used to match CLIPS rules' LHS's to working memory facts. This paper has explained how a Rete network can be modified for this purpose, and how the historical information thus stored can subsequently be used by a question-answering system to be designed to help with debugging a CLIPS program.

It should be feasible to store and maintain this historical information without degrading CLIPS run-time performance too badly. In fact, it is noteworthy that it can be integrated so easily into the Rete network. The basic Rete propagation is almost unchanged; the only additions involve peripheral actions such as appending time-tags, and moving no-longer-true instantiations from current to past partitions. Using hashing in various places, the time to perform these additional duties should be quite reasonable. Keeping an agenda-changes list as well will also allow reasonable agenda reconstruction whenever the system builder (or question-answering system) needs a past agenda state.

Maintaining this information is a necessary first step for providing explanation to help with debugging CLIPS, since CLIPS, being a forward-chaining language, requires temporal information to determine why a program behaved as it did. Such explanation will reduce the tedium for system builders, reducing the time they must spend examining system traces or single-stepping through a program to find out if or when something occurred.

Future work will include actually implementing these ideas, to see if integrating this information into the Rete network works as well in practice as it appears that it should in principle. Such investigation may also reveal ways to reduce the space needed to store historical data. Then, we will build the question-answering system described, on the top-level of CLIPS. We will determine what types of questions are useful for debugging CLIPS programs, and will design a system that can answer such questions, using the historical information stored in the historical Rete network and associated data structures. Empirical experiments will also be needed for evaluating the effectiveness of the resulting explanations in helping with debugging. Both debugging and testing require knowledge of what has occurred during a program run — therefore, additional future work may also include using this stored historical information in the development of further testing and debugging tools for CLIPS programs. As the size of CLIPS programs increases, such software engineering tools will become more and more necessary. These methods for maintaining and storing run history, and the explanation that they will facilitate, should help in developing these future CLIPS programs.

REFERENCES

Barker, V. E. and O'Connor, D. E. (1989). Expert systems for configuration at Digital: XCON and beyond, *CACM*, Vol. 32, No. 3 (March), pp. 298-318.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5*, Addison-Wesley.

COSMIC (1989). *CLIPS Reference Manual*, Version 4.3, Artificial Intelligence Section, Lyndon B. Johnson Space Center, COSMIC, 382 E. Broad St., Athens, GA., 30602.

Domingue, J. and Eisenstadt, M. (1989). A new metaphor for the graphical explanation of forward-chaining rule execution, *Proceedings of IJCAI '89*, Detroit, Michigan, pp. 129-134.

Eick, C. F. (1991) Design and implementation of a rule-based forward chaining language that supports variables, encapsulation, and operations, submitted for publication to *IEEE Transactions on Knowledge and Data Engineering*, February 1991.

Eick, C. F., Yao, C., and Fu, H. (1989). More flexible use of variables in rule-based programming, *Proceedings of the 2nd Int. Symposium on Artificial Intelligence*, Monterrey, Mexico.

Forgy, C. L. (1982). Rete: a fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence*, Vol. 19 (September), pp. 17-37.

Giarratano, J. (1989). *CLIPS User's Guide*, Version 4.3, Artificial Intelligence Section, Lyndon B. Johnson Space Center, COSMIC, 382 E. Broad St., Athens, GA., 30602.

Gupta, A., Forgy, C., Kalp, D., Newell, A., and Tambe, M. (1988). Parallel OPS5 on the Encore Multimax, *IEEE International Conference on Parallel Processing*, pp. 271-280.

Gupta, A. (1987). *Parallelism in Production Systems*, Morgan Kaufmann.

IEEE (1989). *IEEE's Software Engineering Standards*, IEEE.

Jacob, R. and Froscher, J. (1990). A software engineering methodology for rule-based systems, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 2 (June), pp. 173-189.

Scales, D. (1986). Efficient matching algorithms for the SOAR/OPS5 production system, Knowledge Systems Laboratory, Stanford University, Report No. KSL 86-47 (June).

N92-16592

CLIPS APPLICATION USER INTERFACE FOR THE PC

Jim Jenkins, Rebecca Holbrook, Mark Shewhart, Joey Crouse, and Capt Stuart Yarost

Air Force Logistics Command (AFLC)
Acquisition Logistics Division, Joint Technology Applications Office (ALD/JTI)
Artificial Intelligence Support Center (AISC)
Wright Patterson Air Force Base, Ohio 45433

Abstract. The majority of applications that utilize expert system development programs for their knowledge representation and inferencing capability require some form of interface with the end user. This interface is more than likely an interaction through the computer screen. When building an application the user interface can prove to be the most difficult and time consuming aspect to program. Commercial products currently exist which address this issue. To keep pace CLIPS will need to find a solution for their lack of an easy-to-use Application User Interface (AUI). This paper represents a survey of the DoD CLIPS' user community and provides the backbone of a possible solution.

INTRODUCTION

A transparent, easy to use, and more visually pleasing end-user interface for applications developed in CLIPS is needed. Bernard Engel of Purdue University clearly presented this point in the paper he gave at the First CLIPS Conference Proceedings concerning their in-house expert system development. Reference [1]. In addition, McDonnell Douglas' MCAIR Artificial Intelligence Center is developing a similar product. Reference [2]. At MCAIR, they developed the CLIPS Advanced Development Enhancement (CADE) as part of the Technical Expert Aircraft Maintenance System (TEAMS) project. At the same time we were developing our own enhancements to CLIPS and researched the above contacts concerning the simililarities and differences. Given this activity and our own in-house requirement for a more palatable user interface, we established the following initial requirements for a CLIPS AUI:

(1) The interface should consist of <u>simple</u> function calls from within CLIPS.

(2) The interface should not over burden the CLIPS applications programmer with excessive control of the interface screens.

(3) The interface must provide easy to use window-like Input and Output functions with the following minimal capabilities:

(a) Display blocks of text.

(b) Allow users to input single or multiple responses to multiple choice questions via arrow keys.

(c) Obtain user input through the keyboard.

(4) The text, questions, and menu options associated with an application can be read from a flat ASCII text file. That is, the text is neither compiled nor entered as part of the CLIPS knowledge base. The CLIPS applications programmer must have the ability to use multiple ASCII files for this purpose.

The prototype of the CLIPS AUI developed at Purdue University meets requirements 1 throug 3 above. However, the Purdue implementation places all the text within the CLIPS knowledg base. We found that separating the text from the knowledge base allows easy review/editin by the expert and improves memory management. The product developed by MCAIR meet all of the above requirements and includes some additional features such as rule-chaining an a pseudo-nonmonotonic reasoning capability. Our own in-house product, INTCLIPS, als meets all of the above requirements and was developed for use on current and future projects All these efforts address the same AUI issues and approached the problem in similar ways. T avoid further duplication of effort, and to support the CLIPS community, we present this paper

This paper provides a technical survey of the user requirements for a more effective applicatio user interface to CLIPS. To begin with, a detailed description of the AUI prototype INTCLIPS, is presented. Second, an outline of the AUI customer feedback is given followe by the technical issues concerning the implementation issues of the feedback. Finally, a surve is presented of the commercial products which may provide an alternative solution for buildin the end-user interface.

AUI PROTOTYPE

We have developed a set of AUI functions that can be embedded within the CLIPS source code. These functions provide the developer of an expert system application with the basics require for an AUI, thereby facilitating the process of rapid prototyping and development. We designed this set of functions to provide a friendly window-like environment for the end user. The basi of the AUI is the capability to retrieve data from either a standard ASCII text file or th keyboard and the capability to display data to the terminal.

The function **write_values** is used to access and display the contents of the facts list. The following will describe in detail the utility of each function.

Init_clips_display

The user defined function, init_clips_display, is used to initialize each paragraph and question display routine. The function call is required to initialize the pointers to the flat ASCII files which contain the questions and text. See **Table 1** for an example.

Usage : (**init_clips_display** *arg_1 arg_2 arg_3*)

arg_1 : Name of the flat ASCII file that contains the questions or text. Arg_1 may not be omitted.

arg_2 : Name of the file that will be created to hold the index information for the fla ASCII file named in arg_1. Arg_2 may not be omitted.

arg_3 : Either "paragraph" or "question" depending upon the contents of arg_1. ARG_3 may not be omitted.

```
(defrule Initialize
   (initial-fact)
=>
   (init_clips_display "pgh.txt" "pgh.idx" "paragraph")
   (init_clips_display "ques.txt" "ques.idx" "question")
   (banner "on" "Engine Diagnostic Expert System")
   (assert (ready)))
```

Table 1. Example of Rule Using Init_clips_display and Banner

Clear_screen

The user defined function, clear_screen, is used to clear the contents of the video screen. See **Table 2** for an example.

Usage : (clear_screen)

Banner

The user defined function, banner, is used for aesthetic purposes to add a title or banner to the expert system screen. See **Table 1** for an example.

Usage : (**banner** *arg_1 arg_2*)

arg_1 : The argument takes on two values either "on" or "off". Arg_1 may not be omitted.

arg_2 : The title which is to be placed within the banner at the top of the screen (ie. "My program title"). Arg_2 may be omitted.

Write_paragraph

The user defined output function, write_paragraph, is used to display a paragraph of text within a specified paragraph block. This function uses formatted or unformatted paragraphs and displays a block of text associated with a key-word on the screen in a window-like format. See **Table 2** for an example.

Usage: (**write_paragraph** *arg_1 arg_2*)

arg_1 : Name of the file that holds the index information for the flat ASCII file which contains the textual information associated with the key-word in arg_2. Arg_1 may not be omitted.

arg_2 : The key-word or "paragraph name" which identifies the block of text to be retrieved. Arg_2 may not be omitted.

```
(defrule print-repair ""
  (list-repairs)
  (name name)
   =>
  (clear_screen)
  (write_values "pgh.idx" "repair_title" "repair" ?name)
  (clear_screen)
  (write_paragraph "pgh.idx" "final_statement" ?name)
  (system "cls")
  (exit))
```

Table 2. Example of Rule Using Clear_screen, Write_paragraph, and Write_values

Ask_menu_question

The user defined input function, ask_menu_question, is used when a question is asked and multiple responses are presented in a menu selection block. This function displays the question text and list of possible answers for the question associated with a key-word. The user selects an answer using the arrow keys. Single and multiple answers are both permitted. See Table 3 for an example.

Usage: (**ask_menu_question** *arg_1 arg_2 arg_3 arg_4*)

arg_1 : Name of the file that holds the index information for the flat ASCII file which contains the textual information associated with the key-word in arg_2. Arg_1 may not be omitted.

arg_2 : The key-word or "question name" which identifies the block of text to be retrieved for the question. Arg_2 may not be omitted.

arg_3 : A string that is to be associated with the selected answer(s) in the CLIPS fact list. Arg_3 may not be omitted.

arg_4 : Either "single" or "multiple". "Multiple" allows one or more answers to be selected from the menu. If arg_4 is omitted, the default is "single".

```
(defrule determine-engine-state ""
   ?rem <- (query phase)
   (not (working-state engine ?))
   =>
   (retract ?rem)
   (clear_screen)
   (ask_menu_question "ques.idx" "determine_engine_state"
                                   "working-state engine"))
```

Table 3. Example of Rule Using Ask_menu_question

Typed_in_question

The user defined input function, typed_in_question, is used when there is a need for input from the keyboard. The function displays the text of a question associated with a key-word and the user enters the answer from the keyboard. Only single answers are permitted. See Table 4 for an example.

Usage : **(typed_in_question** *arg_1 arg_2 arg_3 arg_4*)

arg_1 : Name of the file that holds the index information for the flat ASCII file which contains the textual information associated with the key-word in arg_2. Arg_1 may not be omitted.

arg_2 : The key-word or "paragraph name" which identifies the block of text to be retrieved for the question. Arg_2 may not be omitted.

arg_3 : A string that is to be associated with the selected answer(s) in the CLIPS fact list. Arg_3 may not be omitted.

arg_4 : Either "int", "float", "string", or "word". This indicates the type of answer expected. Type checking is done. If the type check fails, the user is prompted to re-enter the response. If arg_4 is omitted, the default is "string".

```
(defrule get-user-name
   (need name)
   =>
   (clear_screen)
   (typed_in_question "pgh.idx" "get-name" "name"))
```

Table 4. Example of Rule Using Typed_in_question

Write_values

The user defined function, write_values, displays the contents of the CLIPS fact list whose firs entry matches a particular attribute name. Above this attribute list a text block will display th text associated with the key-word found in the question file.

This function can include the values of CLIPS variables (string or float) within the body o a paragraph. This format option requires the inclusion of a corresponding number of additiona arguments within the function call ,write_values, to provide the ability to create formatte paragraphs. See **Table 2** for an example. **Table 5** illustrates an ASCII file that contains a format character, %s, therefore the function call to write_values would require one additiona argument in order for this format character to be utilized.

Usage : (**write_values** *arg_1 arg_2 arg_3 arg_n*)

arg_1 : Name of the file that holds the index information for the flat ASCII file whicl contains the textual information associated with the key-word in arg_2. Arg_1 may not be omitted.

arg_2 : The key-word or "paragraph name" which identifies the block of text to be retrieved. Arg_2 may not be omitted.

arg_3 : A string that represents an attribute name. Each element in the CLIPS fact list tha has arg_3 as its first element is displayed without that element.

arg_4 : The variable name, **?var**, which corresponds to a valid variable name defined withi the INTCLIPS knowledge base. Arg_4 through arg_n may be omitted.

arg_n : The variable name, **?var**, which corresponds to a valid variable name defined withi the INTCLIPS knowledge base. Arg_4 through arg_n may be omitted.

------ COMMENTS -------------	----- ASCII FILE -----------
Delimiter between text blocks	@
Identifier for text block	repair_title
Line #1 with format character	The repairs for %s are listed below.
Delimiter between text blocks	@

Table 5. Sample ASCII File with the Format Character, %s, Included

ASCII File Format

Table 6 and Table 7 provide examples of the INTCLIPS ASCII text files that would be used for a typical application. The format of a paragraph file is different than the format of a question file. Separating the two file formats was implemented in order to provide a more organized method of developing the knowledge base.

------ *COMMENTS* -------------	----- ASCII FILE -----------
Delimiter between text blocks	@
Identifier for text block	system_banner
line #1 for text block	The Engine Diagnostic Expert System
Delimiter between text blocks	@
Identifier for text block	ask-users-name
line #1 for text block	What is your name?
Delimiter between text blocks	@
Identifier for text block	suggested-repairs
line #1 for text block	The Engine Diagnostic Expert System
line #2 for text block	is now complete.
Delimiter between text blocks	@

TABLE 6. Sample ASCII File for Paragraph Text

---------- *COMMENTS* -------------	----- ASCII FILE -------
Delimiter between question blocks	@
Identifier for question block	determine_engine_state
Number of responses in menu	3
Response #1	normal
Response #2	unsatisfactory
Response #3	does-not-start
Line #1 for question text	What is the working state of the
Line #2 for question text	engine?
Delimiter for explanatory text	#
Line #1 of explanatory text	-- Optional Explanatory Text --
Delimiter between question blocks	@
Identifier for question block	determine_rotation_state
Number of responses in menu	2
Response #1	yes
Response #2	no
Line #1 for question text	Does the engine rotate?
Delimiter for explanatory text	#
Delimiter between question blocks	@
Identifier for question block	determine_sluggishness
Number of responses in menu	2
Response #1	yes
Response #2	no
Line #1 for question text	Is the engine sluggish?
Delimiter for explanatory text	#
Delimiter between question blocks	@

TABLE 7. Sample ASCII File for Questions

AUI CUSTOMER FEEDBACK

The CLIPS Help Desk provided the names of DoD CLIPS' users. Users requested a total of 83 copies of INTCLIPS for review. As a result of the accompanying INTCLIPS survey (see Attachment), we defined the following additional requirements for the CLIPS AUI.

(a) Allow retrieval of answers from a pre-processed ASCII file.

(b) Add the ability to size and move windows around.

(c) Make tool portable across multiple platforms.

(d) Adapt AUI to CLIPS version 5.0.

(e) Make CLIPS AUI MS-windows 3.0 compatible.

(f) Add mouse support.

(g) Add hypertext capability.

(h) Add on-screen form fill capability.

(i) Allow multiple text windows per screen.

(j) Allow scrollable text.

(k) Add graphics import capability.

(l) Add sufficient error trapping.

TECHNICAL ISSUES

We developed the AUI functions using Turbo C and embedded them within CLIPS as user defined functions that we then recompiled and renamed as INTCLIPS. The development of an expert system from within CLIPS and INTCLIPS is identical except that the knowledge base developed from within INTCLIPS now can call upon the user defined functions.

The AUI consists of a keyword based text retrieval system and a character based windowing and menu system - both of which are very basic in their underlying concepts. Certainly the functionality in our AUI prototype is basic. Our customer feedback indicates that the CLIPS user community is interested in more functionality than our AUI currently provides. Each suggested enhancement has direct impact on both the AUI developer as well as the CLIPS developer who will use the CLIPS AUI. Based upon our experience, we address the impact of several suggested AUI improvements below.

Portability of AUI Code

The only compiler specific code in the AUI involves the functions used to manipulate the screen. Generic screen manipulation functions could be used in the AUI code and would depend on designated compiler options set by the user. Such changes would have no impact upon the CLIPS application developer. Development of this capability would require familiarity with a variety of compilers and their screen manipulation or graphics functions.

Graphical Based Interface

We can address this issue the same way the portability of AUI code is addressed above. Such a change would have no impact upon the CLIPS application developer. Due to the finer resolution of the graphics screens, the construction of screens may be more tedious.

Mouse Support

Basic mouse support that replicates the functions of the arrow keys is a simple process to implement. On the other hand, advanced mouse support that allows window resizing and scrolling of text would be a more complicated matter.

CLIPS 5.0 Upgrade

Only slight modifications would be required for this implementation since only the names of a few external function calls have been modified for CLIPS version 5.0.

Ability To Move Windows

Many CLIPS AUI users would like the ability to place the windows of text anywhere on the screen as opposed to the default centered window. We can accomplish this feature by adding arguments to the CLIPS functions to indicate the location of the window. This feature would provide no major complications for the AUI developer.

Multiple Window Overlays

This capability is similar to the ability to move the location of a window. A toggle key must be present for switching back and forth from one window to the next. Mouse interaction also could accomplish toggling from one window to the next .

Scrollable Windows

Experience and customer feedback indicated that scrollable windows are a basic requirement that any AUI should meet. Scrollable windows would add no complications for the CLIPS application developer, but implementing this feature would add a bit more complexity to the AUI code.

Ability To Size Windows

An additional screen control feature that some CLIPS AUI users have requested is the ability to size windows. While on the surface this looks comparable to adding the ability to move the window about the screen, there are some fundamental issues involved when choosing to implement this option. The underlying text retrieval system implemented in the AUI prototype is WYSIWYG - What You See Is What You Get. That is, the text is displayed on the screen as it appears in the ASCII file. The implementation sizable windows would require manipulation of the text in a wrap around fashion and the use of scrollable windows. The only advantage this would provide is the ability to display the same paragraph in various sized windows.

Hypertext Capability

One key motivation behind the development of the AUI prototype was the large number of expert system applications that are "information intensive." Many applications simply ask a few questions and then display large amounts of information. The most basic method of displaying

information is the block of text in a window as implemented with the INTCLIPS function write_paragraph. Hypertext offers a more sophisticated method of information display. A very basic hypertext feature can be implemented using the existing AUI keyword text retrieval functions. In fact, the implementation would involve simple modifications to the existing write_paragraph function. Words surrounded by special symbols could be placed in the ASCII text files to be used as links to other blocks of hypertext. These words would simply be keywords to other blocks of text in that ASCII text file. In this way, the function implementing write_paragraph would be recursively called each time the user selected a highlighted "hot link" in the displayed text.

Form Fill Capability

We could develop a form fill capability with the same technique of utilizing a flat ASCII file and special characters to identify the various input sections of the form. The development could become somewhat complicated due to the variability and number of parameters that it may require to use.

Ability To Import Graphics

Some applications may require the ability to display graphic images. One such application that is being developed by the Department of the Army's Environmental Lab. Reference [3]. There project consists mainly of graphic presentations and therefore user defined graphic display functions were recompiled into an enhanced version of CLIPS named GCLIPS. The most direct way to implement this is with some of the freeware applications that can display graphic images saved in an appropriate file format. Since (1) there are many such freeware and commercial products available, (2) the implementation would involve simply a system call, and (3) CLIPS currently supports system calls, therefore by default this feature is available in the AUI prototype.

MS-Windows Interface

Several customers indicated a strong desire for MS-Windows compatibility. While MS-Windows compatibility has advantages, there are three major drawbacks to implementing such a feature: (1) A large amount of time and effort would be required, (2) the entire CLIPS application would probably have to be modified, and (3) potential users would be limited to MS-Windows users.

COMMERCIAL PRODUCTS

User Interface Management Systems

The following is a sampling of the companies that provide user interface management systems. These products are tools that can be used to develop a CLIPS AUI but involve integration issues and runtime costs. They offer a variety of functions for developing windows, menus, data entry screens, importing graphics, help functions, mouse support, scrollable windows, text editors, key input validation, etc. In addition, many of these products offer interactive screen designers and code generators. These products also support libraries of functions over a wide variety of platforms. References [4,5].

Creative Programming Consultants Inc.
Box 112097
Carrollton, Texas 75011
(214)416-6447

Product: VitaminC 3.2 and 4.0 with VCscreen 3.2

Vermont Creative Software
Pinnacle Meadows
Richford, VT 05476
1-800-848-1248
Product: Vermont Views with Designer v2.0 and GraphEx

Solution Systems
372 Washington Street
Wellesley, MA 02181
1-800-677-0001
Product: C-Worthy 2.0

Copia International LTD.
Roundhill Computer Systems
1342 Avalon Court
Wheaton, IL 60187
(706)682-8898
Product: Panel Plus II

Oakland Group Inc.
(Subsidiary of Liant Software Corp.)
Cambridge, Mass
1-800-233-3733
(617)491-7311
Product: C-scape 3.2 with Look & Feel 3.2

Expert System Development Environments

The following is a sampling of some of the companies that provide expert system development environments that assist with the development of the AUI.

Information Builders, Inc.
503 Fifth Avenue
Indialantic, Florida 32903
1-800-444-4303
Product: Level 5

Neuron Data Systems
156 University Avenue
Palo Alto, CA 94301
1-800-876-4900
Product: Nexpert Object

AI Corp, Inc.
1700 Rockville Pike
Suite 400
Rockville, MD 20852
(301)881-8100
Product: KBMS and 1st Class

CONCLUSION

After completing a comprehensive survey of the DoD CLIPS' user community we have found that there is a great need to enhance CLIPS with an AUI development tool. We recommend the results of this survey be utilized as a basis for such a tool development, and the next version of CLIPS should carefully weigh the importance of implementing such a tool. The user defined functions that we offer to share with the CLIPS community unfortunately, provide only the first step in meeting the customer needs. The more important contribution is the user contacts and the corresponding feedback acquired. Copies of INTCLIPS and reprints of this paper are available to government agencies upon request.

REFERENCES

[1] Engel, Bernard A., et al, "CLIPS Interface Development Tools and Their Applications", First CLIPS Conference Proceedings, Vol II, August 1990, p. 458 - 469.

[2] Blankenship, Keith and Reichart, Rick, McDonnell Douglas, MCAIR Artificial Intelligence Center, Mail Code: 1065205.

[3] Smith, Craig, Research Biologist, Department of the Army, Environmental Laboratory, Waterways Experiment Station, Corps of Engineers, 3909 Halls ferry Road, Vicksburg, Mississippi 39180-6199.

[4] Mirecki, Ted, "Interface Tools Smooth Transition to OS/2", PC Week Reviews, May 20, 1991, p. 125 - 129.

[5] Robie, Jonathan, "Three C Language Screen-Utility Packages for PCs", Byte, October 1987, p. 223 - 229.

CLIPS Application User Interface Feedback Questionnaire

ALD/JTI
ATTN: Rebecca Holbrook
Wright-Patterson AFB, OH 45433
DSN: 785-3303 COMM: (513) 255-3303

All CLIPS Application User Interface (AUI) testers/users:

Please complete this feedback questionnaire to provide us with an opportunity to make CLIPS AUI a better product. We hope to compile the survey results in July, so please return this questionnaire to the address above by **30 June 1991.**

NAME: ____________________ DATE: __________

ADDRESS: ____________________ COMM: ( ) __________

____________________ DSN: __________

MY JOB IS:
_ Engineer Scientist
_ System Manager
_ Maint Technician
_ Equipment Specialist
_ Item Manager
_ Logistician
_ Senior Mngt
_ Middle Mngt
_ Other:__________

PLEASE ANSWER THE FOLLOWING QUESTIONS. CIRCLE ALL APPROPRIATE ANSWERS WHERE APPLICABLE, ADDING YOUR COMMENTS WHENEVER POSSIBLE.

GENERAL:

1-1. Were you able to test the CLIPS AUI software as planned? If not, please explain.

1-2. What computer configuration did you use?

CPU:	Z-248	Desktop III	Other:________
monitor:	EGA	VGA	Other:________
mouse:	Yes	No	

1-3. What are your overall impressions of CLIPS AUI?

1-4. Specifically, what did you like?

1-5. Specifically, what did you dislike?

1-6. Of what relative value did you find the following:
(Range from 1 (no value) to 10 (highest value), 0 - did not use) Comments

_ CLIPS AUI User's Manual

_ CLIPS AUI Application Color Custimization Program (MODCFG.EXE)

_ CLIPS AUI Source Code

CLIPS AUI SPECIFIC:

2-1. Of what relative value did you find the following:
[Range from 1 (no value) to 10 (highest value), 0 - did not use]

Comments

_ CLIPS AUI ***clear_screen*** function

_ CLIPS AUI ***write_paragraph*** function

_ CLIPS AUI ***ask_menu_question*** function

_ CLIPS AUI ***typed_in_question*** function

_ CLIPS AUI ***write_values*** function

_ CLIPS AUI ***banner*** function

2-2 Do you have any recommendations for additional CLIPS AUI functions?

2-3 What level of expertise do you have as a CLIPS programmer?

None Novice Journeyman Expert

2-4 If you were planning to develop an expert system in CLIPS, would you use CLIPS AUI?

yes no unsure

2-5 Do you intend to make run-time versions of your CLIPS applications?

yes no unsure

N92-16593

EXPERT NETWORKS IN CLIPS[1]

S.I. Hruska, A. Dalke, J.J. Ferguson, and R.C. Lacher

Department of Computer Science
Florida State University, Tallahassee

Abstract. Rule-based expert systems may be structurally and functionally mapped onto a special class of neural networks called *expert networks*. This mapping lends itself to adaptation of connectionist learning strategies for the expert networks. Following a process introduced by Kuncicky, Hruska, and Lacher, a parsing algorithm to translate CLIPS rules into a network of interconnected assertion and operation nodes has been developed. The translation of CLIPS rules to an expert network and back again is illustrated. Measures of uncertainty similar to those used in MYCIN-like systems are introduced into the CLIPS system and techniques for combining and firing nodes in the network based on rule-firing with these certainty factors in the expert system are presented. Several learning algorithms are under study which automate the process of attaching certainty factors to rules.

EXPERT NETWORKS

The idea of mapping an expert system onto a neural network in order to make the most of what each technology offers is a timely one [Fu and Fu 1990][Gallant 1988] [Hall and Romaniuk 1990] [Kuncicky 1990] [Kuncicky et al. 1991] [Lacher et al. 1991a] [Towell et al.1990]. Hruska, Kuncicky, and Lacher define the special class of networks created from an expert system as *expert networks*. In these systems, rules of the form *if A then B* where *A* and *B* are assertions are mapped in an intuitive manner to nodes labelled *A* and *B* with a forward connection tying the two together. In expert systems which incorporate uncertainty, the certainty factor, a value between -1 and $+1$, is mapped to the connection strength in the network.

The functioning of the network is also based on the expert system, with the combining and firing functions of the network nodes derived from the inference engine functions for the expert system. Inference engine functions which have counterparts in the network include those for combining evidence for rules that have consequents in common, methods of handling conjunction and negation in rules, thresholding functions, and those functions that govern firing of rules. The nodes are not simple perceptron-type nodes, but rather are more complex, based on how the expert system works internally. When activated with input, an expert network functions identically to the expert system inference engine operating on the rule base from which the network was derived.

One of the most compelling reasons for pursuing the mapping of expert system concepts onto neural networks is the possibility for introducing connectionist learning techniques into traditional AI systems. Automation of even part of the expensive and time-consuming knowledge acquisition process is a valuable contribution of this technology.

---

[1] Research partially supported by the Florida High Technology and Industry Council, the US Office of Naval Research, and Oak Ridge Associated Universities.

There are several levels of knowledge acquisition for an expert system [Hruska et al. 1991a][Hruska et al. 1991b]. These include refining or adjusting the certainty factors for rules, forming new rules from existing concepts, and forming new concepts. Ideally, most of the knowledge acquisition process could be done directly from data. This training data consists of examples of input information and the associated desired (correct) output. Bypassing the knowledge elicitation process in which a human expert must express their expertise in the form of if-then rules is an ultimate goal. Automation would be useful at any of the levels listed above.

The current research efforts reported in this paper focus on the area of automating the refinement of knowledge. Certainty factors represent perhaps the most subtle form of knowledge in a system, the confidence which the expert has in the consequent being related to the antecedent of the rule. In decision-making, the use of certainty factors enables the system to rank outcomes of the system based on the certainty factors attached to them. In CLIPS, we are in the process of implementing certainty factors via the *declare* option available in CLIPS rule formats.

THE TRANSLATION PROCESS

In order to apply the connectionist learning techniques under development, it is useful to translate the expert system rule base to expert network format. The process of translating an expert system rule base to the type of expert network described above was prototyped by Kuncicky and colleagues [Kuncicky 1990] [Kuncicky et al. 1991] for the expert system M.1 [M1 1986]. Following these ideas, an implementation of the translation process was written for a subset of CLIPS. Preliminary work on this project is summarized in [Johnson and Franke 1989].

The translation is performed in two steps. First, the rule base is simplified somewhat by replacing rules of designated complexity (such as those with disjunctions in their antecedents) with several simpler but collectively equivalent rules. For example, rules (with cf certainty factors) of the form

if A or B then C (cf)

are simplified to

if A then C (cf)

if B then C (cf).

Another type of rule base simplification is reduction of the rule

if A and (B or C) then D (cf)

to

if A and B then D (cf)

if A and C then D (cf)

which distributes conjunction over disjunction. Rules of the form

if not(A or B) then C (cf)

are simplified via DeMorgan's law to

if not A and not B then C (cf).

Simplification of conjunction in the consequent converts rules of the form

if A then B and C (cf)

to the equivalent set of rules

if A then B (cf)

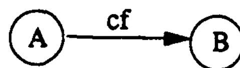
if A then C (cf).

Through this process, disjunctions in the antecedents and conjunctions in the consequents are eliminated. Disjunctions in the consequents are not allowed.

After the rule base simplification transformations are complete, the rule base is translated into an expert network. At this point, there are three basic forms of rules: regular rules, rules with conjunctions in the antecedent, and rules with negations in the antecedent. These three types of rules may be compounded to form rules of arbitrary complexity in the antecedent. a regular rule, of the form

if A then B (cf)

is converted to a portion of an expert network as

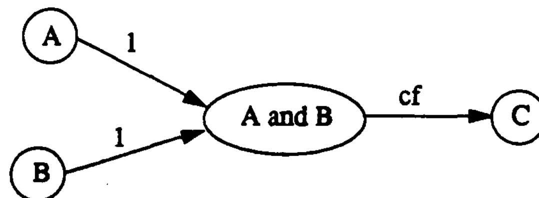


Rules with conjunctions and negations involve creation of special operator type nodes. Connections from assertions to these operator nodes are fixed with a weight of 1.0.

Typically, a rule involving a conjunction such as

if A and B then C (cf)

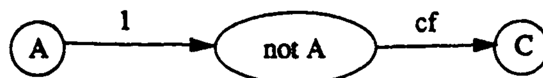
is converted to



while rules with negations of the form

if not A then B (cf)

become



A short example of the translations process in CLIPS is shown in Tables 1 and 2 and Figure 1.

As of this date, the translation program handles input rule bases which contain defrules, deffacts, and asserts. Plans are now underway to extend the flexibility of this prototype to accept variables, pattern matching, and retraction.

```
(defrule clear
(declare (certainty 0.96))
; If the barometer is high and there are no clouds then it is clear.
     (and (high)
          (not (cloudy)))
=>
     (printout t "Clear" crlf)
)
(defrule sky-water
(declare (certainty 0.7))
; If the barometer is low or it is cloudy then it is rainy.
     (or (not (high))
         (cloudy))
=>
     (assert (precipitation))
)
(defrule snowy
(declare (certainty 0.8))
; If there is precipitation but it is not hot then time for snow.
     (and (precipitation)
          (not (hot)))
=>
     (printout t "Snow!" crlf)
)
(defrule rainy
(declare (certainty 0.9))
; If there is precipitation and it is hot then it must be rainy.
     (and (precipitation)
          (hot))
=>
     (printout t "Get your umbrellas." crlf)
)
```

Table 1. Original Rule Base

```
(defrule clear-1
(declare (certainty 0.96))
; From rule number 1
     (high)
     (not (cloudy))
=>
     (printout t "Clear" crlf)
)
(defrule sky-water-2
(declare (certainty 0.7))
; From rule number 2
     (not (high))
=>
     (assert (precipitation))
)
(defrule sky-water-3
(declare (certainty 0.7))
; From rule number 2
     (cloudy)
=>
     (assert (precipitation))
)
(defrule snowy-4
(declare (certainty 0.8))
; From rule number 3
     (precipitation)
     (not (hot))
=>
     (printout t "Snow!" crlf)
)
(defrule rainy-5
(declare (certainty 0.9))
; From rule number 4
     (precipitation)
     (hot)
=>
     (printout t "Get your umbrellas." crlf)
)
```

Table 2. After Rule Base Transformations

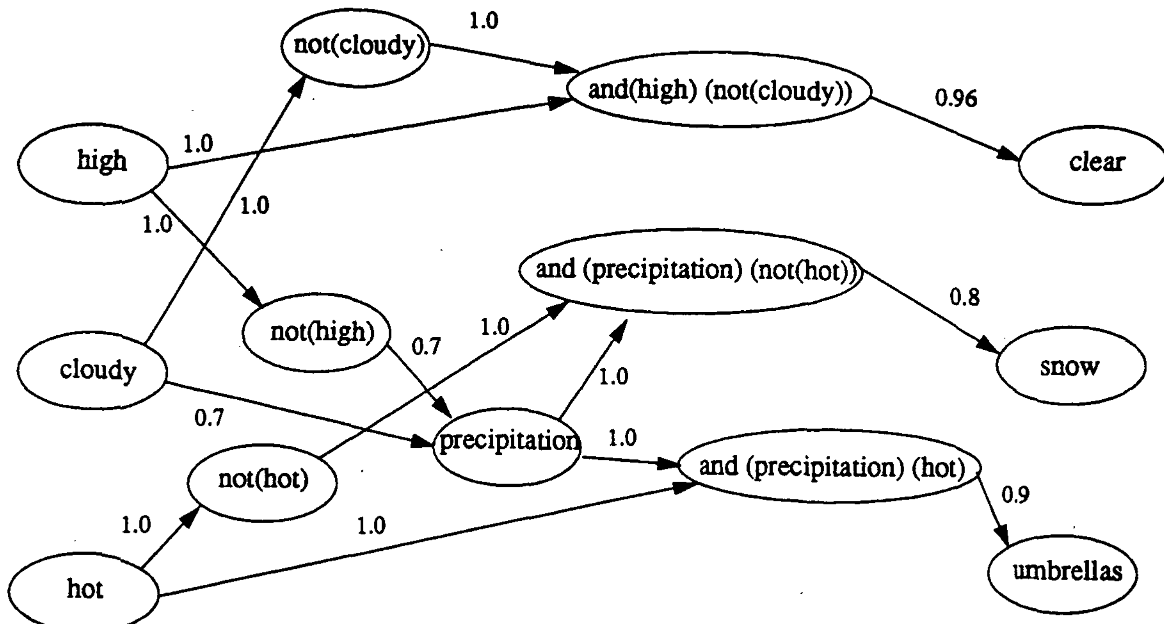


Figure 1. Expert Network From Rule Base

PROCESSING CERTAINTY FACTORS IN CLIPS

The study of M.1 and its manner of handling certainty factors has led us to contemplation of improvements which could be made in porting this process to CLIPS. Our introduction of certainty factors into CLIPS rule bases led to the necessity of altering the CLIPS source code to implement the processing of certainty factor information. This includes a set of algorithms for evidence accumulation, combining evidence, outputting a certainty factor value for the rule, and firing the rule based on that certainty factor output. One possibility, currently under consideration by Lacher and Traphan [Traphan 1991] is to build and improve on the inference engine for the expert system shell EMYCIN (upon which M.1 is also based). EMYCIN uses a threshold of 0.2 to determine whether a rule will fire or not. The proposal is to smooth this thresholding function, resulting in a system which yields analog values very close to those of the original EMYCIN system, but which is continuously differentiable. This feature will enable gradient descent learning techniques to be applied in a more consistent manner over the entire range of values. This work is currently in progress.

Learning techniques developed for expert networks include Goal-Directed Monte Carlo Search [Hruska et al. 1991b] and Expert System Backpropagation [Lacher et al. 1991b]. The first of these is designed using the principles of reinforcement learning with increasing levels of noise applied to the connections between nodes. The second is an adaptation of the standard backpropagation of error learning algorithm which uses gradient descent to find the weight settings (certainty factors) which minimize the difference between the system's output and the ideal (correct) output. The major innovation of Expert System Backpropagation is the use of the expert system's inference engine functions in the complex nodes of an expert network to perform the computations necessary for

gradient descent. Both of these learning algorithms have been tested on the M.1 system and are currently being upgraded to work with the altered version of CLIPS described above.

CONCLUSION

A system for translating a subset of CLIPS expert system rule bases into expert networks has been prototyped and is under ongoing development. CLIPS rules will have certainty factors attached to them which may be used to express uncertainty in the inferencing process. A proposal for smoothing the evidenciary combining and firing functions of traditional EMYCIN-like systems is described. Learning algorithms for expert networks are briefly described. As work progresses on this system, we draw nearer to realization of a tool for automating the knowledge acquisition process involved in traditional AI systems.

REFERENCES

Fu, L.M. and L.C. Fu (1990). Mapping rule-based systems into neural architecture, *Knowledge-Based Systems*, Vol 3, No 1.

Gallant, S.I. (1988). Connectionist expert systems, *Communications ACM,* **24**, 152-169.

Hall, L.O. and S.G. Romaniuk (1990). FUZZNET: Toward a fuzzy connectionist expert system development tool, *Proceedings IJCNN 90* (Washington, DC), vol. II, 483-486.

Hruska, S.I., D.C. Kuncicky, and R.C. Lacher (1991a). Learning in acyclic expert networks, *Proceedings of WNN-AIND 91*, February 1991.

Hruska, S.I., D.C. Kuncicky, and R.C. Lacher (1991b). Hybrid learning in expert networks, *Proceedings of IJCNN 91, Seattle*, July 1991.

Johnson, J. and J. Franke (1989). Design specification for CLIPS parser (CP ver. 0.5B), Undergraduate Honors Project for S.I. Hruska, Florida State University, Fall, 1989.

Kuncicky, D.C., S.I. Hruska, and R.C. Lacher (1991). Hybrid systems: The equivalence of rule-based expert system and artificial neural network inference, *International Journal for Expert Systems,* accepted with revisions, May 1991.

Kuncicky, D.C. (1990). Isomorphism of reasoning systems with applications to autonomous knowledge acquisition, PhD dissertation, (R.C. Lacher, major professor), Florida State University, December, 1990.

Lacher, R.C., S.I. Hruska, and D.C. Kuncicky (1991a). Expert networks: A Neural Network Connection to Symbolic Reasoning Systems, *Proceedings of FLAIRS-91,* April 1991.

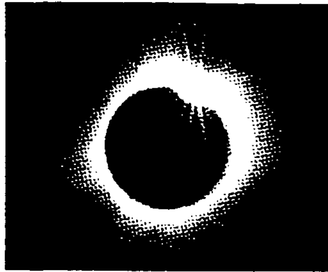
Lacher, R.C., S.I. Hruska, and D.C. Kuncicky (1991b). Backpropagation learning in expert networks, *IEEE Transactions on Neural Networks,* accepted with revisions, May 1991.

M.1 Reference Manual (1986). (Software version 2.1), Teknowledge, Palo Alto, CA.

Towell, G.G., J.W. Shavlik, and M.O. Noordewier (1990). Refinement of approximate domain theories by knowledge-based neural networks, *Proceedings of AIII-90, Eighth National Conference on Artificial Intelligence,* July 1990.

Traphan, B. and R.C. Lacher (1991). Smoothing EMYCIN for backprop learning, work in progress, June 1991.

N92-16594



ECLIPS: An Extended CLIPS For Backward Chaining and Goal-Directed Reasoning

Peter V. Homeier and Thach C. Le

Information Technology Department
The Aerospace Corporation

Abstract. Realistic production systems require an integrated combination of forward and backward reasoning to reflect appropriately the processes of natural human expert reasoning. A control mechanism that consists solely of forward reasoning is not an effective way to promptly focus the system's attention as calculation proceeds. Very often expert system programmers will attempt to compensate for this lack by using data to enforce the desired goal-directed control structure. This approach is inherently flawed in that it is attempting to use data to fulfil the role of control. This paper will describe our implementation of backward chaining in CLIPS, and show how this has shortened and simplified various CLIPS programs. This work was done at the Aerospace Corporation, and has general applicability.

1. DESIGN CONSIDERATIONS

The Aerospace Corporation has been using expert system technology since the mid-1980s, beginning with a system to diagnose anomalies in the attitude control system of the DSCS III satellite. These experiments showed the value of expert system technology in Air Force programs, and identified key special requirements.

The Portable Inference Engine (PIE) project (Le and Homeier 1988) was intended to produce a single language and environment for Air Force expert systems to be written and run across a wide variety of hardware bases. CLIPS was identified as meeting most of these requirements.

Singular among these is real-time response (Laffey et al. 1988), which we interpret in the context of expert systems as time efficiency. The Rete net algorithm is known to possess optimal efficiency for matching many patterns to many objects (Forgy 1982). It is based on the assumption of a slowly changing state. Air Force requirements involve high rates of data to be processed in real time. Thus the state is changing rapidly, perhaps completely in a short time. This condition does not satisfy the stated assumptions of the Rete net, and thus adaptations of the algorithm are needed.

Typically, 90% of the execution of an expert system is spent in matching (Gupta 1985). Our approach to improving the speed of matching is to reduce the number of rules being considered for matching at any one time. Most real expert systems do not have a flat structure, where all rules are expected to be ready to fire at any point (Winston 1984). Rather, in many cases, there is effectively a current focus of attention, where a few rules are doing the work for the moment, and the rest of the expert system is essentially waiting around for its turn to contribute to the task. Although that waiting sounds passive, it is truly active, since the left-hand-sides of those rules are

participating in the Rete net matching process, and all facts that apply are being pushed as far as possible into the net.

These effects are exacerbated in an environment where the fact database is changing rapidly. Here the inflow of new facts create a large number of mostly unimportant rule activations and deactivations, when relevant facts are removed in favor of more recent data. Limiting the Rete net activity to those rules that are appropriate to the current focus of attention brings significant savings in avoiding unneeded matching.

Human experts often employ a combination of forward and backward reasoning (Geogeff and Bonollo 1983). A control mechanism that consists solely of forward reasoning does not effectively model this process. In response to this, in many cases expert system programmers have built a goal-directed structure into their programs by adding a clause at the beginning of each rule to select the context of the goal for which the rule applies. These clauses then match facts that the programmer asserts manually to invoke the goal. This reduces the set of applicable rules and effectively provides backward reasoning. However, this approach is clumsy, and adds extra overhead for the programmer, who has to perform housekeeping to ensure the timely removal of goal facts, and settle auxiliary issues like conflicts between two concurrent goals. Also, this practice is only a convention, not supported or checked by the expert system language.

We saw these problems arising from the attempt to use data to fulfil the role of control. What is really needed is a new control structure, that manages the goals cleanly and properly with a minimum of effort.

2. APPROACH

We propose the *module* as a collection of rules that participate together in a "focus of attention." These rules are strongly linked, in that they can be considered a small expert system dedicated to solving a single subgoal of the original total problem. There are strong arguments to be made in favor of modules from the points of view of generality, security, software engineering, and simple clarity.

Backward chaining has been generally recognized as an important inferencing capability. While a system may be constructed using only forward or backward chaining, an integration of the two provides an increase in effective computational power, allowing more natural and direct reasoning, and may reduce the length of inference chains. The module system establishes the module as the unit of control for backward chaining, while establishing forward chaining within the module (see Figure 1). The two work symbiotically; it is forward chaining that invokes new goals or returns from them, while the backward chaining controls which rules can be used for forward chaining. This synergy produces expressive and deductive power.

Modules give protection to the expert system. While a module is active, only rules within the module can fire; no rule contained within another module can fire, even if its left-hand-side is satisfied. This helps to prevent an error in expert system programs, where due to an unforeseen combination of interactions between rules during execution an unexpected rule becomes satisfied and fires, which had no relevance to the focus of attention at the time. This problem occurs while maintaining or enlarging a rule base, because of the difficulty in foreseeing all possible interactions between hundreds or thousands of rules. The module concept provides "bulkheads" to contain the flow of control within a module, similar to the independently sealable compartments aboard a submarine.

Modules also support reliability and good software engineering. It is important to construct large systems in pieces, where each piece has a distinct and well-described objective or function, and where the different pieces fit together with simple and clear interfaces. The flat structure of traditional rulebases of hundreds or thousands of independent rules, all at the same level and all interacting, is a software engineering nightmare. The concept of either exhaustively testing or actually forming a mathematical proof of correctness of such a system is clearly beyond question, due to its size and complexity. However, with a set of rules broken up into modules, conceivably each module could be verified independently, since it would contain only a handful

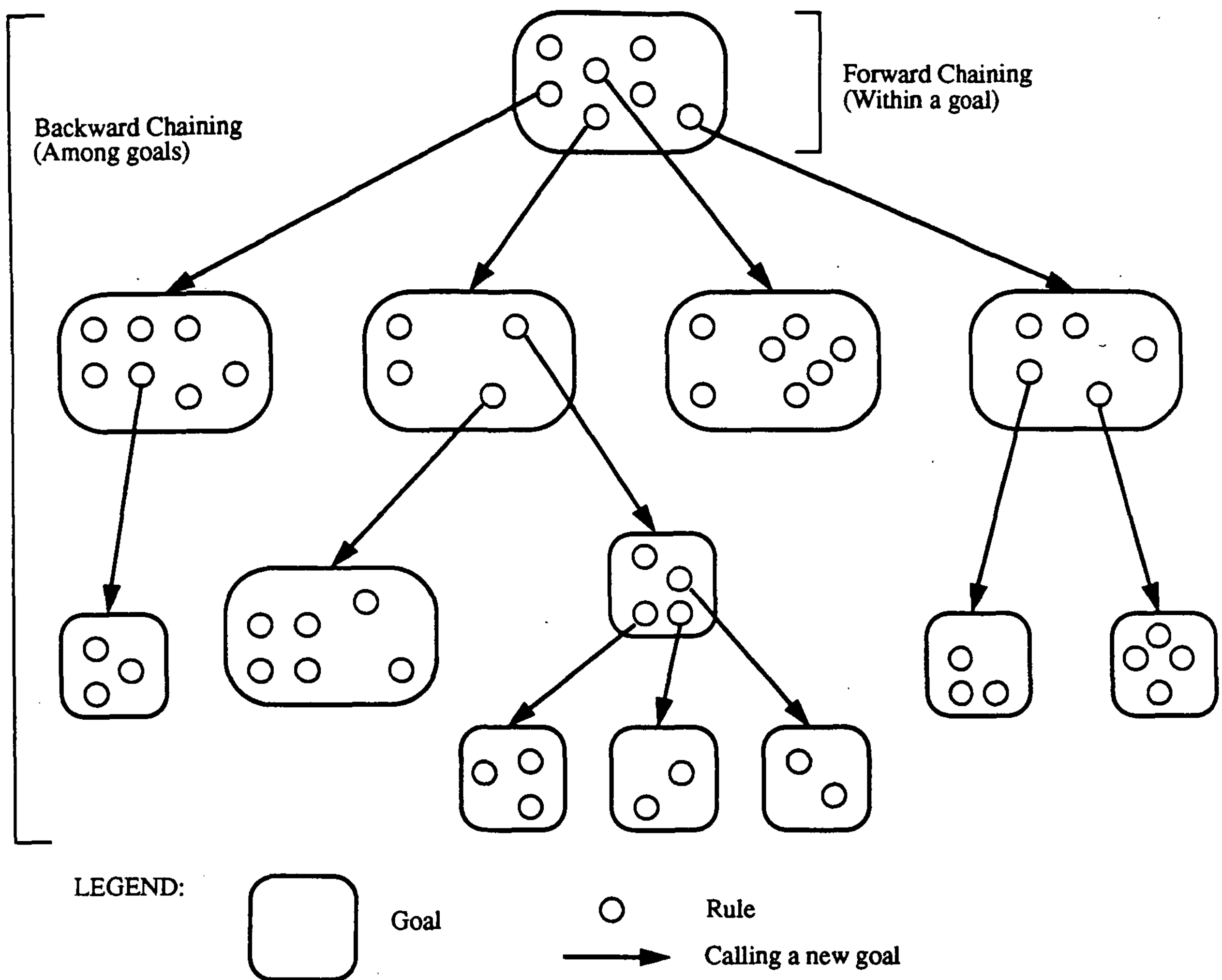


Figure 1. Forward and Backward Chaining in ECLIPS

of rules by comparison, and then the results could be combined for a verification of the entire expert system.

Finally, modules clarify expert systems. Replacing the "goal" clauses in the front of rules by a single module header joining all such related rules made the rules shorter and the rule base shorter. Each rule now simply lists the conditions under which it should fire, given the context that the task at hand is the particular subgoal of this module. Data is no longer being used for control, rather, a simple control mechanism provides that function. Having a module construct to group a set of related rules that together accomplish a single purpose clarifies the whole intent of the expert system, and standardizes the meaning of "the current context." The creation of subgoals is nested like subroutine calls; the module header clearly describes the interface for each "call." Thinking of a module as a subtask that is accomplished out of view and then returns clarifies the thinking of the expert system writer; he can work on rules to solve one goal at a time, without being concerned with the implementation of how other subgoals are achieved.

3. INFORMAL SYNTAX & SEMANTICS

A module is a collection of rules. There is at most one module active at any one time. Only rules in the currently active module, or rules which are global to all modules, may fire. At the beginning of execution, after a reset, no module is active. A rule can activate a module by executing a "goal" statement. When a module is activated, the formerly active module is suspended until the newly activated module returns. Activations of modules are nested, similar to subroutine calls, and may be recursive. A module activation may return by executing a "return" statement. It will also return if there are no rules which may currently be fired. There is no return value; all results must be transmitted through the fact database. A module declaration may also include formal parameters. Each invocation of the module must present a corresponding list of actual parameters, which become accessible within the rules of the module via the formal parameter names. The lifetime of these bindings is the lifetime of the module activation. Activations of modules are also called "goals"; this terminology is intuitive, but introduces possible ambiguity between these activations and the goal statements that invoke them; the ambiguity is resolved by context.

ECLIPS is presently implemented using CLIPS version 4.1. The following discussion illustrates the module syntax and defines the extensions to CLIPS implemented in ECLIPS. We assume a familiarity with expert systems in general and with CLIPS in particular (CLIPS 1987).

3.1. Module Syntax

An ECLIPS program in a file may contain module definitions as well as rules and initial fact definitions. Syntax:

```
(defmodule <module name> ( <list of formal parameters> )[ "comment" ]
   <rule definition> ... )
```

Each module contains one or more rule definitions. The formal parameters are variable names, as in this example:

```
(defmodule move (?obj ?place) "Move the object ?obj to be at ?place." ... )
```

These variables are defined throughout the text of the module, and may be used on the left or right-hand-sides of rules, for example,

```
(defrule move-object-to-place ""
  (monkey ~?place ? ?obj)
  =>
  (goal walk-to ?place))
```

3.2. New Right-Hand-Side Statements

Modules are invoked as goals by using the "goal" statement on the right-hand side of a rule:

```
(goal <module name> <parameter value> ... )
```

for example,

```
(goal at ladder a5-7)
```

The actual parameter values are listed successively after the name of the module. A goal is ended and closed when a "return" statement is executed on the right-hand side of a rule:

```
(return)
```

A goal is also ended and closed when the agenda becomes empty; that is, when no rules of that activation of that module or global rules are ready to fire. When a goal is ended, the suspended goal most recently invoked is resumed.

3.3. New User Commands

The user can display the set of modules that are currently defined with the "modules" command:

```
(modules)
```

The user can display a particular module with the "ppmodule" command:

```
(ppmodule <module name>)
```

During a run, the user can display the current stack of goals, with their actual parameter values, with the "goal-stack" command:

```
(goal-stack)
```

Also during a run, the user can trace the activation and deactivation of goals with the "watch goals" command:

```
(watch goals)
```

"Watch all" now turns on "watch goals" as well as rules, facts, and activations. "Unwatch" also handles the "goals" option.

3.4. New Debugging Commands

The user can trace the development of the Rete join net with the "watch drives" command. Every node in the join net that has a binding driven into it is displayed. This adds considerably to the volume of output.

```
(watch drives)
```

"Unwatch" also handles the "drives" option. To print out the entire Rete join net at a time, the user can give the "show-jn" command:

```
(show-jn)
```

This also generates a considerable amount of output.

```
;;;**************************
;;;* chest unlocking rules *
;;;**************************

(defmodule unlock (?chest) "To unlock ?chest."

  (defrule hold-chest-to-put-on-floor ""
    (object ?chest ? light ~floor ? ?)
    (monkey ? ? ~?chest)
    =>
    (goal holds ?chest))

  (defrule put-chest-on-floor ""
    ?f1 <- (monkey ?place ?on ?chest)
    ?f2 <- (object ?chest held light held ?contains ?key)
    =>
    (printout "Monkey throws " ?chest " off " ?on " onto floor." crlf)
    (retract ?f1 ?f2)
    (assert (monkey ?place ?on blank))
    (assert (object ?chest ?place light floor ?contains ?key)))

  (defrule get-key-to-unlock ""
    (object ?chest ?place ? floor ? ?key)
    (monkey ? ? ~?key)
    =>
    (goal holds ?key))

  (defrule move-to-chest-with-key ""
    (monkey ?mplace ? ?key)
    (object ?chest ?cplace&~?mplace ? floor ? ?key)
    =>
    (goal walk-to ?cplace))

  (defrule unlock-chest-with-key ""
    ?f1 <- (object ?chest ?place ?weight ?on ?obj-in ?key)
    (monkey ?place ?on ?key)
    =>
    (printout "Monkey opens chest with " ?key " revealing " ?obj-in crlf)
    (retract ?f1)
    (assert (object ?chest ?place ?weight ?on nil ?key))
    (assert (object ?obj-in ?place light ?chest nil nil))
    (return))
)
```

Figure 2. An example module

4. EXAMPLE

The example in Figure 2, taken from the monkey-and-bananas problem, shows a module called "unlock", whose purpose is to accomplish the goal of unlocking a chest; the particular chest to unlock is indicated by the formal parameter ?chest. This module represents the rules that accomplish this subgoal of the overall goal of the monkey to eat the bananas. There are five rules in this module, three of which invoke further subgoals as part of solving this one, and two which are able to take immediate action. Only one rule has an explicit return statement. This is an example of a well-coded module, with rules which cooperate in solving a single, well-defined task.

Compare these rules with the corresponding normal CLIPS counterparts:

```
(defrule hold-chest-to-put-on-floor ""
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

versus

```
(defrule hold-chest-to-put-on-floor ""
  (goal-is-to active unlock ?chest)
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  (not (goal-is-to active holds ?chest))
  =>
  (assert (goal-is-to active holds ?chest)))
```

The ECLIPS code for the "unlock" module is shorter by 5 lines, over 12%, and the lines are shorter and less complex. In particular, note the CLIPS need for special code to prevent goal duplication. This section is typical of the entire monkey-and-bananas example.

5. IMPLEMENTATION

The fundamental idea in the implementation is inspired by the example above, but different. The example shows an attempt to implement backward chaining in normal CLIPS using data for control. In ECLIPS, every rule within a module is compiled into the Rete net with an additional clause at its front, describing the module for which this rule is active. For example, the first rule in the example above would be compiled not as

```
(defrule hold-chest-to-put-on-floor ""
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

but as

```
(defrule hold-chest-to-put-on-floor ""
  (goal unlock ?chest)
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

The new clause consists of the standard word "goal", then the name of the module, then the formal argument names. (The word "goal" is not reserved in ECLIPS, but should not be used by the ECLIPS programmer as the first word in facts, to avoid confusion with the module implementation.) This clause is prepared while parsing the module header line; it is stored in the internal CLIPS structures for a clause. Then while parsing each rule within the module, the goal clause structure is copied and prepended to the structure being prepared to represent the list of clauses on the left-hand-side of the rule. The goal clause will match a fact of the correct form, for example

```
(goal unlock red-chest)
```

The "goal" statement, which appears on the right-hand-side of a rule, has the semantics of activating a module. This is implemented by creating a "goal" fact of the form shown above and adding it to the fact database. The new goal fact is automatically driven into the Rete net, and is combined in the usual method with facts that satisfy other clauses of the rules in the module to create activations which are put on the agenda in the normal way. These "goal" facts are removed from the fact database by returning from a module, either by an explicit return statement or by having no further rules to fire.

This method of implementation means that the Rete net does most of the work of managing the applicability of rules. While applicable facts do get accumulated at the upper leaves of the Rete net, none of them can merge with superior clauses until the first clause, the "goal" clause, is matched. When that happens, then all the potential subordinate matches and activations are free to occur.

A data structure is maintained in ECLIPS, called the "goal-stack," which is a stack of the goal facts that have been introduced and not yet removed; it is thus a statement of the modules that have been entered, each with their actual parameter values. The fact on top of the goal-stack describes the currently active module.

When a new module is activated and the prior one is suspended, that suspension is not accomplished by removing the associated goal fact, as might be imagined, but rather through a modification of the conflict resolution strategy employed by CLIPS. CLIPS maintains an agenda which is a list of activations of rules that are ready to fire. The agenda is kept ordered by priority, and within each level of priority, the agenda is ordered by recency, with newest first. As new rule activations are generated, they are added to the appropriate place in this agenda, so that the desired order is preserved. Every expert system must have some policy for deciding which activation of the available set will be chosen to fire. This is called "conflict resolution." The conflict resolution policy implemented in CLIPS simply chooses the first activation in the agenda, i.e. the most recent of the highest priority. However, in ECLIPS, the conflict resolution is modified to choose the first activation in the agenda from the current module, or which is from a global rule, not contained within any module. Thus there may be more recent or higher priority activations which are passed over if they belong to a module which is not the currently active one. Activations therefore are continuing to occur for rules in suspended modules; however, none can fire until those modules become the current module.

It is important to allow these activations from suspended modules, and to leave these activations on the agenda, even though they are "inert" as long as the current module is active. Otherwise, we would lose the property of reflexivity, which assures that if a rule fires, then it will not fire again on the same data that matched its left-hand-side. Reflexivity is accomplished in the CLIPS implementation by simply putting activations onto the agenda when they are generated, and removing them when the activation is actually fired, or when any of their fact support is removed. As long as the facts do not change, the rule is only activated once, and once it is fired, it is off the agenda. This information, whether the rule has fired, would be lost if the goal fact for the rule's module were retracted and later re-asserted when the module ended its period of suspension; we would not know which rules had already fired.

Unlike the attempted CLIPS implementation of backward chaining, these "goal" facts are not visible during normal "(facts)" queries. This is accomplished by giving them negative ID numbers, similar to the "not" facts that are generated to help implement negative clauses, such as

```
(not (object ladder ?place ? ? ? ?))
```

Facts with negative ID numbers are not printed by the "(facts)" command; therefore these goal facts do not clutter up the user's view of the fact database with control-related constructs.

Modules may invoke themselves recursively, either directly or through intermediate modules; this is accomplished automatically in the implementation described above. New goal facts do not interfere with the presence of older ones, even for the same module and with the same actual parameters. The goal stack enables the recursion by keeping track of which goal is current. The Rete net keeps track of which rules are ready to fire and for which module activations. Every rule activation on the agenda keeps a list of the facts which satisfied its left-hand-side; for rules within modules, this includes the goal fact which satisfied the rule's goal clause.

When the search of the agenda cannot find an activation to fire, normal CLIPS will end; ECLIPS however will pop the goal-stack, removing the goal fact associated with the last goal, and re-search the agenda. Removing a goal fact will automatically cause all rule activations from that activation of a module to be removed from the agenda. The goal stack will continue to be popped

until either an appropriate activation is found or the goal-stack is empty, at which time ECLIPS will end.

The activation and return from goals are particularly interesting events to an ECLIPS expert system programmer, and so the "(trace goals)" utility was added to keep track of this changing context. In addition, at any point the user can give the "(goal-stack)" command to print out the entire stack of module activations, with the current module on top of the stack.

During the study of CLIPS, it was necessary to understand its data structures in detail, particularly the Rete net. Some utilities, "(show-jn)" and "(watch drives)" were built to provide visibility to the Rete net and its changes during computation, and these have been retained as generally useful learning tools for those who wish to study a real-life implementation of the Rete net algorithm.

6. COMPARISONS BETWEEN CLIPS AND ECLIPS

Here we compare CLIPS and ECLIPS in size and time. Two of the original examples distributed with CLIPS were recoded in ECLIPS. "mab" is a version of the traditional monkey-and-bananas problem, and "wine" is an expert system to choose an appropriate wine for dinner.

6.1. Size Comparisons

	CLIPS lines	ECLIPS lines	% reduction
mab	235	211	10 %
wine	419	346	17 %

Table 1. Size Comparisons

Some of the savings in the wine example were the result of eliminating 6 rules which only controlled the sequencing between phases of the wine selection process; the module structure allowed a less cumbersome coding.

6.2. Time Comparisons

Here are the results, in seconds, of 100 iterations on a SPARCstation 1, with i/o dependencies reduced by directing output to /dev/null:

	CLIPS	ECLIPS	% reduction
mab	28.1	26.8	4.6 %
wine	12.13	11.06	9.0 %

Table 2. Time Comparisons

The question arises, what is the time efficiency cost of the new features that ECLIPS provides? ECLIPS is completely backwards compatible with CLIPS, so a normal CLIPS program will run exactly the same under ECLIPS; but how much of a performance penalty will it suffer for the additional parsing, new statements and commands, goal stack maintenance, and additional agenda processing? The answer is that it is very difficult to measure any appreciable difference at all! It appears to be about 0.25%; in any case, it is well within the normal variance between runs.

7. FUTURE WORK

As mentioned, this implementation was built on CLIPS version 4.1. CLIPS has undergone many subsequent changes (version 5.0 is being released) but still does not support true backward chaining. We hope to port the ECLIPS modifications to CLIPS version 5.0, and make this capability available to interested parties.

There were several ideas presented in the prior paper (Le and Homeier 1988) that were not included in the implementation, for reasons of time. We described a capability for the user to specify the conflict resolution strategy, enabling the use of dynamic prioritization or other application-specific control strategies. These may support the creation of rule bases more directly matching the expert's control knowledge of how to apply his rules. We also intended originally to allow a *set* of modules to be activated at one time, instead of just one. This would allow, for example, the dynamic "widening" or "narrowing" of the search for the explanation of an anomaly to include more or fewer subsystems of a satellite.

We also considered having the activation of a module immediately cause a suspension of the execution of the RHS of the rule being fired, and then after the module activation returned, the RHS would be resumed at the next statement. Instead, the current implementation merely changes the currently active module, which has its effect upon the next selection of a rule to fire. In addition, we considered extensions where modules returned either a flag indicating success or failure, or an arbitrary value computed as the result of the subgoal.

8. SUMMARY AND CONCLUSION

ECLIPS provides backward chaining in the CLIPS environment in a clean and simple manner, yet it shortens program code, generalizes the inferencing, increases clarity, provides security, supports good software engineering, and runs faster. There is no significant penalty for using ECLIPS in place of CLIPS.

We believe that the module concept is applicable to the majority of all domains, whenever the size of the rule base grows beyond a certain size. We hope that the ideas in ECLIPS contribute to the practical work of the CLIPS community and beyond, enabling the creation of more effective expert systems in all domains.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Russ Abbott, Dr. Jim Hamilton, Dr. Stephen Hsieh, and Charles Simmons who made many valuable comments on this paper. The authors would also like to express their appreciation to Rick Cowan, who provided the initial vision and impetus to the PIE project.

BIBLIOGRAPHY

Brownston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5, Addison Wesley Publishing Company, 1985.

CLIPS Reference Manual, Version 4.0, March 1987, Mission Support Directorate, Mission Planning and Analysis Division, NASA, 87-FM-9, JSC-22552.

Forgy, C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, vol. 19, 1982, p. 17-37.

Forgy, C. and Shepard, S., "Rete: a Fast Match Algorithm," AI Expert, Jan. 1987, p. 34-40.

Geogeff, M. and Bonollo, U., "Procedural Production Systems," Proceeding of the Eighth International Joint Conference on Artificial Intelligence, vol. 1, 8-12 Aug. 1983, Karlsruhe, West Germany, p. 151-157.

Gupta, A., "Parallelism in Production Systems: The Sources and the Expected Speed Up in Expert Systems and Their Applications," Fifth International Workshop Agence de l'Informatique, Avignon, France, 1985, p. 26-57.

Laffey, T., Cox, P., Schmidt, J., Kao, S., and Read, J., "Real-Time Knowledge-Based Systems," AI Magazine, Spring 1988, p. 27-45.

Le, Thach, and Homeier, Peter, "PORTABLE INFERENCE ENGINE: An Extended CLIPS for Real-Time Production Systems", Proceedings of the Second Annual Workshop on Space Operations Automation and Robotics (SOAR '88), July 20-23, 1988, NASA Conference Publication 3019, p.187-192.

Mettry, W., "An Assessment of Tools to Build Large Knowledge-Based Systems," AI Magazine, Winter 1987, p. 81-89.

Michie, D., "Expert Systems," The Computer Journal, vol. 23, 1980, p. 369-376.

Nilsson, N., Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California, 1980

Winston, P., Artificial Intelligence, Second Edition, Addison Wesley Publishing Company, July 1984.

SESSION 5 B

PRECEDING PAGE BLANK NOT FILMED

N92-16595

Extensions to the Parallel Real-time Artificial Intelligence System (PRAIS) for Fault-tolerant Heterogeneous Cycle-stealing Reasoning

David Goldstein
Faculty Associate
goldstn@cse.uta.edu

University of Texas, Arlington
Automation and Robotics Research Institute
7300 Jack Newell Blvd S.
Ft Worth, Texas 76118
USA

Abstract. Extensions to an architecture for real-time, distributed (parallel) knowledge-based systems called the Parallel Real-time Artificial Intelligence System (PRAIS) are discussed. PRAIS strives for transparently parallelizing production (rule-based) systems, even under real-time constraints. PRAIS accomplished these goals (presented at the first annual CLIPS conference) by incorporating a dynamic task scheduler, operating system extensions for fact handling, and message-passing among multiple copies of CLIPS executing on a virtual blackboard. This distributed knowledge-based system tool uses the portability of CLIPS and common message-passing protocols to operate over a heterogeneous network of processors. Results using the original PRAIS architecture over a network of Sun 3's, Sun 4's and VAX's are presented. Mechanisms using the producer-consumer model to extend the architecture for fault-tolerance and distributed truth maintenance initiation are also discussed. Also, recently designed approaches and extensions, including improvements to RETE and an entirely new pattern matching algorithm to meet hard-real-time deadlines are discussed.

This paper is deliberately presented at a high-level, discussing the real-time, fault-tolerance, and distributed nature of the architecture as more detailed descriptions of the work are available elsewhere [Gol90][GT91][Gol91].

0.0 Introduction

Real time artificial intelligence (AI) is an ideal application for parallel processing. Many problems including those in vision, natural language understanding, and multi-sensor fusion entail numerically and symbolically manipulating huge amounts of sensor data. Real time reasoning in these domains is often accomplished via specialized computing resources which are often (1) very difficult to use, (2) very costly to purchase (as in the $250,000 - $2,000,000 PIM [GL]), and (3) guarantee only fast- not real time - performance.

This paper extends some of the ideas behind PRAIS, the Parallel Real-time Artificial Intelligence System, a cost-effective approach real-time computing combining the 'C' Language Integrated Production System, TCP/IP and some novel concepts in pattern matching and real-time control to provide a flexible development environment for distributed knowledge-based systems. The goals of the system are to simplify parallelization, increase portability, and maintain a consistent knowledge representation throughout the system. The system accomplishes these goals by providing transparent scalability of fielded CLIPS applications and by cycle-stealing small amounts of resources over large networks of existing processors.

PRECEDING PAGE BLANK NOT FILMED

1.0 Original Blackboard Architectures

The blackboard architecture [Nii86] has probably been the most successful architecture for addressing complex problems where control structures were not well-defined. KBS's using this architecture feature multiple, independent knowledge sources (KS) each of which reasons about a portion of the domain. Knowledge sources share a global data structure (the metaphorical "blackboard") to share information, in an analogy to experts examining data and hypothesizing solutions on an actual blackboard.

Parallel versions of blackboards provide several advantages. First, each knowledge source may have its own knowledge-base (KB - a database of knowledge driving reasoning), thereby partitioning the system's knowledge and reducing rule interactions. This simplification generally making the system easier to understand and more predictable. Blackboards also facilitate intuitive, hierarchical problem-solving; results from lower level knowledge sources can be used to drive the reasoning processes of higher level knowledge sources. The hierarchical development of hypotheses is very useful, especially useful for problems where disparate data is encountered from multiple sources (e.g. vision, multi-sensor fusion).

An illustration of a real time blackboard system for music generation is depicted in Figure 1. At any given time the system might receive a variety of auditory inputs. These inputs are examined by signal processing resources to extract and place on the blackboard primitives such as frequencies, pulse widths and pulse intervals. These primitives are then used by other processors to determine notes, "instruments", pauses, and durations, which are in turn combined to ascertain tempos, progressions, chords. At the highest levels of processing these deductions are combined with music styles, artistic profiles, scores and music theory to predict future sensor inputs and generate appropriate auditory output.

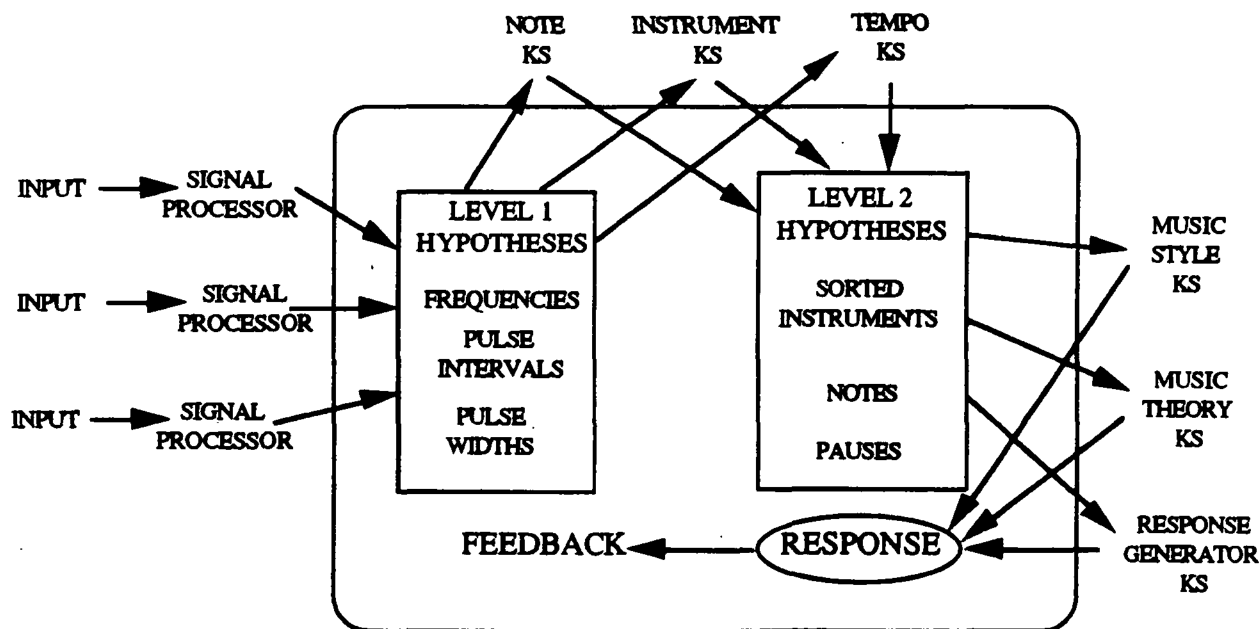


Figure 1. Hierarchical Blackboard Processing

2.0 Directed Blackboards

Directed blackboards is an architecture explicitly designed for forward-chaining production systems to accommodate real-time processing over a heterogeneous network of processors derived from examining numerous investigations into blackboard processing. Like many architectures

derived from blackboards, directed blackboards attempt to improve the basic blackboard model by improving the control mechanisms, reducing the event scheduling required, and minimizing the amount of information that must be distributed (via the blackboard).

3.0 Message Flow and Control

Message-passing among knowledge sources in this architecture is facilitated by associating a goal to each knowledge source. Each goal is managed by a Dijkstra-like guard which administers the actual communications of messages via a single-entry, single-exit point for all information. Messages flow from producing knowledge-sources (as per the producer-consumer model), to the information guards, to consuming knowledge-sources. This model assumes that each information guard knows the recipients of its information; this assumption is fulfilled by analysis of an application's productions to associate knowledge-sources which might use information (which satisfies a given patterns to goals represented by the pattern), see Figure 2: Message Traffic. Further, the information guards administer fault-tolerance algorithms and initiate distributed truth-maintenance algorithms for fault-recovery, essentially isolating these aspects of the knowledge-based system in the communications functions. Finally, although control resides almost solely in the information guards, each processor in the system shares in the distributed control of not only message traffic, but also information focus, in some fashion.

The information guards presented here can act as backing stores of the data transmitted-without the overhead of reasoning upon facts as performed in true backing processors - while the goals are currently being processed. Further, once goals are no longer being currently processed, transaction management can be performed to save the state of the system prior to operating upon new goals. This procedure treats the state-saving algorithms associated to forward-chaining as a nested transaction, with each goal change as a child transaction.

Heterogeneous networks of processors are easily accommodated in this architecture by using standard communications protocols over internet (TCP/IP). In keeping with the philosophy of the underlying inference engine, the 'C' Language Integrated Production System (CLIPS), many of the communications parameters - such as message packet size, packet destinations, etc. - are stored as facts and can be inferenced upon. Because communications functions are isolated in the information guards, evolving communications standards can easily be accommodated. The information contained in and the resources (such as processors) used by real-world systems can be easily accommodated and interfaced with via the same communications mechanisms; the information guards care not whether the producer or consumer of information is a knowledge-based system or a military simulation, as long as communication proceeds using an established protocol.

4.0 Concurrent Processing

The use of network resources to facilitate concurrent processing is straightforward at a coarse level of granularity; placing individual knowledge sources on their own processors in a multi-processing environment is intuitive and has been incorporated in many systems. The next step in increasing concurrency requires partitioning individual knowledge sources. Many message-based systems strive for "rule-level" parallelism, but such parallelism can be trivially accomplished via placing one rule in each knowledge source and running one or more processes containing knowledge sources on each processors. Several other types of parallel processing that are currently being explored include "greedy processing" where multiple rule firings occur internal to a processor before dispersing the information to other processors, based upon a mathematical estimate of the usefulness of the information to other processors, "rule-level" parallelism - but with slept threads instead of processes, and MIMD transputer matching algorithms. Several new match algorithms have already been internally developed for handling real-time processing and to better the performance of RETE, and the parallelization of these algorithms is under investigation.

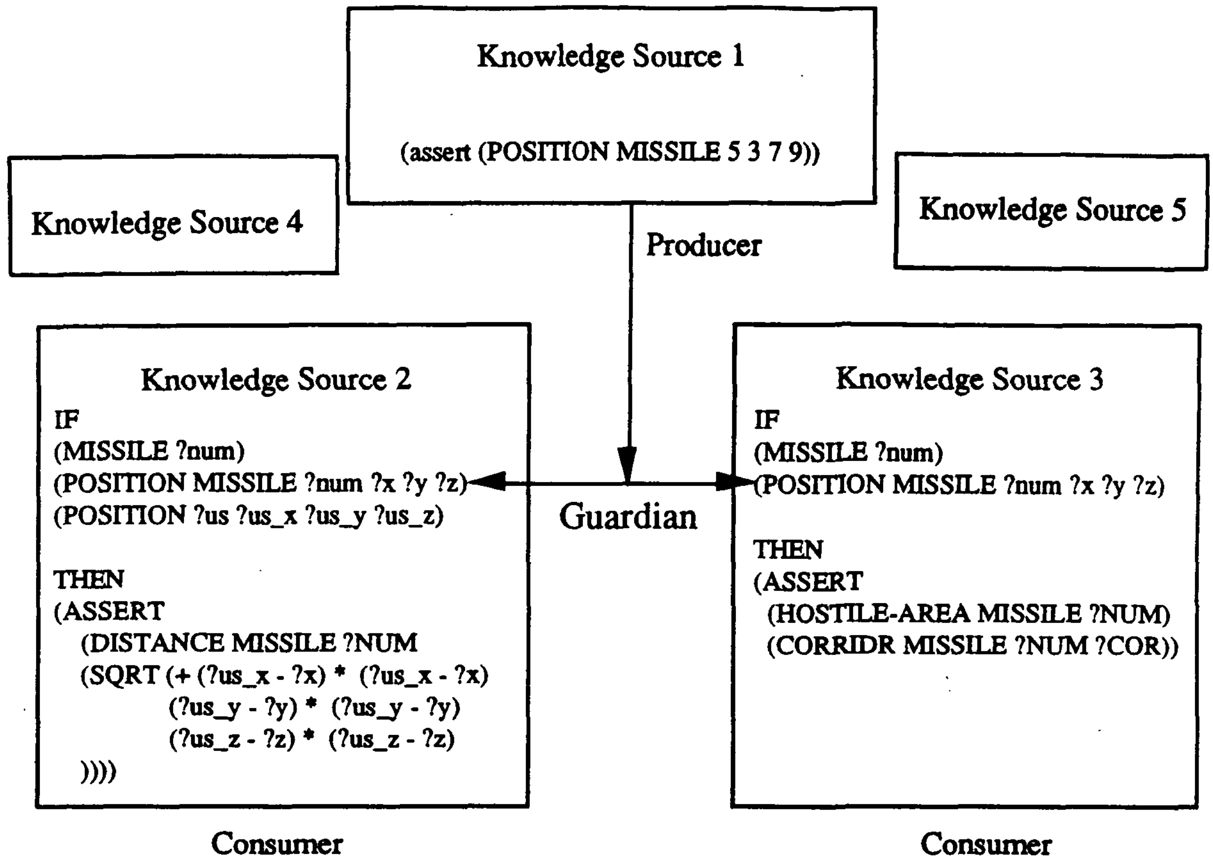


Figure 2. Message Traffic

5.0 Real Time Control Mechanisms

Our original approach for real-time control features dynamically prioritizing tasks based upon criticality and timeliness; each task has a time-varying salience function which accounts for how critical its decision is to the overall system in its current state. The salience of a task is initially low, and increases as the task becomes more important until it become mandatory. Untimely tasks or those of lesser importance can be dropped by the system. Tasks are scheduled for execution based upon a hypothesis of the task's usefulness and likelihood to complete. This algorithm employs resource estimates of the task derived via directed acyclic graphs generated during preprocessing and user guidelines as to the task's importance for generating and using the time-varying salience function (see Figure 3). This approach is in stark contrast to the typical - and computationally expensive - approach of scheduling tasks such that their hard-real-time constraints (deadlines) are met via meta-reasoning. The author feels that planning is a very expensive process, especially for large numbers of tasks, by far exceeds the time frames of executing "ladder-logic" that many real-time applications actually use (such as robotic control). Since the publication of this technique, other parties interested in real-time control, such as Boeing, have investigated similar measures [EB91].

Finally, an "anytime algorithm" extension to RETE is currently under investigation. Such an algorithm should ideally provide an answer from a knowledge-based system regardless of how little time the system is given for reasoning, with the accuracy of the answer proportional to the time allotted for reasoning. Combined with the interruptable reasoning features that have already been placed in PRAIS, this should permit a system to take the best course of action at any time, regardless of system demands.

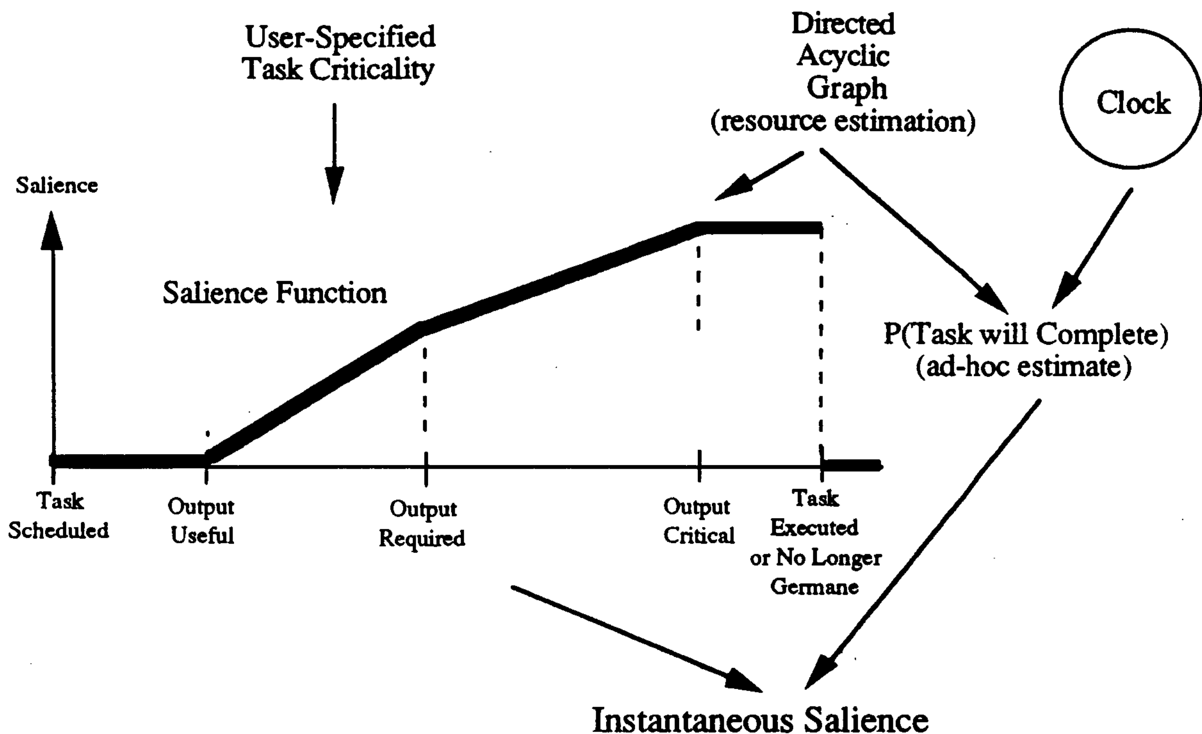


Figure 3. Time-Varying Salience for Task Scheduling

6.0 Knowledge Representation

The knowledge representation used and reasoning processes permitted are virtually identical to those already used by CLIPS. The rule format is depicted in Figure 4: Production Format. Assertions and retractions are handled exactly as there are normally in any serial version of CLIPS, but if the information change should have some affect on some global goal, the goal must be specified.

```
(rule {rule-name}
            (salience {priorities})
            (importance {mandatory/optional/dropable})
            (goal {goal_name})
      ({pattern 1 to be matched as a tuple})
      •
      • ({left-hand-side patterns})
      •
      ({pattern n to be matched as a tuple})
=>
      ({action 1})
      •
      •
      •
      ({right-hand-side action n})
)
```

Figure 4. Production Format

7.0 An Example Application

Presented below is a rule (Figure 5: Rules) pertaining to an image processing application. The $WHERE fact determines where future messages (facts) are to be passed. For fairness, the application partitions the work according to tasks to be performed, and <u>not</u> the data; many experimenters use image processing to demonstrate concurrent processing because image processing parallelizes very easily (by assigning equal size rectangles in the image to each processor). Each processor in this experiment must operate over all the image data, and pass its results to some other processor for further processing.

```
(defrule connect4 "determines if a point is 4-connected"
        (pt ?x ?y)
        (pt ?x =(- ?y 1))
        (pt ?x =(+ ?y 1))
        (pt =(+ ?x 1) ?y)
        (pt =(- ?x 1) ?y)
=>
        (assert     ($WHERE csr)
                     (connected4 ?x ?y))
)
```

Figure 5. Rules

Performance characteristics (with respect to speedup) vary widely from one application to another (because of complexity of the RETE net, size of the factbase, number of facts sent per message, etc) and network resources employed (such as processor types, operating systems and network transmission media). Each of these drives a variety of underlying computational concerns; network and factbase sizes can cause disk swapping while operating systems and processor types can require different conversion strategies at the byte (bit) level.

The current implementation requires approximately one and one-half times as long to receive a transmitted fact via internet as it takes to deduce the fact itself and over seven times the time to send the same fact via internet. Therefore, if decidedly different tasks are worked upon concurrently - perhaps using "island driving" techniques - concurrent processing could yield almost linear speedup. However, with small rule networks (as experimented with here) and geographically separated resources (twenty miles apart) the speedup was not nearly linear.

8.0 Status

The Parallel Real-time Artificial Intelligence System (PRAIS) has been implemented to provide coarse-grained parallel processing over a heterogeneous network of machines, including Sun 3's, Sun 4's, Transputers and DEC VAX's. At the time of this writing the knowledge-based system shell has been modified to accommodate real time processing. The communications algorithms (with an accompanying partially ordered indexing system) have already been implemented. A large number of operating systems issues are currently being addressed, as well as fault-tolerance algorithms and the aforementioned pattern-matching algorithms. Past, current and anticipated PRAIS application areas include real-time sensor-fusion, distributed simulations, robotic control at the workstation level, and parallel planning. Larger rulebases and fault-tolerance/distributed

control experiments will be presented in future papers as more and larger research projects use the system.

9.0 Conclusions

PRAIS already offers a variety of advantages such as:

- heterogeneous hardware capability,
- uniform data flow through the system,
- real time control via dynamic scheduling,
- data - as opposed to algorithm - driven requirements, and
- the use of standard programming practices.

Future capabilities to be incorporated include fault-tolerance, automatically scalable applications, and distributed truth maintenance. This system strives to permit serial code to be converted into a parallel, real time KBS by incorporating many desirable features and functions at low levels of processing.

10.0 References

[CG88] C. Culbert and J. Giarrantano, CLIPS Reference Manual Version 4.2, Artificial Intelligence Section Lyndon B. Johnson Space Center, Houston, Texas, April 1988.

[EB91] W. Erickson and L. Baum, "Real-Time Erasmus", in Proc. of Blackboard Systems Workshop Notes from the Ninth National Conference on Artificial Intelligence, American Association for Artificial Intelligence, Anneheim, CA, July, 1991.

[Gol90] D. Goldstein, "PRAIS: Parallel Real-time Artificial Intelligence System", Fourth International Parallel Processing Symposium Proceedings, Volume 3, Fullerton, CA, USA, 1990.

[Gol91] D. Goldstein, "Integrating Knowledge-bases into Heterogeneous Networks of Processors for Real-World, Real-time Systems", in Proc. of the International Joint Conference on Artificial Intelligence's Integrating Knowledge-bases into Real-world Systems Workshop, Sydney, Australia, USA, 1990.

[GT91] D. Goldstein and J. Tieman, "Heterogeneous Distributed Knowledge-based Systems", in Proc. of the American Association for Artificial Intelligence 1991 Conference's Workshop on Heterogeneous Systems, Anneheim, California, July, 1991.

[Nii86] P. Nii, "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures", The AI Magazine, Summer 1986, pp. 38 - 53.

N92-16596

PCLIPS : Parallel CLIPS

Center for Productivity Enhancement
University of Massachusettes at Lowell

Coranth Gryphon   Mark Miller

Introduction

PCLIPS (Parallel CLIPS) is a set of extensions to the CLIPS expert system language. CLIPS was developed by the Software Technology Branch of NASA, at Johnson Space Center.

PCLIPS is intended to provide an environment for the development of more complex, extensive expert systems. Multiple CLIPS expert systems are now capable of running simultaneously on separate processors, or separate machines, thus dramatically increasing the scope of solvable tasks within the expert systems.

An expert system such as CLIPS attributes its power to the flexibility inherent in its method of storing and disseminating information. An expert system is composed of a rule-base, or set of actions to be performed when certain conditions hold true, and a fact-base, that information the system has regarding the various objects and situations that it knows about. The facts, some of which the system has when it starts off and some of which it acquires through input, are filtered through chain of rules until either a dead-end is reached or a solution is achieved.

As a tool for parallel processing, PCLIPS allows for an expert system to add to its fact-base information generated by other expert systems, thus allowing systems to assist each other in solving a complex problem. This allows individual expert systems to be more compact and efficient, and thus run faster or on smaller machines.

PCLIPS is designed to be used as a tool for multi-processing, with each expert system being responsible for some portion of an overall project. This type of system would have each individual expert system module responsible for the execution of it own individual task, with interaction in the form of information exchange occurring only when necessary. With this model, all of the expert systems execute in parallel, rather than having a single expert system performing multiple roles. When a condition arises within one of expert system which one or more of other modules needs to know about, that information can simply be transferred, rather than requiring the original expert system to deal with the problem itself. This can save significant overhead, as well as making it much less likely that any given expert system will get saturated by having to deal with too much information at any one time.

Parallel Communications

PCLIPS is built upon a communications package specifically designed to handle communications between expert systems. The package operates on a familiar client-server-broker model, with the pclips expert system as the client, and is used to send facts between different pclips objects.

Upon startup, the pclips process spawns off another process, the *server*, whose job is to handle all incoming and outgoing communications with other pclips servers. When it is activated, this server 'registers' with a global broker, which resides upon a well known port of a well known machine. The combination of pclips expert system process and server process is known as a 'pclips object'.

Upon termination of the pclips (client) process, the server notifies the broker that it is now 'unregistering', and then terminates. If the server is terminated by some outside agent, it undergoes the same shutdown procedure, and then notifies the pclips (client) process which is free to deal with this problem according to its defined behavior. This behavior may be to terminate, or to spawn off another server, or to take any other defined behavior.

Names

Each pclips object is assigned a unique name which identfies it to other pclips objects. This name is communicated to the broker upon startup (registration), and the broker verifies that no other pclips object has chosen this name. If no name is specified at the command line, then the object generates one (of the form *<host>-<pid>-<port>*), based upon its current host, process id number, and the port that its server connects to.

Standalone pclips processes (those running without a server as an isolated expert system) use only the host and process id number when generating a name.

Classes

It is also possible to specify a class that the pclips object is an instance of. If no class is specifed, then the pclips object defaults to be a member of the *"pclips"* class. Classes are used to control access and to enable secure communications across certain zones.

A more complete class definitions system for pclips objects is under development which will allow for subclasses and superclasses, resulting in true object inheritance.

Zoned Communications

To prevent the excessive network traffic which would result if every pclips object always talked to every other pclips object, a zoning mechanism has been implemented. There are three actions that can be taken with regards to a zone.

First, the pclips object may 'post-enable' to a zone. The object sends a message to the broker, informing it of its intent. If the object is on the brokers list of those authorized to send a message over that zone (write), then the broker sends back to the server a list of all other pclips objects that can receive a message over that zone. Note that it is not necessary to be able to receive on a zone to send over that zone. When a pclips object is not going to send messages over a zone any more, it can 'post-disable' using the same procedure.

Second, the pclips object may send a message across a zone. Once 'post-enabled' to the zone, the pclips server maintains a list of all other objects that are to receive messages sent over that zone. The pclips (client) processes sends the message to its own server, which then traverses the list of those objects, connects to each one sequentially, and sends the message.

Finally, a pclips object can receive any messages sent on a zone. This is accomplished by sending a message to the broker informing it that the pclips object is 'subscribing' to the specified zone. The broker then sends back to the server the current list of those pclips objects which are allowed to post to that zone. The server then notifies each of those objects to update the internally maintained list of which objects to send to. A pclips object may at any time stop listening to messages over a particular zone by sending an 'unsubscribe-zone' message to the broker, which in turn notifies all pclips objects which are post-enabled to that zone.

Each pclips is automatically subscribed and post-enabled to the zone "*all*", subscribed and post-enabled to a zone based upon its host, "*local-<host>*", subscribed and post-enabled to a zone based upon its class, "*class-<class>*", post-enabled to a manager's zone, "*<class>-manager*", and subscribed to a personal zone based upon its name that only it may receive messages on. This personal zone, of the form "*personal-<name>*" allows point-to-point communications, thus facilitating the query-reply system described below.

In addition, other zones may be created during execution, subscribed or post-enabled to, and communicated over, all controlled by the expert system.

Under CLIPS 5.0, each zone an instance of the defclass "*Zone*", and each pclips object is considered an instance of the defclass "*PClips*". Individual pclips objects are kept track of by their instance name. Since point-to-point communications is done by sending to the zone "*personal-<name>*", the defclass "*PClips*" is a subclass of "*Zone*". This eliminates the conceptual distinction that would otherwise occur between personal zones, and group zones.

A more detailed zone definition mechanism is under developement which will allow zones to fit into the object-oriented paradigm, thus providing for inheritance by way of a zone heriarchy.

Industry Standards

The client-server object, with a global broker, conforms to the Object Management Group's model of an Object Management Architecture *[1]*. By utilizing this standard, greater internal consistency is assured, and more options are left open for other applications to utilize both the PCLIPS communications mechanism and to interface directly to a pclips object itself.

Enhancements to CLIPS

In addition to communications, other related extensions have been added to the basic CLIPS language. The ones discusses here are in some way related to the communications package described above.

Dynamic Construct Creation

Under current expert systems, anytime the user wishes to create a new rule, they must sit down with an editor, create that rule in a file, then load that rule into the expert system. If the system has a command line interface then they also have the option of stopping whatever processing they are doing, again typing in the entire rule by hand, and hopefully saving it out so they don't have to do it all again.

Using the tools provided here, rules (and for CLIPS 5.0, other constructs) can now be automatically generated from fact or instance definitions, thus reducing the work necessary to configure the expert system.

Construct Modification

A second facet of this mechanism is the ability to use existing constructs as a basis to generate new ones, and even to modify existing constructs by reading in the current form, editing it, and replacing the old construct witht he modified form. To this end, the ability to return and parse the pretty-print forms of constructs has been added, thus allowing a pclips expert system to treat these constructs as other forms of data.

Remote Construct Assertion

Using the communications mechanism described above, it is also possible to send directives, in the form of templated facts, to other pclips objects.These directives inform the receiving pclips expert systems in how to generate rules to deal with specific situations, or defines a new network-wide class of objects, or any other type of construct that can be auto-configured.

Constructs can also be stored in the archive system described above in a more compact form and generated, either locally or remotely, as needed.

Archive System

While the fact structure associated with any given fact takes a significant amount of storage space, there is even more that relates to that fact's effects on the rest of the expert system. The CLIPS manual *[2]* specifically mentions that "the fact-list should not be used as a data-base for storage of extraneous information." This statement holds even more for the instance list since there may be many outstanding instances that need to be kept track of for data-base considerations which should not show up under an instance query function search.

The solution to this dilemma is the archive system, which operates in alongside the normal fact and instance managers, and maintains a seperate list of information that the expert system can access, but which does not show up under normal fact or instance searches. Thus only those facts and instances which are needed for the execution of the expert system would be kept in the normal lists, while all others can be archived out.

Remote Archive

Also, using the communications package described above, it is possible to create a remote archive process, whose sole function is to maintain yet another seperate fact and instance list, this one being global to all of the pclips objects that can communicate with it. Rarely used information can be kept in this archive, to be made available on a query basis to any pclips object that needs it.

Construct Storage

An additional feature of both the local and remote archive systems is that, using the automated construct creation mechanism described below, it is possible to efficiently store out any construct form, thus freeing up much more memory in the process. Those rules, functions, class definitions, and message-handlers can then be loaded in only when needed. By storing them in a remote archive site, those constructs now become available to an entire network of pclips objects, on a need-to-know basis, without every pclips object being forced to maintain its own copy of each of them.

Other extensions

Direct Routers

The CLIPS router system allows great flexibility in terms of directing input and output, but it lacks the ability to use this mechanism internally, without resorting to file operations. The direct router system allows a pclips expert system to use internal string routers in much the same way that it uses file routers. The string routers "*direct*" and "*assert*" are automatically defined, the former for generic use, and the latter to provide a global fact construction mechanism. Other string routers can be created and initialized at run time, and then used by the pclips experts system as needed.

Environment Preservation

CLIPS inherently comes with the capability of saving out various portions of its current working state (facts, constructs, instances) to files. The environment preservation package makes use of that and automatically stores the environment when a pclips object terminates. This behavior is selectable, and controlled either at compile-time or at run-time.

This has three potential uses. The first is simply for debugging purposes. It can be very enlightening to look at the 'core' image of the pclips expert system. The second, and more practical use, is to allow a pclips object to 'simulate' any given event-sequence by saving out the current working state, then adding the simulated input, then restoring the environment afterwards.

Finally, the preservation mechanism can enable a pclips object to 'reincarnate' itself on the local host, or on a remote host that has access to the local file system. This is done by storing out the current environment, and then spawning off another copy of the pclips process, either locally or on a remote host, which would then load in the environment and continue with the knowledge of what had occurred. This can be done directly by the user, via the command line flag "*-restore*", or automatically if the appropriate flags are set.

System Operations

Currently, CLIPS allows access to operating system functions via the "*system*" function call. However, not only is this dependent upon specific operating systems, but it requires the expert system to keep track of what host type it is executing on, and to issue the commands appropriate to that system.

The functionality that has been added here is a standardization of the more commonly used command functions, and the ability for the pclips expert system to ignore the specifics of what operating system it is on when all it needs to do is simple file interactions.

In addition, the capability of communicating with users in the outside world has been incorporated into PCLIPS, both by providing a consistent interface to system specific "write/reply" utilities, and by enabling clips to send electronic mail.

There are going to be cases where the external action to be taken is independant, as far as the pclips expert system level, of the specific operating system. One example of this is the execution of a user application. To provide greater flexbility of control, the system interaction capabilities of CLIPS have been expanded to allow for creation of processes on either local or remote host, and to specify whether those processes are run directly by the pclips process or spawned off as child processes.

Finally, this allows for secure systems by eliminating the need for the entire pclips object to run as root when simple built-in system commands or simple stand-alone utilties (which themselves would have root access) can instead be spawned off and made to return the information to the pclips object via local communication with the server.

Host/Network Knowledge

Since PCLIPS makes use of network and host information, both for communication, and when executing processes on remote hosts, it was determined that the pclips expert system needed the ability to extract information about its local host, as well as remote hosts on connected subnets. This takes the form of utilities which allows the expert system to extract information about its host and its subnet environment from the machine itself, as well as being able to generate a 'subnet map' of the local networks which a host resides upon.

Command Line Arguments

Finally, much of the capability and flexibility of PCLIPS has been made readily usable by the addition of a large number of command line arguments.

These include the setting of all the internal flags used by the extensions, the ability to specify a name, class, or password for a pclips process, zone control (subscription and post-enabling) at startup, debuging behavior, both in terms of the clips "*watch*" command and for internel debugging and warning messages, as well as more flexibility to load in different formats of facts and contructs, whether to run as an isolated process, or expand to be a full pclips object (client and server pair), and finally the user can specify at run-time whether to drop into interactive or 'batch' mode.

Future Work

Through the use of parsing functions, information about the local host, its operating system, and its file system can be made directly available to a pclips expert system. This enables a pclips expert system to directly process externally generated information, without needing the user to translate.

PCLIPS Architecture

Certain source files of the original CLIPS code needed to be modified to provide the necessary access to the extensions listed above. The command line module now includes calls to read in pending messages from the server. The rule engine also now includes calls to the read in pending messages. Finally the "*clips-main*" module has been almost completely rewritten, allowing for a non-interactive execution mode different from the *RUN_TIME* setting provided for, as well as making the necessary calls to initialize other portions of the extended code.

In addition, there are many new user defined functions that a pclips expert system has access to. Some of these merely provide greater access to existing CLIPS mechanisms. Others provide system independant access to the underlying machine. In both cases, the expert system has been given as much flexibility of control as possible.

Finally, the entire communications package has been turned into a library, both to provide greater access, and to ensure ease of portability. This allows other programs to better interact with PCLIPS, and to make use of the already developed communications protocols.

Customization

Each group of extensions is fully controllable at the source code level through the setting of compilation flags. Original CLIPS uses a setup header include file to control its internal customization. PCLIPS uses an additional header file to control the extensions.

Other Programs

In addition to programs described above (pclips process, server, and broker), other utilities have been added to the PCLIPS family to enable greater flexibility and usage. Each of these makes use of the "*plib*" communication library described above.

Scratch-Pad

There is a scratch pad for communication with pclips objects, which is able to use either the zoned communications method described above, or to communicate directly with a specific pclips server on the local host. It also has a built in dribble file for debugging and log purposes.

Under a windowing environment (such as X-Windows, or the Apollo Domain pads), the scratch pad is designed to be run in a seperate window, thus allowing the pclips process to run non-interactively, and freeing up that window (if there is one) to show only expert system output and whatever debugging information is being displayed.

Pipe-Assert

To enable access to system utilities, a "*pipe-assert*" program exists which reads from standard input and transmits each line as fact (with appropriate prefacing) to the pclips server, which in turn returns it to the pclips process itself as a local fact. Other utilities have been written which use a built-in version of this to communicate directly to the server.

PCLIPS Security

Since PCLIPS can be used for sensitive projects (the MASE project *[3]*), it is important for PCLIPS to have internal security capabilities, to minimize the possibility of a foreign process reading messages it is not supposed to see, and writing out messages over zones that it is not cleared for. The first level of security in PCLIPS is achieved using the name/password scheme.

Passwords

A pclips process can be given a password, which allows for authentication of that pclips object. If a pclips object attempts to register with a "known" name, then that process must also supply the associated password for that object. Otherwise, the registration will fail. The password is sent as part of the registration message to the broker.

register name: *<name>* port: *<port>* pid: *<pid>* uid: *<uid>* host: *<host>*
ip: *<ip_address>* [password: *<password>*] \n

The password is currently sent in its unencrypted form. When the broker receives the "*register*" request, it encrypts the password, and then checks that encrypted form against against the entries in its password file.

Password/Access Control File

The broker maintains a password file, which contains the names and passwords of certain "known" pclips objects, as well as class information, and any restrictions or requirements on zone access.

It is possible to specify that certain names are reserved for certain users, or can only run on certain hosts, or that they can only belong to certain classes. If a pclips object which does not meet these criterion tried to use that name, it would fail to register.

Alternately, for a given specified class, it is possible to limit the allowable names that objects in that class can use. Likewise, for a given host, there can be restrictions on the names of objects that run on that host. A pclips object of a given name which attempts to belong to a restricted class, or run on a restricted host, would similarly fail to register.

Finally, zones can be limited such that only specific named objects are permitted to subscribe and/or post-enable to that zone.

Authorization Numbers

In order to maintain some form of security within PCLIPS, the broker now uses internally generated authorization numbers. Whenever an object registers, a unique authorization number, called the registration number, is passed back. This registration number is known only by the broker and the pclips object. Any further requests from the pclips object to the broker must include this registration number, as a way of authenticating future requests. This greatly reduces the potential of another process mimicking an already registered pclips object, using false requests to gain access to unauthorized information.

The unique authorization number is generated by the broker using the following algorithm: when the broker first starts up, it gets the current time from the system. This is then used a seed for a random number generator. Every time a new event occurs in the broker where a new authorization number is required, a call is made to the random number generator.

Zone Authorization

When an object subscribes to a zone, it is given a unique subscription authorization number for that zone. It is also given a list of all objects which are post-enabled to that zone, and each of their unique poster authorization numbers. The subscribing object then sends a message to each post-enabled object, which includes the unique poster authorization number for that object, and the subscription authorization number of the subscribing object. The post-enabled object then adds this subscription authorization number to its internally maintained list of objects subscribed to that zone. When this object later sends a message out over the zone, it will connect to a subscribed object, and include in the message the unique subscription authorization number for that receiving object. Thus, the subscriber can verify that the message it received actually came from a pclips object which was authorized to post on that zone by the broker.

Authorization numbers inhibit a foreign process from sending an illegitimate message to the pclips server, simply by writing out to that server's port. If the sending object has not gone through the authorization procedure, it should not have the necessary authorization number, and hence any incoming messages from it would be ignored. The authorization routine implemented by the broker attempts to verify that the process is a valid member of the PCLIPS family, and that the process is being run by a valid user.

Unique authorization numbers are needed for combination of object, zone, and access (post-enable or subscribe) to prevent legitimate pclips objects from getting enough information to by-pass the security of the broker. This is very important, since it is left up to the individual pclips objects to send messages directly to other objects, and to notify them of zone access changes, rather than relying upon a centralized message handler.

Specifically, if different subscribe and post-enable were not used, it would be possible for an object to get post-enable permission to zone, and then falsely inform other pclips objects that it is a subscribed to that zone, since it has the single authorization number for that zone, and thus gain illegal subscription access to that zone.

Notification Procedure

When an object post-enables to a zone, the broker sends it a list of all the pclips objects which are subscribed to that zone, along with their individual subscription authorization numbers. When the object posts (transmits a message) over that zone, it must include the subscription authorization number of the pclips object which is receiving that message.

Similarly, when an object subscribes to a zone, the broker sends it a list of all pclips objects which are post-enabled to that zone, along with each of their posting authorization numbers. It then notifies each of those objects that it is subscribed to this zone, and must include in that notification its own subscription authorization number. Those pclips objects each add the subscribed object, and its unique subscriber authorization number, to their internally maintained list of objects to send to.

Future Work

Under this current implementation, the security of PCLIPS is only as good as the security of the host running the broker. Since the broker runs as root on its host, an unauthorized user who gets access to root on the broker's machine would be able to read the broker's database (password and zone access control list) files, thus comprimising the entire authorization scheme. The fault-tolerant, multiple broker scheme currently under development eliminates this problem.

We also have the security hole inherent in unencrypted messages. If an unauthorized user was able to capture packets traversing the ethernet, the user would then be able to read passwords and authorization numbers. We will close this hole by developing an encryption capability, to be used when higher levels of security are required. This will have a performance trade-off, since outgoing messages would have to be encrypted at the point of origin, and then decrypted at the point of reception.

Expert System Applications

Given the above extensions, a number of new expert system mechanisms have already been added. Some of these are themselves extensions to PCLIPS, while others are simply examples of what can be done with those extensions.

The system described below, like most of the systems designed around these extenstions, is implemented using both templated facts and the CLIPS Object System. The reason for this is two-fold; first, most of these mechanisms were origianally implemented under CLIPS 4.3, and thus the Object System did not exist. However, in many cases the object-oriented approach yielded greater functionality with lower overhead. Second, there are many users of CLIPS 5.0 who will not use either the rule-bases portion or the Object System of CLIPS, and providing both forms allows for the greatest possible distribution.

Query-Reply System

One example of the automated contruct configuration mechanism (described above) is the Query-Reply system which has been implemented, intended to allow differnt expert system to transfer information on a need-to-know basis. In other words, if an expert system running on one machine needed to know certain information which was known by an expert system running on a different machine, then the former would send a query to the latter, which would reply with the requested information.

A type of query, and the means of generating the correct reply to it, is specified. A rule is then automatically generated to take incoming queries and formulates the reply, which is then sent out to the specified reply point (usually the sender).

Templated Fact Implementation

A templated fact is specified which defines how the receiving expert system deals with a query.

```
(deftemplate AUTO-REPLY
        (field query-opcode (type WORD))
        (field auto-gen-report (allowed-words yes no))
        (field reply-point (allowed-words specified sender none))
        (field auto-post-enable (allowed-words yes no))
        (field retract-fact (allowed-words yes no))
        (field reply-salience (type NUMBER) (default 0))
        (multi-field query-funcs (type STRING)))
```

The "*query-opcode*" field specifies the type of query. The "*auto-gen-report*" field specifies whether the requested information should be printed out by the local expert system when a query is received.

The "*reply-point*" field is used to determine the point to send the information back to. The value "*specified*" means that the reply should be to the zone specified in the query message. A value of "*sender*" means reply to the sender of the query, regardless of the reply point specified in the message. A value of "*none*" means that no reply message should be sent. This is used in conjunction with report to display information for debugging purposes.

The "*auto-post-enable*" field determines if the replying pclips expert system should automatically attempt to post-enable to the reply-point zone.

The fields "*retract-fact*" and "*reply-salience*" are used when multiple replies are possible to a single query.

Finally, the "*query-funcs*" multi-field specify which functions (either primitive, deffunction, defgeneric, or message-handler) return the needed information. Each string is a single function call, with any needed arguments included.

When a query message is generated by the sending pclips expert system, it has a tag symbol added to it. This tag is used to uniquely associate a specific reply with a given query.

CLIPS Revisions

At the time of this writing, CLIPS 4.3 is supported to the limit of its functionality. There are many weaknesses found in the older version of CLIPS which have been fixed under later versions.

In those cases where our extensions have been mirrored, or superceded, by new functions found in CLIPS 5.0, our extensions have been eliminated, and the inherent CLIPS code used. All of the functionality that exists in our extensions CLIPS 4.3 are found under CLIPS 5.0, either inherent or added. In addition many other changes under the newer version have made possible extensions that could not easily be done, or were never thought of, in the older version of CLIPS. Some examples are listed below.

Because it was almost impossible to access fact addresses at the clips expert system level under CLIPS 4.3, the archive system dealt with fixed strings, tagged by a key name. This mechanism was completely revamped to take advantage of the greater accessability to fact addresses found under CLIPS 5.0, as well as expanding to include archival of instances.

The existance of the "*defglobal*" construct made the global variable system that was under developement obsolete, and thus it was dropped.

Since CLIPS 4.3 did not have the inherent capability to directly generate new constructs at runtime, a rule-creation mechanism was enginerred. This mechanism took strings and built the specified rule accordingly. This rule was then loaded in via a direct routing system. Later versions of CLIPS have this functionality inherent, using the "*build*" function call. Those parts of the old system which were still useful were instead moved up to the expert system level, using the "*deffunction*" construct.

With the addition of the CLIPS Object System, the entire communications package has been revamped to take advantage of it, while still retaining enough internal consistency that older versions of PCLIPS can still send and receive the same messages.

Conclusions

The basic approach throughout all of PCLIPS development has been to add primitive functionality to CLIPS, not simply more used-defined functions. By adding these primitive, the expert system can take advantage of features that before were inaccessible, or did not exist, and combine them as needed. Also, arbitrary distinctions between contruct and object types have been eliminated to provide parallel utlity to similar things wherever possible.

In addition to the work described is paper, there is an entire truth maintenance system that is being developed. It works in conjunction with the extensions described above, but is a seperate enough project to warrant its own paper *[4]*.

Future work includes both rounding out and solidifying the existing code, as well as adding even more major significant extensions in the areas of planning and predition, context dependant reasoning, goal-directed backward chaining, distributed agendas, and neural network interaction.

System Support

Written in native TCP/IP and using C, PCLIPS is portable to most multi-tasking platforms. Versions built on CLIPS 4.3 and 5.0 are currently supported under SunOS, System V Unix (Stellar and DG Aviion), DEC Ultrix (3.1 and 4.1), and Apollo Domain OS. A version running VAX VMS is currently under development

Bibliography

[1] Object Management Group Standards Manual, Draft 0.1 ; Richard M. Stoley, Ph.D.

OMG TC Document 90.5.4 ; (c) May 25, 1990

[2] CLIPS 5.0 Reference Manual, Volume 2 ; (c) January 11, 1991

Software Technology Branch, Lyndon B. Johnson Space Center, NASA

[3] MASE : Management and Security Expert; Mark Miller, Coranth Gryphon, et al.;

University of Massachusettes at Lowell, Center for Productivity Enhancement;

(c) August 1, 1991; Published at the Second Annual CLIPS Users Conference

[4] Temporal Reasoning Extensions to CLIPS; Coranth Gryphon, Marion Williams, et al.

Univesity of Massachusetes at Lowell, Center for Productivity Enhancement;

(c) August 15, 1991; a working paper

Separating Domain and Control Knowledge using Agenda

Paul Haley

Paper not received in time for publication

N92-16597

Integrating CLIPS Applications into Heterogeneous Distributed Systems

Richard M. Adler

Symbiotics, Inc.
875 Main Street Cambridge, MA 02139 (617) 876-3633

Abstract. SOCIAL is an advanced, object-oriented development tool for integrating intelligent and conventional applications across heterogeneous hardware and software platforms. SOCIAL defines a family of "wrapper" objects called *Agents*, which incorporate predefined capabilities for distributed communication and control. Developers embed applications within Agents and establish interactions between distributed Agents via non-intrusive message-based interfaces. This paper describes a predefined SOCIAL Agent that is specialized for integrating CLIPS-based applications. The Agent's high-level Application Programming Interface supports *bidirectional* flow of data, knowledge, and commands to other Agents, enabling CLIPS applications to initiate interactions autonomously, and respond to requests and results from heterogeneous, remote systems. The design and operation of CLIPS Agents is illustrated with two distributed applications that integrate CLIPS-based expert systems with other intelligent systems for isolating and managing problems in the Space Shuttle Launch Processing System at the NASA Kennedy Space Center.

INTRODUCTION

The central problems of developing heterogeneous distributed systems include:

- communicating across a distributed network of diverse computers and operating systems in the absence of uniform interprocess communication services;

- specifying and translating information (i.e., data, knowledge, commands), across applications, programming languages and development shells with incompatible native representational models, programmatic data and control interfaces;

- coordinating problem-solving across heterogeneous applications, both intelligent and conventional, that were designed to operate as independent, standalone systems.

- accomplishing these integration tasks *non-intrusively*, to minimize re-engineering costs for existing systems and to ensure maintainability and extensibility of new systems.

SOCIAL is an innovative distributed computing tool that provides a unified, object-oriented solution to these difficult problems (Adler 1991). SOCIAL provides a family of "wrapper" objects called *Agents*, which supply predefined capabilities for distributed communication, control, and information management. Developers embed applications in Agents, using high-level, message-based interfaces to specify interactions between programs, their embedding Agents, and other application Agents. These message-based Application Programming Interfaces (APIs) conceal low-level complexities of distributed computing, such as network protocols and platform-specific interprocess communication models (e.g., remote procedure calls, pipes, streams). This means that distributed systems can be developed by programmers who lack expertise in system-level communications (e.g., Remote Procedure Calls, TCP/IP, ports and sockets, platform-specific data architectures). Equally important, SOCIAL's high-level APIs enforce a clear separation between

application-specific functionality and generic distributed communication and control capabilities. This partitioning promotes modularity, maintainability, extensibility, and portability.

This paper describes a particular element of the SOCIAL development framework called a CLIPS Knowledge Gateway Agent. Knowledge Gateways are SOCIAL Agents that are specialized for integrating intelligent systems implemented using standardized AI development shells such as CLIPS and KEE. Knowledge Gateways exploit object-oriented inheritance to isolate and abstract a shell- and application-independent model for distributed communication and control. Particular subclasses of Knowledge Gateway Agents, such as the CLIPS Gateway, add a dedicated high-level API for transporting information and commands across the given shell's data model and data and control interfaces. To integrate a CLIPS application, a developer simply (a) creates a subclass of the CLIPS Gateway Agent class, and (b) specializes it using the high-level CLIPS Gateway API to define the desired message-based interactions between the program, its embedding Gateway, and other application Agents.

The remainder of the paper is divided into three major parts. The first section provides an overview of SOCIAL, emphasizing the lower-level distributed computing building blocks underlying Gateway Agents. The second section describes the architecture and functionality of Knowledge Gateway Agents. Structures and behaviors specific to the CLIPS Gateway are used for illustration. The third section presents two examples of SOCIAL applications that integrate CLIPS-based expert systems with other intelligent systems for isolating and managing problems in the Space Shuttle Launch Processing System at the NASA Kennedy Space Center.

OVERVIEW OF SOCIAL

SOCIAL consists of a unified collection of object-oriented tools for distributed computing, depicted below in Figure 1. Briefly, SOCIAL's predefined distributed processing functions are bundled together in objects called *Agents*. Agents represent the active computational processes within a distributed system. Developers assemble distributed systems by (a) selecting Agents with suitable integration behaviors from SOCIAL's library of predefined Agent classes, and (b) using dedicated APIs to embed individual application elements within Agents and to establish the desired distributed interactions among their embedded applications. A separate interface allows developers to create entirely new Agent classes by combining (or extending) lower-level SOCIAL elements to satisfy unique application requirements (e.g., supporting a custom, in-house development tool). These new Agent types can be incorporated into SOCIAL's Agent library for subsequent reuse or adaptation. The following subsections review SOCIAL's major subsystems.

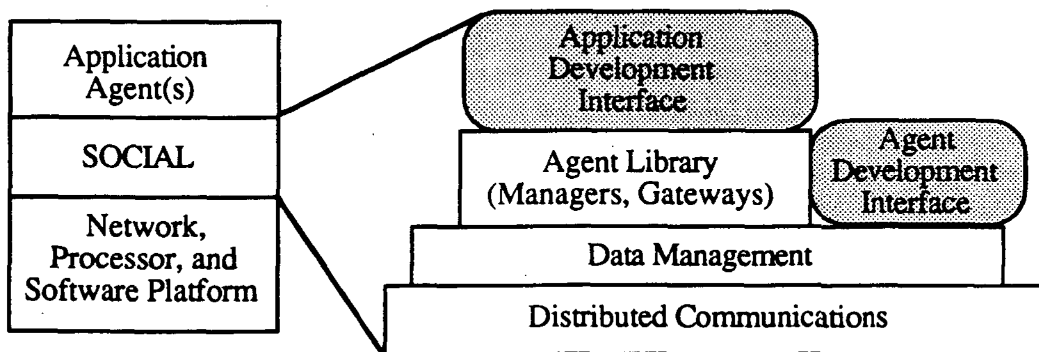


Figure 1. Architecture of SOCIAL

Distributed Communication

SOCIAL's distributed computing utilities are organized in layers, enabling complex functions to be built up from simpler ones. The base or substrate layer of SOCIAL is the MetaCourier tool, which provides a high-level, modular distributed communications capability for passing information between applications based on heterogeneous languages, platforms, operating systems, networks, and network protocols (Symbiotics 1990). The basic Agent objects that integrate software programs or information resources are defined at SOCIAL's MetaCourier level. Developers use the MetaCourier API to pass messages between applications and their embedding Agents, as well as among application Agents. Messages typically consist of (a) commands that an Agent passes directly into its embedded application, such as database queries or calls to execute signal processing programs; (b) data arguments to program commands that an Agent might call to invoke its embedded application; or (c) symbolic flags or keywords that signal the Agent to invoke one or another fully preprogrammed interactions with its embedded application.

For example, a high-level MetaCourier API call issued from a local LISP-based application Agent such as *(Tell :agent 'sensor-monitor :sys 'Symb1 '(poll measurement-Z))* transports the message contents, in this case a command to poll measurement-Z, from the calling program to the Agent *sensor-monitor* resident on platform *Symb1*. The Tell function initiates a message transaction based on an asynchronous communication model; once the message is issued, the application Agent can immediately move on to other processing tasks. The MetaCourier API also provides a synchronous "Tell-and-Block" message function for "wait-and-see" processing models.

Agents contain two procedural methods that control the processing of messages, called :in-filters and :out-filters. In-filters parse incoming messages, based on a contents structure that is specified when the Agent is defined. After parsing a message, an :in-filter typically either invokes the Agent's embedded application, or passes the message (which it may modify) on to another Agent. The MetaCourier semantic model entails a directed acyclic computational graph of passed messages. When no further passes are required, the :in-filter of the terminal Agent runs to completion. This Agent's :out-filter method is then executed to prepare a message reply, which is automatically returned (and possibly modified) through the :out-filters of intermediate Agents back to the originating Agent. Developers specify the logic of :in-filters and :out-filters to meet their particular requirements for application interactions.

A MetaCourier runtime kernel resides on each application host. The kernel provides (a) a uniform message-passing interface across network platforms; and (b) a scheduler for managing messages and Agent processes (i.e., executing :filter methods). Each Agent contains two attributes (slots) that specify associated Host and Environment objects. These MetaCourier objects define particular hardware and software execution contexts for Agents, including the host processor type, operating system, network type and address, language compiler, linker, and editor. The MetaCourier kernel uses the Host and Environment associations to manage the hardware and software platform specific dependencies that arise in transporting messages between heterogeneous, distributed Agents (cf. Figure 2).

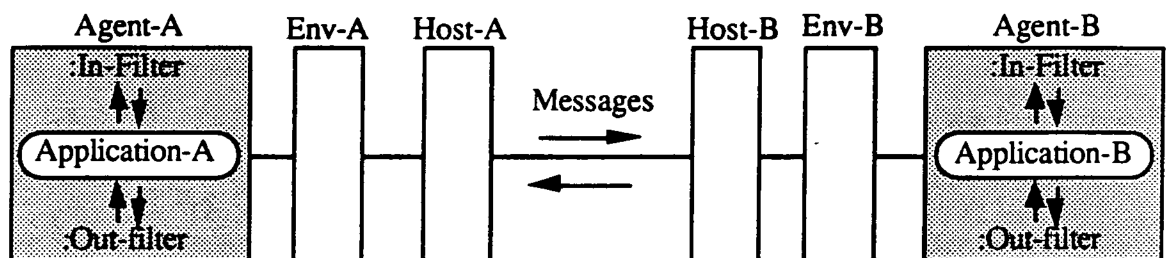


Figure 2. Operational Model of MetaCourier Message-Passing

MetaCourier's high-level message-based API is basically identical across different languages such as C, C++, and Lisp. Equally important, MetaCourier's communication model is also symmetrical or "peer-to-peer." In contrast, client-server computing, a popular alternative model for distributed communication, is asymmetric: clients are active (i.e., only clients can initiate communication) while servers are passive. Moreover, while multiple clients can interact with a particular server, a specific client process can only interact with a single (hardwired) server. MetaCourier's communication model eliminates these restrictions on interprocess interactions.

Data Specification and Translation

A major difficulty in getting heterogeneous applications and information resources to interact with one another is the basic incompatibility of their underlying models for representing data, knowledge, and commands. These problems are compounded when applications are distributed across heterogeneous computing platforms with different data architectures (e.g., opposing byte ordering conventions).

SOCIAL applies a uniform "plug compatible" approach to these issues. This approach consists of two elements, a design methodology and a set of tools to support that methodology. SOCIAL defines a uniform application-independent information model. In the case of Knowledge Gateways, the information model defines a set of common data elements commonly used in intelligent systems, including facts, fact-groups, frames/objects, and rules. SOCIAL's Data Management Subsystem (DMS) provides tools (a) for defining canonical structures to represent these data types, and (b) for accessing and manipulating application-specific examples of these structures. These tools are essentially uniform across programming languages. Equally important, DMS tools encode and decode basic data types transparently across different machine architectures (e.g., character, integer, float).

Developers use SOCIAL's DMS tools to construct intermediate-level APIs for the Gateway Agent class that integrates particular applications. This API establishes mappings between SOCIAL's "neutral exchange" structures and the native representational model for the target application or application shell. For example, the API for the CLIPS Knowledge Gateway Agent translates between DMS frames and CLIPS deftemplates or fact-groups (e.g., Make-CLIPS-fact-group-from-frame Frame-x). Similarly, the KEE Gateway API transparently converts DMS frames to KEE units and KEE units back into frames. If necessary, new DMS data types and supporting API enhancements can be defined to extend SOCIAL's neutral exchange model. This uniform mapping approach simplifies the problem of interconnecting N disparate systems from O(N*N) to O(N), as illustrated in Figure 3.

SOCIAL integrates DMS with MetaCourier to obtain transparent distributed communication of complex data structures across heterogeneous computer platforms as well as across disparate applications: developers embed DMS API function calls within the :in-filter and :out-filter methods of interacting Agents, using MetaCourier messages to transport DMS data structures across applications residing on distributed hosts. DMS API functions decode and encode message contents, mapping information to and from the native representational models of source and target applications and DMS objects. SOCIAL thereby separates distributed communication from data specification and translation, and cleanly partitions both kinds of generic functionality from application-specific processing.

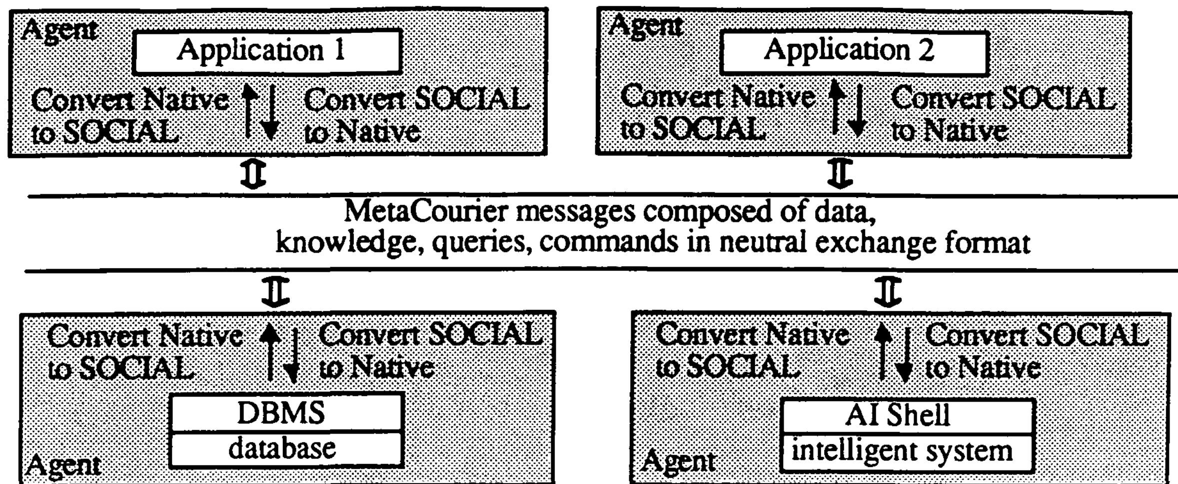


Figure 3. SOCIAL's Plug-Compatible Approach to Managing Heterogeneous Data

Distributed Control (Specialized Agents and Agent APIs)

SOCIAL's third layer of object-oriented tools establishes a library of predefined Agents classes and associated high-level API interfaces that are specialized for particular integration or coordination functionality. MetaCourier and DMS API functions are used to construct Agent API data and control interfaces. These high-level Agent APIs largely conceal lower-level MetaCourier and DMS interfaces from SOCIAL users. Thus, developers typically use specialized Agent classes as the primary building blocks for constructing distributed systems, accessing the functionality of each such Agent type through its dedicated high-level API. If necessary, developers can define new Agent classes and APIs by specializing (e.g., modifying or extending) existing ones.

Currently, SOCIAL's library defines Gateway and Manager Agent classes. Gateways, as noted earlier, simplify the integration of applications based on development tools such as AI shells, DBMSs, CASE tools, 4GLs, and so on. Manager Agents are specialized to coordinate application Agents to work together cooperatively. The HDC-Manager (for Hierarchical Distributed Control) functions much like a human manager, mediating interactions among "subordinate" application Agents and between subordinates and the outside world. The Manager acts as an intelligent router of task requests, based on a directory knowledge base that identifies available services (e.g., data, problem-solving skills) and the application Agents that support them. The Manager also provides a global, shared-memory "bulletin-board." Application Agents are only required to know the names of services within the Manager's scope and the high-level API for interacting with the Manager; they do not need to know about the functionality, structure, location, or even the existence of particular application Agents. The Manager establishes a layer of control abstraction, decoupling applications from one another. This directory-driven approach to Agent interaction promotes maintainability and extensibility, and is particularly valuable in complex distributed systems that evolve as applications are enhanced or added over an extended lifecycle.

KNOWLEDGE GATEWAY AGENTS

Knowledge Gateway Agents combine several important SOCIAL tools and design concepts:

- MetaCourier's high-level, message-based distributed communication capabilities for remote interactions across disparate hardware and software environments;

- DMS data modeling and mapping facilities for transparently moving data, knowledge, and control structures across disparate applications and shells;

• a modular object-oriented architecture that defines a uniform partitioning of integration functionality;

• a non-intrusive design methodology for programming specific, discrete interactions between the application being integrated, its embedding Gateway Agent, and other application Agents;

• extensibility to encompass generalized hooks for security, error management, and session management utilities.

Gateway functional capabilities are distributed across the class hierarchy of Gateways to exploit object-oriented inheritance of behaviors of common utility across Agent subclasses. The partitioning and inheritance of behaviors are summarized below in Figure 4.

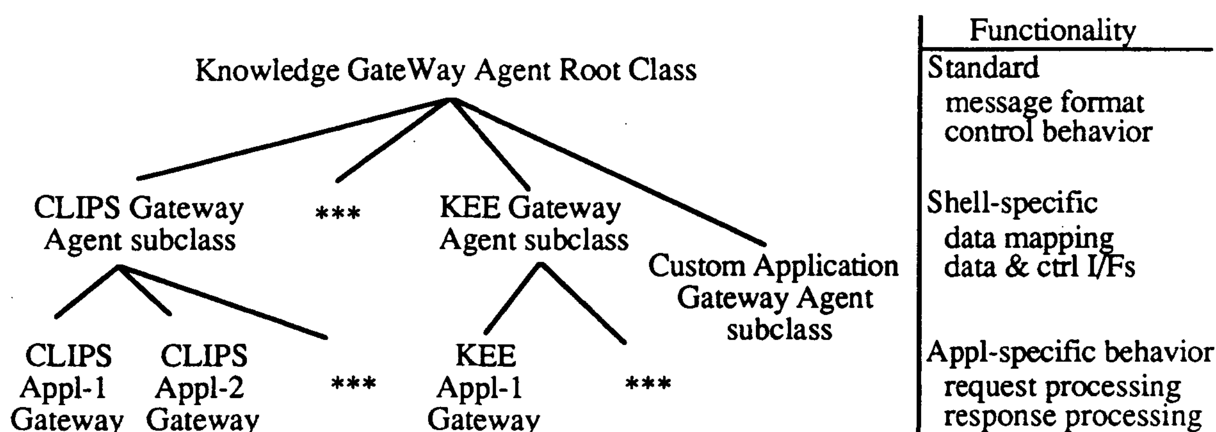


Figure 4. Inheritance of SOCIAL Knowledge Gateway Agent Behaviors

The root Knowledge Gateway Agent, KNOWL-GW, defines the overall structure and functional behavior of all Agent subclasses that are developed to integrate shell-based (or custom) intelligent applications. In particular, KNOWL-GW establishes:

- the uniform MetaCourier/DMS message format structure for communicating with *all* Knowledge Gateway Agents;
- the Agent :in-filter method for parsing and managing incoming messages;
- the Agent :out-filter method for post-processing results;
- default (stub) API methods that are overridden at the Gateway Agent subclass level.

Knowledge-based systems, like most conventional systems, typically function as servers, responding to programmatic (or user) queries or commands. In a server configuration, data and control flow in, while data (results) alone flows out. However, intelligent systems can also initiate control activities autonomously, in response to dynamic, data-driven reasoning. This active (client) role entails "derived" requirements for capabilities to process results returned in response to previous outgoing messages. Since intelligent systems can be configured to act as clients, servers, or play *both* roles within the same distributed application, any generalized integration technology such as Knowledge Gateways must support *bidirectional* flow of data and control.

Accordingly, the generic :in-filter method handles two cases (a) a MetaCourier message coming in from some external application Agent to be handled by the Knowledge Gateway's embedded application, and (b) a message from the embedded application that is to be passed via the Knowledge Gateway Agent to some external application Agent. The KNOWL-GW distinguishes the two cases automatically, based on the message's target Agent. Similarly, the generic :out-filter method handles two cases (a) dispatching the embedded application's reply via the Knowledge Gateway Agent to the external requesting Agent, and (b) processing the response from an external application Agent to a message passed by the Gateway from its embedded application and injecting it back into the embedded application via the shell. This simple dual logic in the :filter methods enables Gateway Agents to function as clients or servers, as required by particular messages.

KNOWL-GW also establishes (a) a uniform, top-level Gateway API; and (b) a uniform model for applying or invoking this API in the filter methods. Specifically, the KNOWL-GW Agent class establishes stub versions of the top-level Gateway API methods. Gateway Agent subclasses for specific development shells override the stubs with method definitions tailored to the corresponding knowledge model,and data and control interfaces. The top-level API consists of the following five methods:

- :extract-data;
- :inject-data;
- :initialize-shell;
- :process-request;
- :process-response.

The first three top-level API methods are defined for each Gateway Agent subclass in terms of intermediate-level DMS API functions, which differ in reflection of variations in shell architectures. However, each subclass API contains elements from the following categories:

- external interface functions;
- data interface functions (for data and knowledge access control);
- shell control functions.

The first API category encompasses shell-specific functions for passing data and commands from an application out to the Knowledge Gateway. Depending on the shell in question, the Gateway API may be more or less elaborate. For example, the Gateway for CLIPS V4.3 defines a single external API function to initiate interactions between rules in a CLIPS application and its embedding Agent, which hides MetaCourier and DMS API functions completely. The CLIPS Gateway API will be extended to reflect the procedural and object-oriented programming extensions in CLIPS Version 5.

Functions in the second category combine (a) the DMS API, which maps data and knowledge between SOCIAL's canonical DMS structures and the representational model native to a specific shell with (b) the shell-specific programmatic data interface used to generate, modify, and access data and knowledge structures in the native representational format. Examples include asserting and retracting structures, sending object-oriented messages, and modifying object attributes. For example, Make-CLIPS-fact-from-fact converts a DMS fact into a string, which is automatically inserted into the current CLIPS facts-list using the CLIPS assert function. The third

category encompasses shell-specific control interface capabilities such as start, clear, run, reset, exit, and saving and loading code and/or knowledge base files.

The top-level :extract-data and :inject-data Gateway methods consolidate the intermediate-level data interface API functions. Typically, :inject-data and :extract-data consist of program Case statements that invoke conversion functions for translating between different types defined in the native information model for a given shell and structures in the SOCIAL/DMS model. For example, :inject-data may call a DMS-level API function to map and insert a DMS frame structure as a fact-group into a CLIPS knowledge base and another to insert a DMS fact. Access direction (reading or writing) is implicitly reflected in the developer's choice of Extract (read) or Inject (write). Both methods are preprogrammed to dispatch automatically on data type, with options to override defaults (e.g., to map a DMS frame into a CLIPS fact-group instead of a deftemplate). Similarly, the :initialize-shell method represents the locus for control interface functions. Behavior is again classified by case and dependent on the target tool or program. For example, CLIPS employs different API functions to load textual and compiled knowledge bases.

The remaining pair of top-level Knowledge Gateway API methods, :process-request and :process-response, are application-specific. A shell-based application is integrated into a distributed system by specializing the Gateway Agent subclass for the relevant shell. Specialization here consists of overriding the stub versions of Process-request and :process-response inherited from KNOWL-GW and defining the required integration behaviors. Developers redefine these two methods by employing the generic API functions :extract-data, :inject-data, and :initialize-shell to pass information and control into and out of the target application via its associated shell.

The Gateway model is particularly powerful for integrating shell-based applications, in that the shell-specific methods (viz., :inject-data, :extract-data, :initialize-shell), are defined only once, namely in a KNOWL-GW subclass for the given shell. Application developers do not have to modify these API elements unless API extensions are necessary. Any application based on that shell can be embedded in a Gateway that is a subclass of the shell-specific Gateway Agent. The application Gateway inherits the generic tool-specific API interface, which means that the developer only has to program the methods :process-request (for server behaviors) and :process-response (for client behaviors). Individual interactions with the shell are specified using the inherited API to extract or inject particular data and to control the shell.

For custom applications, all five API methods are defined in one and the same Gateway Agent, namely the KNOWL-GW Agent subclass level. Therefore, inheritance does not play as powerful a role in assisting the application integrator as it does for multiple programs based on a common shell interface. Nevertheless, the generic :in-filter and :out-filter methods are inherited, providing the standardized message control model for peer-to-peer interactions. Moreover the Gateway model is useful as a methodological template in that it prescribes a uniform and intuitive partitioning of interface functionality: specific interactions between an application, its Gateway, and external systems are isolated in :process-request and :process-response, which invoke the utility API functions such as :inject-data as appropriate.

CLIPS Knowledge Gateway

CLIPS-GW is a subclass of the KNOWL-GW Agent class. As with all other Knowledge Gateway Agent subclasses, it inherits the KNOWL-GW message structure, :in-filter and :out-filter, and stub API methods. CLIPS-GW defines custom :inject-data, :extract-data, and :initialize-shell methods tailored to the CLIPS knowledge model, data and control interfaces. These custom methods are built up from a set of intermediate level API functions, which are summarized in Table 1. Specifically, :inject-data is based on Load-CLIPS-Data, which depends on CLIPS-Dispatch, and Load-CLIPS-Files. :extract-data relies on the function gw-return. :initialize-shell invokes the

basic shell control API functions, based on keyword symbols specified in incoming messages. Analogous APIs are defined for Knowledge Gateways for other AI shells, such as KEE.

Category	Function	Behavior
Shell Control	CLIPS-Start	starts CLIPS and sets a global flag
	CLIPS-Clear	clears all facts from CLIPS fact-list
	CLIPS-Init	if flag is set, calls CLIPS-Clear otherwise calls CLIPS-Start
	CLIPS-Run-Appl	runs CLIPS rule engine to completion accepts optional integer to limit # of rule firings
	CLIPS-Load-Appl	loads a specified rule base file into CLIPS
	CLIPS-Reset	asserts deffacts facts into CLIPS fact-list
	CLIPS-Assert	asserts a fact (string) into CLIPS fact-list\
	CLIPS-Retract	retracts fact (C pointer) from CLIPS fact-list
	CLIPS-Display-Facts	displays facts to output stream
Data Interface	CLIPS-Dispatch	calls a C dispatch routine to translate DMS structures and create CLIPS facts, fact-groups, deftemplates, or rules, as appropriate.
	Load-CLIPS-Data	reads DMS data from composite DMS structure and calls CLIPS-Dispatch on each one (except files)
	Load-CLIPS-Files	reads the composite DMS object for file pathname strings and calls CLIPS-Load-Appl
External Interface	gw-return	an external/user function defined to CLIPS for passing data from rules back to Agent

Table 1. Intermediate Level API for the CLIPS Gateway Agent (Version 4.3)

The CLIPS Gateway API defines an external user function that provides a high-level interface between a CLIPS application and its embedding Gateway . This function, called gw-return, enables CLIPS applications to pass data and/or control information to their Gateway Agents by stuffing a stream buffer that is unpacked using the top-level :extract-data command. gw-return function calls appear as consequent clauses, as illustrated in the example rule shown below. gw-return takes two arguments - a DMS structure type such as a Fact and a string or pointer. The first item is used to parse the datum and convert it into the specified type of DMS structure. Multiple gw-return clauses can be placed into the right-hand side of a single rule, Also, multiple rules can contain gw-return clauses.

```
(defrule TALK-BACK-TO-GATEWAY
   "rule that passes desired result, a fact, that has been asserted
    into appl KB back through the Gateway to requesting Agent"
      ?requestor <- (requestor ?appl-agent)
      ?answer  <- (answer $?result)
 =>
      (printout t "Notifying " ?requestor "of result" $?result" crlf)
      (gw-return FACT (str-implode $?result))
```

Messages to Knowledge Gateway Agents contain five elements, the target Agent, Environment, Host, data, and command options. In server mode (responding to messages from other application Agents), the CLIPS-GW :in-filter executes :initialize-shell for the specified command options to prepare CLIPS, invokes :process-request for the incoming data, and sets the results. Typically :process-request injects data, which includes loading rule bases, runs the rule engine, and extracts results. The :out-filter translates the :in-filter results into SOCIAL neutral exchange format, which constitute the reply that MetaCourier returns to the requesting Agent.

In client mode, a CLIPS application initiates a message to some external application Agent via its embedding CLIPS Agent. Here, the :in-filter invokes :extract-data and passes the message contents and any specified command options to the target application Agent. The :out-filter then invokes :process-response to deal with the reply. Typically, :process-response invokes :inject-data to introduce response data into the CLIPS fact-list and restarts the CLIPS rule engine to resume reasoning.

EXAMPLE APPLICATIONS OF CLIPS GATEWAY AGENTS

This section of the paper describes two demonstration systems that employ CLIPS Gateways to integrate expert systems for operations support for the Space Shuttle fleet. Processing, testing, and launching of Shuttle vehicles takes place at facilities dispersed across the Kennedy Space Center complex. The Launch Processing System (LPS) provides the sole direct, real-time interface between Shuttle engineers, Orbiter vehicles and payloads, and associated Ground Support Equipment to support these activities (Heard 1987). The locus of control for the LPS is the Firing Room, an integrated network of computers, software, displays, controls, switches, data links and hardware interface devices. Firing Room computers are configured to perform independent LPS functions through application software loads. Shuttle engineers use Console computers to monitor and control specific vehicle and Ground Support systems. These computers are connected to data buses and telemetry channels that interface with Shuttles and Ground Support Equipment. The Master Console is a computer that is dedicated to operations support of the Firing Room itself.

Integrating Configuration and Fault Management

The first application illustrates the use of a CLIPS Gateway in a server role to integrate expert systems that automate configuration and fault management operations support tasks (Adler 1990). X-Switcher is a prototype expert system that supports operators of the Switching Assembly used to manage computer configurations in Firing Rooms. X-Switcher was implemented using CLIPS V4.3 on a Sun workstation. OPERA (for Operations Analyst) is an integrated collection of expert systems that automates critical operations support functions for the Firing Room (Adler 1989). In essence, OPERA retrofits the Master Console with automated, intelligent capabilities for detecting, isolating and managing faults in the Firing Room. The system is implemented in KEE and runs on a Texas Instruments Explorer Lisp Machine. PRACA, NASA's Problem Reporting and Corrective Action database, was simulated using the Oracle relational DBMS, again on a Sun workstation.

A distributed system prototype was constructed with SOCIAL, using appropriate library Agents to integrate these three applications - a CLIPS Gateway for X-Switcher, a KEE Gateway for OPERA, and an Oracle Gateway for PRACA. The prototype executes the following scenario. First, OPERA receives LPS error messages that indicate a failure in a Firing Room computer subsystem. OPERA then requests a reconfiguration action from X-Switcher via the OPERA KEE Gateway Agent. This request is conveyed via a MetaCourier message to the CLIPS Gateway Agent. The message contains a DMS fact-group that specifies the observed computer problem, the pathname for the X-Switcher rule base on the Sun platform, and the current Firing Room Configuration Table. OPERA models the Configuration Table as a unit, which is KEE's hybrid frame-object knowledge structure. The OPERA Agent automatically unpacks slot data from the Table unit and appends it to the DMS fact-group via KEE Gateway API calls.

Upon receiving the KEE Gateway's message, the CLIPS Gateway Agent executes the following sequence of tasks. First, CLIPS is loaded, if necessary, and initialized. Second, the X-Switcher expert system rule base is loaded. Third, the DMS OPERA data object from the KEE Gateway message is translated and asserted as a CLIPS fact-group. Fourth, CLIPS is reset and the rule engine is run. X-Switcher rules derive a set of candidate replacement CPUs for the failed Firing

Room computer and prompt the user to select a CPU. It then displays specific instructions for reconfiguring the Switching Assembly to connect the designated CPU and prompts the user to verify successful completion of the switching activity. Finally, X-Switcher interacts with its CLIPS Gateway to reply to OPERA that reconfiguration of the specified CPU succeeded or failed. This process is triggered when CLIPS executes an X-Switcher rule containing a consequent clause of the form *(gw-return result)*. The CLIPS Gateway converts *result* into a DMS fact, which is transmitted to the OPERA KEE Gateway. This Agent asserts this fact as an update value in a subsystem status slot in the Configuration Table KEE unit. Finally, the OPERA Gateway formulates an error report, which is dispatched in a message to the Oracle Gateway Agent, which updates the simulated PRACA Problem-Tracking Database.

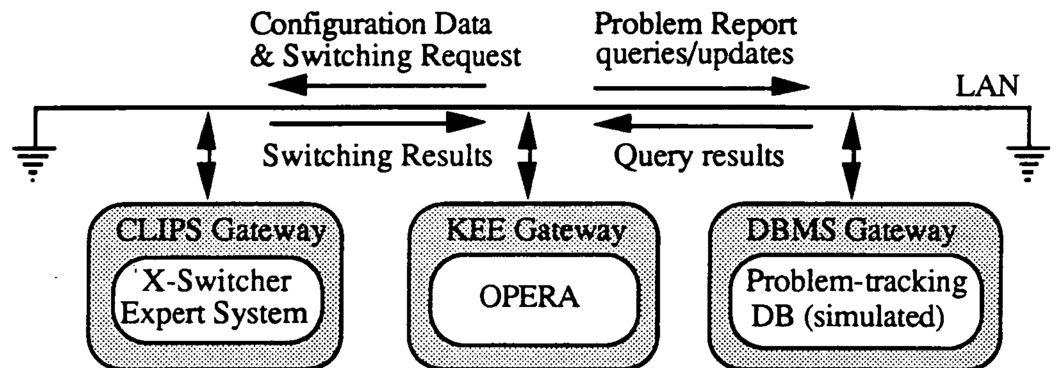


Figure 5. CLIPS Application configured as a Server

Coordinating Independent Systems to Enhance Fault Diagnosis Capabilities

The second distributed application illustrates the use of a SOCIAL CLIPS Gateway Agent in a client role (Adler 1991). GPC-X is a prototype expert system for isolating faults in the Shuttle vehicle's on-board computer systems, or GPCs. GPC-X was implemented using CLIPS V4.3 on a Sun workstation. One type of memory hardware fault in GPC computers manifests itself during switchovers of Launch Data Buses. These buses connect GPCs to Firing Room Console computers until just prior to launch, when communications are transferred to telemetry links. Unfortunately, the data stream that supplies the GPC-X expert system does not provide any visibility into the occurrence of Launch Data Bus switchovers (or the health of the GPC Console Firing Room computer). Thus, GPC-X can propose but not test certain fault hypotheses about GPC problems, which seriously restricts the expert system's overall diagnostic capabilities.

However, Launch Data Bus switchover events are monitored automatically by the LPS Operating System, which triggers warning messages that are detected and processed by the OPERA system discussed above. CLIPS and KEE Gateway Agents were used to integrate GPC-X and OPERA, as before. A SOCIAL Manager Agent was used to mediates interactions between these application Gateway Agents to coordinate their independent fault isolation and test activities.

Specifically, GPC-X, at the appropriate point in its rule-based fault isolation activities, issues a request via its embedding Agent to check for Launch Data Bus switchovers to the Manager. The request is initiated by a gw-return consequent clause in the CLIPS rule that proposes the memory fault hypothesis. When this rule fires, CLIPS executes the gw-return function, which sends a message to the GPC-X CLIPS Gateway Agent. This Agent formulates a message to the Manager which contains a Manager API task request for the LDB-Switchover-Check service.

The Manager searches its directory for an appropriate server Agent for LDB-Switchover-Check, reformulates the task data into a suitable DMS-based message, and passes it to the OPERA KEE Gateway Agent. The :process-request method for this application Agent performs a search of the knowledge base used by OPERA to store interpreted LPS Operating System error messages. The objective is to locate error messages, represented as KEE units, indicative of LDB switchover events. The OPERA Gateway :out-filter uses the Manager API to translate search results into a suitable DMS structure, which is posted back to the Manager. In this situation, the OPERA Gateway Agent contains *all* of the request processing logic: OPERA itself is a passive participant that continues its monitoring and fault isolation activities without significant interruption.

Next, the Manager returns the results of the LDB-Switchover-Check request back to GPC-X's CLIPS Gateway Agent. The Agent :in-filter executes the :process-response method, which transparently converts the Manager DMS object into a CLIPS fact that is asserted into the GPC-X fact base. Finally, the GPC-X Agent re-activates the CLIPS rule engine to complete GPC fault diagnosis. Obviously, new rules had to be added to GPC-X to exploit the newly available hypothesis test data. However, all of the basic integration and coordination logic is supplied by the embedding GPC-X Gateway Agent or the HDC-Manager.

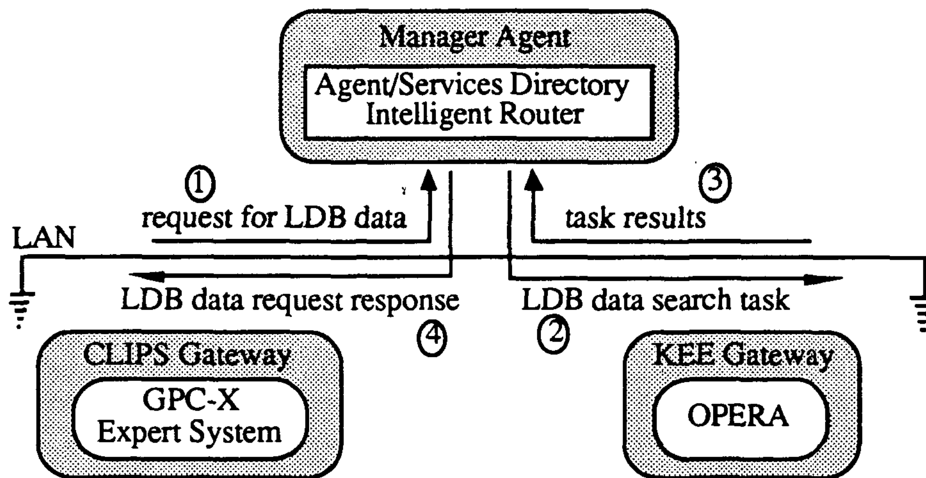


Figure 6. CLIPS Gateway configured as a client

This SOCIAL prototype demonstrates non-intrusive *system-level* coordination of distributed applications that solve problems at the subsystem level. Neither application is capable of full diagnosis individually. GPC-X can generate GPC fault candidates, but lacks data concerning other LPS subsystems that is necessary for testing these hypotheses. OPERA automatically detects LPS error messages that are relevant to GPC-X's candidate test requirements. However, it lacks the contextual knowledge about GPC computers, and awareness of GPC-X's capabilities and current activities, to recognize the potential significance of specific LPS data as a test for GPC-X fault hypotheses. Gateway Agents integrate the two systems, supplying communication and data mapping capabilities. The Manager establishes the logical connections required to combine and utilize the fragmented subsystem-specific knowledge of the two applications to enhance diagnostic capabilities. This coordination architecture is non-intrusive in that neither system was modified to include direct knowledge of the other, its interfaces, knowledge model, or platform. The Manager directory and routing capabilities introduce an isolating layer of abstraction, enhancing the "plug-compatible character of the integration architecture.

RELATED WORK

The most closely related research to SOCIAL CLIPS Gateway Agents is the AI Bus (Schultz 1990), a framework for integrating rule-based CLIPS applications in a distributed environment. SOCIAL and AI Bus both rely on modular message-based communications. AI Bus uses a client-server model based on remote procedural calls that is currently restricted to Unix hosts. SOCIAL's MetaCourier layer supports a fully peer-to-peer model that is transparent across diverse platforms. SOCIAL and AI Bus integrate applications using Agents, whose API functionality are roughly comparable. Each Agent has a dedicated message control module and can communicate directly with one another. Indirect interactions are mediated by a dedicated organizational Agent, the SOCIAL Manager or the AI Bus Blackboard. It appears that AI Bus Agents are currently restricted primarily to CLIPS-based knowledge sources, while SOCIAL Gateways provide broader support for KEE, CLIPS, and other tool-based and custom applications.

Other tools for developing heterogeneous distributed intelligent systems include GBB (Corkill 1986), ERASMUS (Jagannathan 1988), MACE (Gasser 1987), and ABE (Hayes-Roth 1988). These systems lack SOCIAL's modular, layered, architecture, and are considerably less extensible below the top-level developer interfaces. GBB and ERASMUS impose a blackboard control architecture for integrating distributed applications. ABE allows multiple kinds of interaction models (e.g., transaction, data flow, blackboard), but it is not clear how easily these can be combined within a single system. MACE provides few organizational building blocks for developing complex architecture beyond a relatively simple routing Agent. None of these frameworks provide a predefined integration interface to CLIPS, although ABE and GBB include simple "black box" tools such as external or foreign function call passing to build one.

STATUS AND FUTURE DEVELOPMENT

The original CLIPS Gateway Agent was implemented for CLIPS V4.3 in Franz Common Lisp, using a foreign function interface to the CLIPS API, which is written in C. Within the next year, we intend to develop a full C implementation of the Agent. This Agent will also be extended to reflect enhancements in CLIPS V5.0, most notably, procedural programming and the CLIPS Object-Oriented Language.

CONCLUSIONS

CLIPS was designed to facilitate embedding intelligent applications within more complex systems. However, lacking built-in support for distributed communications capability, applications implemented with CLIPS are generally "hardwired" directly to other software systems residing either on the same platform or on a parallel multi-processor. Moreover, CLIPS integration interfaces are typically custom-built, by systems level programmers who are experienced with the mechanics of interprocess communication. SOCIAL CLIPS Gateway Agents provide a generalized, high-level approach to integrating CLIPS applications with other intelligent and conventional programs across heterogeneous hardware and software platforms. Gateways exploit object-oriented inheritance to partition generic distributed communication and control capabilities, shell-specific APIs, and application-specific functionality. Developers need only learn the high-level APIs to integrate CLIPS applications with other application Agents. SOCIAL's modular and extensible integration technologies promote a uniform, "plug compatible" model for non-intrusive, peer-to-peer interactions among heterogeneous distributed systems.

ACKNOWLEDGMENTS

Development of SOCIAL, including the CLIPS Gateway Agent, was sponsored by the NASA Kennedy Space Center under contract NAS10-11606. MetaCourier was developed by Robert Paslay, Bruce Nilo, and Robert Silva, with funding support from the U.S. Army Signals Warfare

center under Contract DAAB10-87-C-0053. Rick Wood designed and implemented the C portions of the CLIPS Gateway API. Bruce Cottman developed the prototypes for SOCIAL's data management tools and the Oracle Gateway Agent.

REFERENCES

Adler, R.M. (1991). A Hierarchical Distributed Control Model for Coordinating Intelligent Systems. *Proceedings, 1991 Goddard Conference on Space Applications of Artificial Intelligence.* NASA CP-3110. pp. 183-198.

Adler, R.M. and Cottman, B.H. (1990). EXODUS: Integrating Intelligent Systems for Launch Operations Support. *Proceedings, Fourth Annual Workshop on Space Operations, Applications, and Research Symposium (SOAR'90).* NASA CP-3103. pp. 324-330.

Adler, R.M., Heard, A., and Hosken, R.B. (1989). OPERA - An Expert Operations Analyst for A Distributed Computer Network. *Proceedings, Annual AI Systems in Government Conference.* IEEE Computer Society Press. Washington, DC. pp. 179-185.

CLIPS Reference Manual. (1989).Version 4.3. Artificial Intelligent Section, Johnson Space Center, Houston, TX.

CLIPS Reference Manual. (1991). Version 5.0. Software Technology Branch, Johnson Space Center, Houston, TX.

Corkill, D.D., Gallagher, K.Q., and Murray, K. (1986). GBB: A Generic Blackboard Development System. *Proceedings Fifth National Conference on Artificial Intelligence.* pp. 1008-1014.

Gasser, L., Braganza, C., and Herman, N. (1987). MACE: A Flexible Testbed for Distributed AI Research. in *Distributed Artificial Intelligence Vol. 1.* M. Huhns (Ed.) Morgan Kaufmann. Los Altos, California, 1987.

Hayes-Roth, F., Erman, L.D., Fouse, S., Lark, J.S., and Davidson, J. (1988). ABE: A Cooperative Operating System and Development Environment. in A. H. Bond and L. Gasser, (Eds.) *Readings in Distributed Artificial Intelligence.* Morgan-Kaufmann. Los Altos, CA.

Heard, A.E. (1987). The Launch Processing System with a Future Look to OPERA. *Acta Astronautica,* IAF-87-215.

Jagannathan, V., Dodhiawala, R., and Baum, L. (1988). The Boeing Blackboard System: The Erasmus Version. *International Journal of Intelligent Systems.* vol. 3. no. 3. pp. 281-294.

Schultz, R.D., and Stobie, I.C. (1990). Building Distributed Rule-Based systems Using the AI Bus. *First CLIPS Conference Proceedings.* NASA CP-10049. pp.676-685.

Symbiotics, Inc. (1990). Object-Oriented Heterogeneous Distributed Computing with MetaCourier. Technical Report. Symbiotics, Inc. Cambridge, MA.

SESSION 6 A

PRECEDING PAGE BLANK NOT FILMED

N92-16598

DATA-DRIVEN BACKWARD CHAINING

Paul Haley
The Haley Enterprise, Inc.
413 Orchard Street
Sewickley, PA 14153
USA
(412) 741-6420

Abstract: CLIPS cannot effectively perform sound and complete logical inference in most real-world contexts. The problem facing CLIPS is its lack of goal generation. Without automatic goal generation and maintenance, Forward chaining can only deduce all instances of a relationship. Backward chaining, which requires goal generation, allows deduction of only that subset of what is logically true which is also relevant to ongoing problem solving.

Goal generation can be mimicked in simple cases using forward chaining. However, such mimicry requires manual coding of additional rules which can assert an inadequate goal representation for every condition in every rule that can have corresponding facts derived by backward chaining. In general, for N rules with an average of M conditions per rule the number of goal generation rules required is on the order of N*M. This is clearly intractable from a program maintenance perspective.

We describe the support in Eclipse for backward chaining which automatically asserts as it checks rule conditions. Important characteristics of this extension are that it does not assert goals which cannot match any rule conditions, that 2 equivalent goals are never asserted, and that goals persist as long as, but no longer than, they remain relevant.

Introduction

Suppose we were developing an application concerning genetically transmitted traits. Our application might need several rules that guided its reasoning. One such rule might be, "if a person has a trait and a cousin of that person has the same trait, then consider the possibility that the trait is inherited." Such a rule might be coded as follows:

```
(defrule cousins-may-inherit-trait
     (has ?grandchild-1 ?trait)
     (parent ?grandchild-1 ?parent-1)
     (parent ?parent-1 ?grandparent)
     (parent ?parent-2 ?grandparent)
     (parent ?grandchild-2 ?parent-2)
     (has ?grandchild-2 ?trait)
     =>
     (assert (inherited (status possible) (trait ?trait)))
     )
```

This is a fine rule when viewed in isolation. However, there are probably lots of rules in this application that examine conditions across siblings. All of these rules will share conditions similar to:

```
(parent ?parent-1 ?grandparent)
(parent ?parent-2 ?grandparent)
```

This amounts to a low-level encoding of the notion of a sibling. The following conditions amount to a low-level encoding of the notion of a cousin:

```
(parent ?grandchild-1 ?parent-1)
(parent ?parent-1 ?grandparent)
(parent ?parent-2 ?grandparent)
(parent ?grandchild-2 ?parent-2)
```


PRECEDING PAGE BLANK NOT FILMED

In an application of hundreds of rules that consider blood relationships in many different way and combinations, having notions of "sibling" and "cousin" available as simple relationships rathe than as more complex pattern matching operations not only makes the rules more perspicuous, i makes them more reliable and easier to maintain. As an example, the above rule could be recode as:

```
(defrule cousins-may-inherit-trait
     (has ?x ?trait)
     (cousin ?x ?y)
     (has ?y ?trait)
     =>
     (assert (inherited (status possible) (trait ?trait)))
     )
(defrule cousin
     (parent ?x ?parent-1)
     (sibling ?parent-1 ?parent-2)
     (parent ?y ?parent-2)
     =>
     (assert (cousin ?x ?y))
     )
(defrule sibling
     (parent ?x ?parent)
     (parent ?y&~?x ?parent)
     =>
     (assert (sibling ?x ?y))
     )
```

Deduction using Forward Chaining

The cousin and sibling rules above make the high-level semantics of cousin and sibling explicit while in the original rule they were implicit. With these relations made explicit, coding of all rules that consider these relations can use a single pattern rather than its corresponding, implicit, constituent patterns.

Reducing the number of patterns per rule clearly improves the reliability of those rules. Also, maintaining rules that use the more abstract patterns is simplified since only the rules that maintain the relevant relation need to be modified. Furthermore, if a relation can be deduced by any of several methods (i.e., disjunction occurs) then the number of rules is reduced with resulting improvements in performance and reliability. Finally, using relations rather than unshared joins over several patterns can dramatically improve performance and reduce space requirements.

The problem with the above sibling and cousin rules is that they will assert every cousin and sibling relationship that exists given a set of parent relationships. The number of these deduced relationships can become very large, especially for the cousin relationship.

This is a fundamental problem. For most domains, there are at least an infinite number of irrelevant truths that can be deduced. The challenge in building a rational problem solving system is to actually deduce truths that are (or have a good chance of being) relevant to the problem at hand.

Deduction using Backward Chaining

Focusing deduction such that it furthers problem solving, rather than merely deducing irrelevant truths, is often done by generating subgoals during problem solving. Goals are generated as the conditions of rules are checked. These goals then trigger the checking of rules that might deduce facts that would further the matching of the rule which generated the goal.

To be more concrete, the cousin and sibling rules from above could be recoded as:

```
(defrule cousin
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y ?p2)
     (sibling ?p1 ?p2)
     =>
     (assert (cousin ?x ?y))
     )
(defrule sibling
     (goal (sibling ?x ?y))
     (parent ?x ?parent)
     (parent ?y ?parent)
     =>
     (assert (sibling ?x ?y))
     )
```

In these rules the goal condition is triggered when a goal to establish a sibling relationship is generated. The actions of these rules assert facts which satisfy the goals, thereby deducing only facts which might further the matching of the rules which led to the goals' generation.

We call the above rules data-driven backward chaining rules. Of course, for these rules to be driven some goal data is required. Either other rules or the inference engine architecture itself must assert these goals. In either case, goals must be generated as if by the following rules:

```
(defrule cousins-may-inherit-trait-goal-generation-1
     (has ?x ?trait)
=>
     (assert (goal (cousin ?x ?y)))
     )
(defrule cousin-goal-generation-1
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y&~?x ?p2)
     =>
     (assert (goal (sibling ?p1 ?p2)))
     )
```

Manual Goal Generation

The above goal generation rules, if they could be implemented, would correctly generate the goals required to implement the explicit cousin and sibling relations using the previously mentioned data-driven backward chaing rules. However, beyond the need for an adequate representation for goals, the manual coding of goal generation rules would remain problematic.

In general, for a rule of N conditions, N+1 rules will be needed to implement those rules such that they can support sound and complete reasoning. The original rule which matches in the standard, data-driven, forward chaining manner is, of course, required. An additional rule per condition is needed to assert the goals that will allow backward chained inference to deduce facts that will further the matching of the original rule.

Clearly, multiplying the number of rules required by one plus the average number of goal generating conditions per rule is unacceptable. Even if the effort is made, it is extremely error prone. Even automating the maintenance of goal generation rules would increase space and time requirements significantly, just to encode the actions and names of the rules and to activate the rules and interpret their actions.

Representing Goals

Even though manual coding of goal generation is impractical, CLIPS, OPS5, and many other production system languages are unable to implement the above rules for several even more fun-

damental reasons. The most obvious reason is that they provide no capability for distinguishing facts from goals. Moreover, these systems provide no means of representing unspecified values (for unbound variables) that occur in the conditions for which they might otherwise assert goals. For example, the following goal generation rule, in which the variable ?y is unbound, cannot even be simulated without explicit support for goals which include universally quantified values:

```
(defrule cousins-may-inherit-trait-goal-generation-1
     (has ?x ?trait)
=>
     (assert (goal (cousin ?x ?y)))
     )
```

Even supporting universally quantified values within goals is not enough to support backward chaining, however. If the variable ?y in the first condition of the following rule matches a literal value, CLIPS or OPS5 extended to support goal generation could function properly. If, however, ?y matches a universally quantified value, then neither CLIPS or OPS5 could join that unbound variable with any parent fact corresponding to the third condition, as would be logically required.

```
(defrule cousin-goal-generation-1
     (goal (cousin ?x ?y))
     (parent ?x ?p1)
     (parent ?y&~?x ?p2)
     =>
     (assert (goal (sibling ?p1 ?p2)))
     )
```

Clearly these systems are unable, not only to generate goals in the first place, but also to join those goals with facts.

Automatic Goal Generation

Eclipse is a syntactically similar language to NASA's CLIPS and Inference Corporation's ART, each of which include functionality similar to that of OPS5. Unlike CLIPS and OPS5, however, Eclipse supports a goal database and automatic generation of goals. In fact, the above pseudo-CLIPS rules which reference goals in their conditions and which assert facts are legal Eclipse rules. However, Eclipse does not require the addition of goal generation rules.

Eclipse automatically asserts goals precisely as would the goal generation rules described earlier. Goal generation in Eclipse adds no scheduling or interpretive overhead. There is no space overhead per rule or condition that generates a goal. Moreover, Eclipse goals can represent and include universally quantified (or unbound) values. Eclipse also supports the unification of universally quantified values with literals that occur in facts.

Goals as Data

Procedural backward chaining languages, such as Prolog, do not represent goals as data. In Prolog, goals are equivalent to procedure calls, if an invoked goal procedure fails the goal cannot be achieved. Moreover, in Prolog, if a goal fails at the time it is initially pursued, it will not be achieved unless a new and equivalent goal is reestablished. Thus, Prolog would fail to deduce a sibling relationship if a goal were established before a relevant parent relationship were known.

In Eclipse, goals are represented as propositions in a database. By representing goals as data and allowing patterns to distinguish facts from goals, Eclipse allows goals to drive pattern matching in combination with facts in the normal, data-driven manner. By representing goals in a database several goals can exist simultaneously and each goal can persist even if - at the time it is generated - it cannot be achieved. The simultaneous existence of multiple goals allows rules to do strategic reasoning and planning which is not possible if only one goal at time can be considered. The

persistence of goals allows goals to be achieved opportunistically. Unlike Prolog, if Eclipse generates a sibling goal which cannot be established, subsequent assertion of a relevant parent relationship would result in proper deduction.

Goal Maintenance

Eclipse also supports truth maintenance. In its standard application, truth maintenance allows a fact to be given *a prior* and/or supported by a disjunction of facts or sets of data which satisfy all or part of the conditions of one or more rules. For example, the following rule would make fact C logically dependent on the fact A and the absence of fact B[1]:

```
(defrule A-and-not-B-implies-C (A) (not (B)) => (infer (C)))
```

Subsequent retraction of A or assertion of B would lead to the retraction of the match for rule A-and-not-B-implies-C which would support the inference of C. If this support is removed, C (which we assume has not been asserted without logical dependency) would no longer be logically grounded and would therefore be automatically retracted.

In effect, Eclipse goal generation behaves as the goal generation rules described earlier using these logical dependencies. That is, if the following rules lead to the generation of a (goal (D)) after the assertion of A:

```
(defrule A-and-D-implies-C (A) (D) => (infer (C)))
(defrule D-is-implied-by-A (goal (D)) (A) => (infer (D)))
```

and A is subsequently retracted, the (goal (D)) will also be automatically retracted.

Using dependencies on goals results in a number of functional advantages. The most immediately obvious advantage is that goals only persist as long as they are relevant. This is another advantage over attempting to assert and maintain a crude representation of goals using OPS5 or CLIPS which do not support such dependencies. Secondly, if the facts inferred by goals are made to depend on the continued existence of those goals, then facts which were deduced but which subsequently become irrelevant to the ongoing process of problem solving are automatically retracted from the database. The automatic maintenance of deductions versus goals frequently improves performance and certainly reduces the need for manual coding of "cleanup" rules.

Goal Canonicalization

It is common for several rules to generate equivalent goals. For example, the following rules would both generate (goal (C ?1)) given facts A and B[2]:

```
(defrule A-and-C (A) (C ?x) =>)
(defrule B-and-C (B) (C ?x) =>)
```

Just as asserting equivalent facts twice results in one fact, so does the generation of two goals from these two rules result in only one goal. This one goal will have two sources of support. Removing one source of support will not result in the automatic retraction of the goal. Eclipse automatically maintains goals only as long as they are relevant by allowing a goal to persist only as long as at least one of the reasons for its generation persists.

An Example of Opportunistic Forward and Backward Chaining in Eclipse

In what follows we give an extensive trace and explanation of the simple genetic trait rules discussed earlier.

---

1 Under the closed-world assumption, absence of a fact is equivalent to a fact being false.

2 The ?1 denotes the first universally quantified value in a goal.

Given an empty database, first assert that John has freckles:

```
==> f-1 (has John freckles)
```

The above fact matches the first condition of the first rule. This causes a goal for the second condition of that rule, given John, to be generated as follows:

```
==> g-1 (cousin John ?1) by for 1 of cousin-may-inherit-trait f-1
```

This goal in turn matches the first condition of the cousin rule. However, since we do not know either of John's parents, the rule cannot apply... yet. Unlike Prolog and other procedural backward chaining languages, using a declarative representation for goals (rather a function-call semantics) allows the goal to persist in a database. In fact, representing goals as data allows an application to consider multiple goals that exist in the database rather than focus solely and ignorantly on only the most recently generated goal pushed onto a stack.

At this point then, a fact and a goal exist in the database. By asserting one of John's parents, problem solving can continue in the light of the established and outstanding goal.

```
==> f-2 (parent John George)
```

This parent fact matches the second pattern of the cousin rule and satisfies the mutual occurrence constraint on "John" given the outstanding goal. This leads to a match for the first two conditions:

```
==> thru 2 of cousin g-1,f-2
```

which results in a goal to establish facts which satisfy the third condition of the cousin rule given John's parent, George:

```
==> g-2 (sibling George ?1) by thru 2 of cousin g-1,f-2
```

Again, none of George's parents are known so problem solving cannot proceed. Note that at this time two facts about John having freckles and his father, George, and two goals exist in the database. By establishing one of George's parents, problem solving gets a little bit further:

```
==> f-3 (parent George Adam)
```

This fact, given the goal to determine George's siblings, matches the first two conditions of the sibling rule.

```
==> thru 2 of sibling g-2,f-3
```

Again, problem solving cannot proceed since no other offspring of Adam are known. If we also identify Sally as a child of Adam's:

```
==> f-4 (parent Sally Adam)
```

then the third and final condition of the sibling rule is satisfied for George, Sally, and their mutual parent, Adam.

```
==> thru 3 of sibling g-2,f-3,f-4
==> activation 0 sibling g-2,f-3,f-4
```

Executing this rule asserts the sibling relation between George and Sally:

```
==> f-5 (sibling George Sally)
```

which in turn satisfies the goal and matches condition 3 of the cousin rule.

```
==> thru 3 of cousin g-1,f-2,f-5
```

Once again, problem solving halts, however, since no children of Sally are known. By asserting a child for Sally,

```
==> f-6 (parent Mary Sally)
```

the fourth and last condition of the cousin rule is satisfied for John given his father George, George's sister Sally, and Sally's daughter, Mary.

```
==> thru 4 of cousin g-1,f-2,f-5,f-6
==> activation 0 cousin g-1,f-2,f-5,f-6
```

Executing this rule asserts the cousin relationship between John and Mary:

```
==> f-7 (cousin John Mary)
```

which matches the second condition of the first rule and joins with the fact that John has freckles to match the first two conditions of that rule:

```
==> thru 2 of cousins-may-inherit-trait
```

From which point, asserting that Mary has freckles activates the rule, thereby leading to the assertion the freckles may be inherited.

Conclusion

By supporting automatic generation and maintenance of goals across multiple rules with many potential causes for each goal to exist and by allowing each goal to represent literal and universally quantified values which constrain the facts that could satisfy any given goal to a subset of all possible facts which has a higher probability of furthering problem solving, Eclipse allows data-driven rule technology, which is the only practical technology for rule-based programming and the implementation of expert systems, to perform logical reasoning. Moreover, the software engineering, maintenance, extensibility, and performance characteristics of rule-based programs afforded by reduced coding and interpretation of redundant, potentially disjunctive, combinations of patterns within many rules is also considerable. Finally, the resulting programs are simply much easier to understand since they make explicit the high-level knowledge which would otherwise be encoded implicitly using more complicated, less perspicuous, less efficient, and less maintainable combinations of patterns across many rules.

N92-16599

AUTOMATED INFORMATION RETRIEVAL USING CLIPS

Rodney Doyle Raines III
United States Coast Guard
Department of Computer Science
Cal Poly, San Luis Obispo

James Lewis Beug
Department of Computer Science
Cal Poly, San Luis Obispo

Abstract. Expert systems have considerable potential to assist computer users in managing the large volume of information available to them. One possible use of an expert system is to model the information retrieval interests of a human user and then make recommendations to the user as to articles of interest. At Cal Poly, a prototype expert system written in CLIPS serves as an Automated Information Retrieval System (AIRS). AIRS monitors a user's reading preferences, develops a profile of the user, and then evaluates items returned from the information base. When prompted by the user, AIRS returns a list of items of interest to the user. In order to minimize the impact on system resources, AIRS is designed to run in the background during periods of light system use.

THE INFORMATION RETRIEVAL PROBLEM

Introduction

A potentially important source of information available to both industrial and university researchers is electronic news. Like its printed counterparts, electronic news consists of a set of articles on various topics. Apart from the obvious difference of medium of distribution, electronic news differs from printed news in that it covers a wide range of topics, has typically non-professional contributors and a much greater volume of information. The volume of electronic news at Cal Poly is estimated to be over 50 megaBytes per day. Data gathered by the Network Measurement Project at the DEC Western Research Laboratory in Palo Alto (Reid 1991), California, show the following for a sample of 675 news sites for July 1991.

Average traffic per day (*megabytes*):	12.189
Average traffic per day (*messages*) :	5674

Clearly, a single user cannot read all the electronic news available, nor in fact are all articles relevant to a given user. Furthermore, much of what is distributed via electronic news is redundant.

Reading electronic news

In order to make use of the news, the user must have a strategy for selecting a relevant subset of what is available each day. The news itself is partitioned into news groups consisting of articles usually related to some common topic, such as artificial intelligence. Current news readers (programs for reading the news) provide several strategies:

- Articles may be read from selected news groups in an order determined by the user.
- Articles within a selected news group can be read in the order received.
- Articles which are related (a thread) may be read in order.

Additionally, a user may choose to display a list of article headers or subject lines, choosing articles to be read from that list. In any case, typically the user is responsible for determining which articles are relevant to his or her information needs.

The Problem

The major problem with these approaches is that the user will have to read more articles in selected groups than necessary to find information of interest (low relevance) and may miss information in news groups not typically read (low precision). In fact, electronic news shares many of the same problems which have been addressed by both the information storage and retrieval (IS&R) and library communities. Typically however, IS&R systems are expected to retrieve either exactly (neither fewer or more) the relevant documents published previously or, in the case of current awareness systems currently. For current awareness systems, an information specialist will prepare a profile of user interests, which is then used to route selected journals and/or articles to the reader. Complicating the issue is the fact that news articles tend to be very time sensitive and granular. Electronic news is time sensitive in content, news tends to deal with current issues which change rapidly, as well as space, news is not retained in readily accessable storage at Cal Poly for more than a week. The work described in this paper consists of using a CLIPS based expert system for automating information retrieval. Such a system should outperform the unaided reader, be adaptive to his or her changing information needs, and be extensible to other information systems such as electronic mail or messaging systems.

Related Work

Improving the performance of information retrieval systems is an area of research which falls into two general areas: improving the quality of the stored information and improving the performance of the user. Artificial Intelligence techniques have been used in both areas. Efforts to improve the quality of stored information can be categorized by three main approaches:

- Representing documents and search terms as an associative network
- Using natural language techniques to select index items
- Building a knowledge base from the document's contents

The system that Wyle and Frei have developed for managing network news is an example of improving the quality of stored information. They assume a passive user and perform algorithmic information gathering, filtering and dissemination (Wyle&Frei 89). This system lacks the sophistication provided by using an expert system to interact with the users profiles as we have developed.

Much of the use of expert systems in information processing has concentrated on determining a user's needs and then directing the appropriate information to the user. Gauch has recently developed an expert system which assists a user in searching through full-text knowledge-bases. Her system works with syntactically and semantically unprocessed text and uses a domain independent search strategy to present passages to a user in decreasing order of relevancy (Gauch 91). The system also does not deal with the entire information base; instead it sends out selected queries to database servers acting as an intermediary for the user.

The SCISOR project at MIT (Jacobs&Rau 91) has successfully demonstrated using natural language processing techniques to extract relevant information from online financial news. Much of this success can be attributed to its narrow domain which belies a lack of extensibility.

The LENS program at MIT sorts and prioritizes E-mail, based on importance and urgency. This system also suggests a course of action to the user and can automatically respond to some messages (Robinson 91). The initial implementation of the Automated Information Retrieval System (AIRS), running under *arn - adaptive read news*, concentrates on determining the user's needs and then directing information to the user. Later iterations will attempt to improve the quality of stored information.

AUTOMATED INFORMATION RETRIEVAL SYSTEM (AIRS) PROTOTYPE

To assist a user in sorting through the tremendous volume of information available at a computer center, we are developing an intelligent information management system. The heart of this management system is an Automated Information Retrieval System (AIRS). We chose to develop this system using a rapid prototyping methodology. At the onset, we established several objectives for the project. Our general objectives were to develop an environment for research into: the use of artificial intelligence in information retrieval; the application of traditional software engineering techniques to expert system development and the exploration of distributed computing in an expert system environment. More specific objectives are:

Programming Objectives

- Scalability to large databases
- Timely implementation
- Code reuseability
- Modularity
- Exportability to other computers
- Extensibility to other applications

User Interface Objectives

- Ease of use
- Transpancy to the user

To support these objectives, we chose CLIPS as the programming language. As a rule-based system, CLIPS readily adapts to a rapid prototype. An additional advantage is that the prototype need not be disposed of after the requirements specifications have been validated. Instead, the expert system can be reused after the prototype is finished, meeting yet another objective. CLIPS readily lends itself to modularity and scalability In addition, it is easy to learn. The availability of the source code, coupled with the wide base of computers already using CLIPS, makes it easily exportable to other computer systems.

The NeXT computer (NeXTstations), was selected for the initial implementation. A NeXTcomputer running the User Interface Builder provides a good rapid prototyping platform for Graphic User Interfaces (GUI's), which meets another of our objectives. The built-in primitives within the Mach operating system provide tools for distributed computing and easily allow for CLIPS to be run in a client/server configuration. Perhaps most important, the NeXT is very reasonably priced, providing more bang for the research buck.

AUTOMATED INFORMATION RETRIEVAL SYSTEM (AIRS) DESIGN

Overview

The Automated Information Retrieval System (AIRS) is just one component of a newsreader system. All articles are stored in a hierarchal file structure which we refer to as the newsbase. A user retrieves articles from the newsbase using arn - adaptive read news. Within arn, there are three main components:

- The Newsreader *(tass)*
- The search engine *(lqtext)*
- The Automated Retrieval System *(AIRS)*

An overview of the newsreader system is provided in Figure 1. The newsreader, *tass,* is a thread-based newsreader available via the Internet (Skrenta 1990). *tass*, like most newsreaders, allows a user to select newsgroups of interest, and then to browse through the newsgroups for articles of interest. The necessity of selecting newsgroups of interest is due to the sheer volume of news which must be otherwise eliminated. In addition to reading by newsgroups, *tass* allows a user to consider news threads, which are articles grouped by themes, within a newsgroup. Articles of interest contained in newsgroups not selected are never considered.

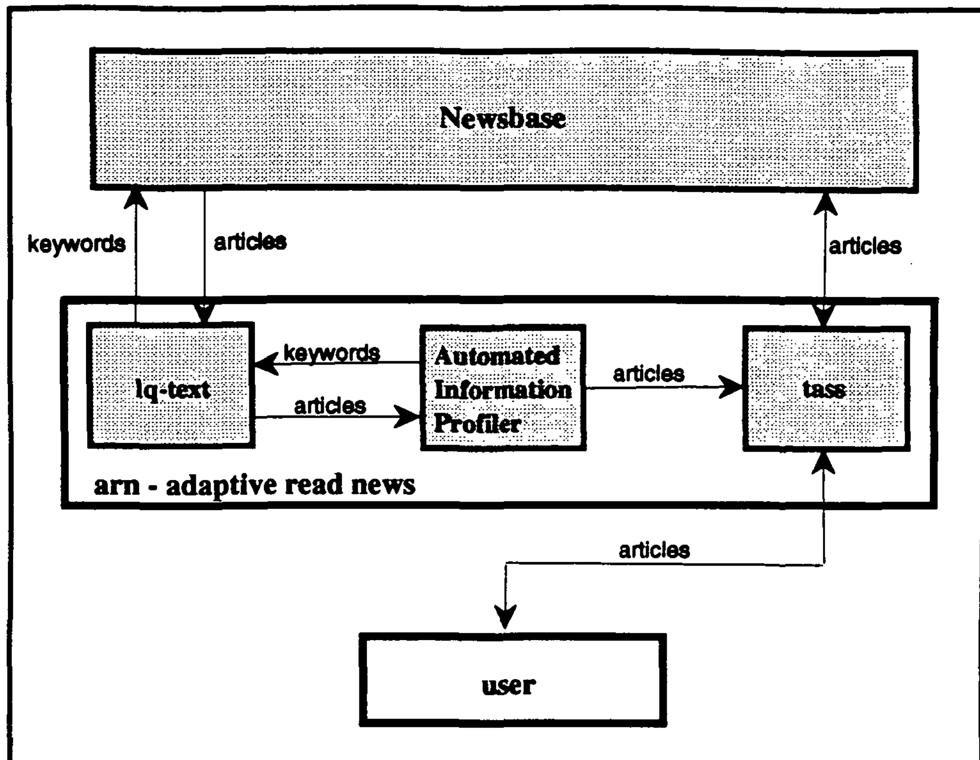


Figure 1. An overview of the newsreader system

To support a user who may need to locate information of interest in newsgroups other than those normally selected, a search engine component has been added to the newsreader system. The search engine component selected for the prototype newsreader is *LQ-text* (Quam 1990). Although not a true Boolean search engine, *LQ-text* allows for both keyword and keyphrase searches of a database. Using the search engine component, a user can compose a search of the newsbase and attempt to locate news of interest. This method of searching the newsbase is very labor intensive and does not lend itself to browsing. To alleviate this, an expert system component has been added to the newsreader, to compose and evaluate searches of the newsbase. This is the Automated Information Retrieval System (AIRS) component.

AIRS is the key component of the newsreader system. Written in the CLIPS expert system shell, it develops a profile of the user's news interests and disinterests. The profile is developed by monitoring the user's reading preferences at several levels. At the highest level, groups read by a user, both past and present, are considered by AIRS. Newsgroup information is obtained by examining the .newsrc file created by the newsreader. Historical information is kept in the .newsclps file. Information as to threads which have been read provide information as to the interests of the user, and threads which are "killed" provide information as to the disinterests of the user. Articles which are read or killed provide even more specific information as to the user's interests. Finally, a history of user-generated searches over the newsbase provides very detailed information as to the interests of the user. The various files containing this information relating to the user are parsed by AIRS, and a profile of user's interests, established by examining keywords, is created.

Once the user profile is created, the newsbase is searched for information which matches the user profile. This is done by sending queries to the search engine. The search tactics heuristics are used to improve query performance. Articles of interest returned by the search engine are evaluated and if necessary revised and sent back to the search engine. Once the information search process is complete, the findings are evaluated, ordered and presented to the user for reading via the newsreader. By using the newsreader, the user can continue to indicate their interests and the profile is continuously updated.

Rule Base Design

As discussed by Mehrotra and Johnson, we agree that rule base design is an important factor in the scalability of an expert system. Their work in using an expert system to automate the grouping of rules within a rule base is interesting and useful (Mehrotra&Johnson 1990). We propose, however, that rule base design and rule groupings should be established in the design phase of the expert system development to provide maximum benefit from the rule groupings. With this in mind we designed the AIRS rule base.

Rule base Structure
User Profiling Rules - parse files related to the user - adaptive keyword filter - develop user profile
Search Tactics Rules - formulate queries - evaluate queries - rank articles for user consideration
Command and Control Rules - distribute search engine workload over network - provide control elements to rules - interface with newsreader

Table 1. Rule base Structure

The AIRS rule base is organized into three general classes of rules. This organization was chosen not only to facilitate rule writing, but also to improve the extensibility of the rule base to other applications. For example, the first of these classes are User Profiling Rules, rules which develop a profile of the user. As discussed in the project overview, there are several files created by the newsreader which contain information about the user's news interests. The User Profiling Rules access these files, extract and filter the

information, then build the user profile. These rules, as initially implemented, appear to be very specific to the news domain. However, one would expect most information about a computer user to be stored in a file. Given this assumption, then any profiling of a user will be a process of accessing files, extracting and then filtering the information. It is expected then, with minimal modification, these rules could be extended to point at a different type of file, extract and filter that information and then build a profile of a user's interests in the new area.

Table 1 depicts the structure of the rulebase and shows some of the types of rules which may be found within a class. Rules in the search tactics class develop and evaluated the queries which are sent to the search engine for processing. The heuristics for developing and evaluating these queries have been previously validated and draw heavily upon work in library science (Bates 1979). A similar search tactics rulebase implemented in OPS5 has provided an excellent example, and many ideas for the search tactics rules (Gauch 1991). Rules within the search tactics class are intended to be immediately extensible to any type of query generation. The only modification required would be format changes due to using different search engines.

The third class of rules are the control and interface rules. These rules provide structure and execution order to the rulebase, and act as an interface to the operating system.

CLIPS Client/Server

Several of the modules of the newsreader system are anticipated to consume considerable system resources. The search engine will initially be very I/O bound while it constructs a reverse index, and then it is anticipated to swing over to being a heavily cpu bound process. AIRS places a heavy load on system memory resources as well as I/O due to the need to parse many files and then perform the appropriate pattern matching functions. To compensate for this, AIRS takes advantage of the Mach Operating system, and runs as a client/server based distributed system. Figure 3 depicts the current configuration of the *arn - adaptive news reader* with the AIRS component.

A client which serves as a triggering device is initiated by a rsh shell and an accompaning shell script. Once the client ensures that the parameters for low use time and minimal load have been met it sends the server which is blocked on receive. Our production model will migrate several of these functions to the chrontab but for prototype implementation it was simpler to use the rsh/client combination. When the clips server receives the message, it fetches the AIRS rule base, and begins profiling the user. Currently it is anticipated that AIRS will run on only one machine on the network. The AIRS component itself, when it has a query to send to the search engine, will send a remote procedure call to a machine on the net, to instantiate the search engine and run the query. As the search engine is the heaviest drain on system resources, the expert system will sequence through the various available resources, assigning a job to each, continuing to sequence through the resources until all of the jobs have been assigned. In the interest of efficiency, the platform running the AIRS component will not receive a search engine job. Developing a set of rules which would examine use and load of the the various platforms on the net was considered and rejected. By the time the evaluation was made and the job was assigned, the situation could have changed significantly, invalidating the initial findings.

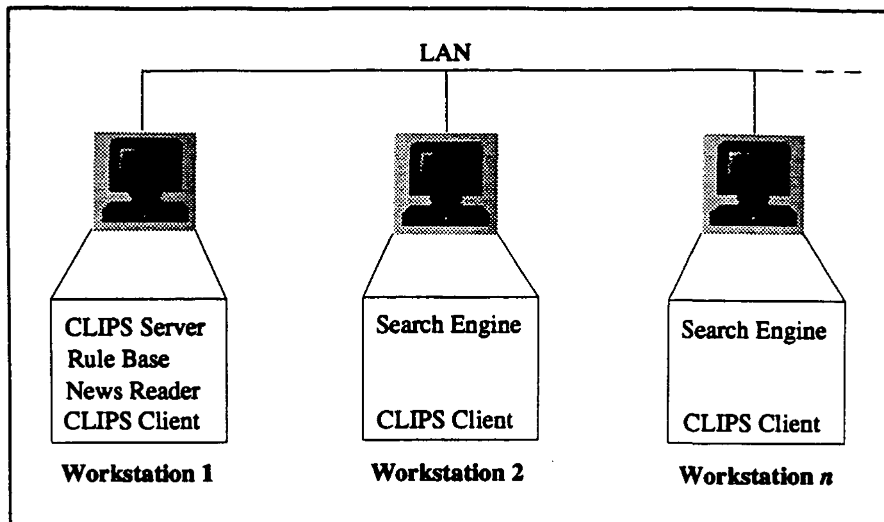


Figure 3. Newsreader configuration as a distributed network

AUTOMATED INFORMATION RETRIEVAL SYSTEM METHODOLOGY

Automated information retrieval methology in AIRS consists of developing a profile of the user, searching the newsbase for articles of interest, evaluating the articles returned, if necessary continuing the search, and finally ranking the articles and presenting them to the user via the newsreader. Figure 4 is a depiction of this process using a Petri Net. We find Petri Nets to be an excellent tool for modeling the behavior of an expert system and plan on validating and expanding upon previous work in this field (Etessami&Hura 1991). The first step in the AIRS process is user profiling.

User Profiling

User profiles are developed with keywords located in several files which are modified while a user reads the news. The newsreader creates two files which profile the newsreading habits of a user. The first of these is the .tindx file. The .tindx file is a linked list of news article "threads." A thread is an article with all of the responses generated by the article. Threads provide themes of interest within a newsgroup. The original *tass* newsreader has been modified to create a second file, the .keykill file. This file contains a reference to articles and threads which have been killed. This indicates a lack of interest in a news item or theme. A third file which indicates a user's interest is called the .keysrceng file and is created by the search engine. This file contains information pertaining to previous keyword searches over the newsbase which have been conducted by the user.

The profile of the user's interest is constructed using a weighted keyword scheme. First all three files are parsed and piped through a stop-list. The stop-list is an adaptive filter which eliminates keywords with a low discrimination ability. These words are kept in a .keyfilter file. Initially the .keyfilterfile is populated with a list of words which either are non-discriminate because of the English usage or are non-discriminate because of

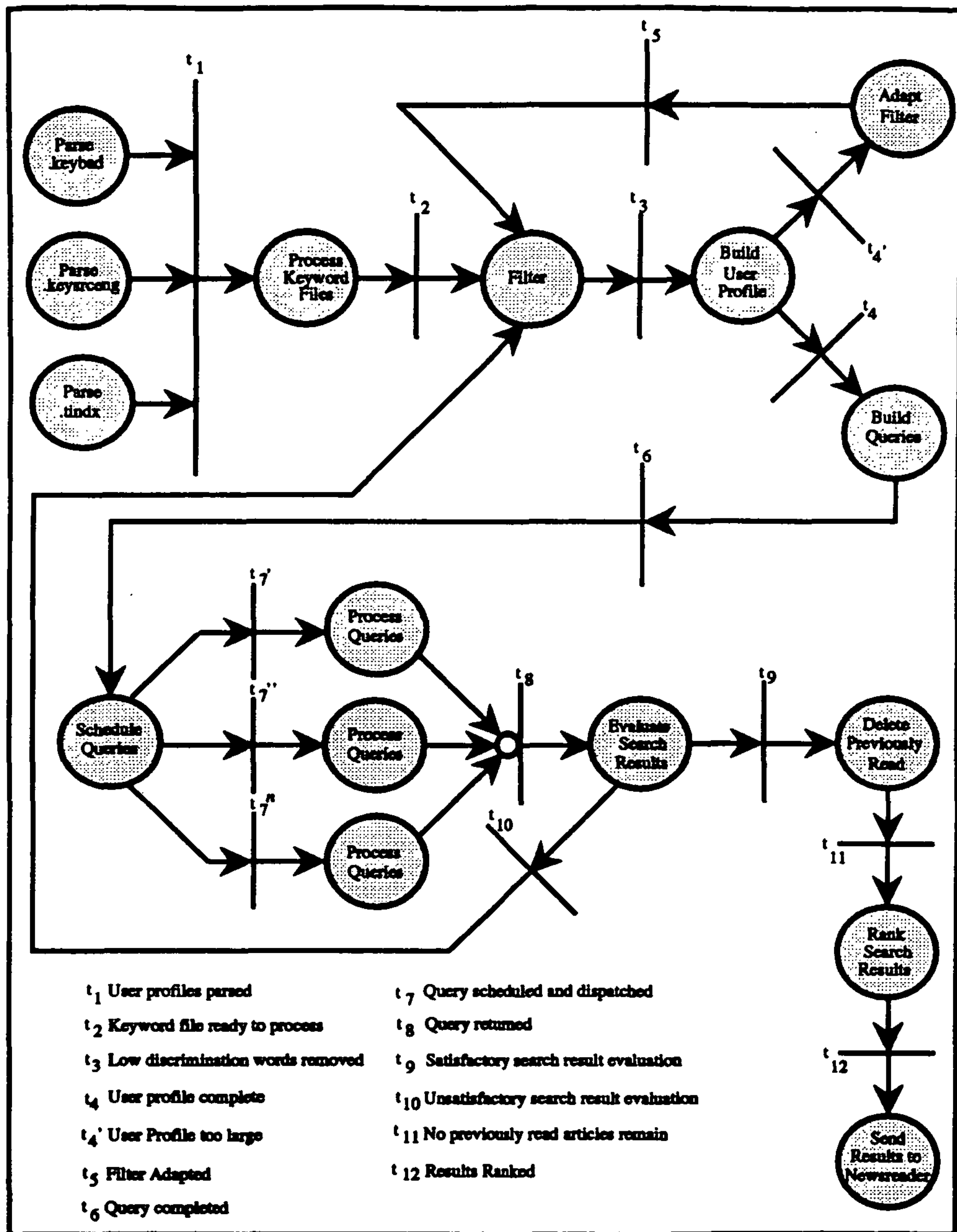


Figure 4. Petri Net representation of the AIRS prototype.

their relation to the news. Table 2 provides a sample of some of these words. It is interesting to note that many words fall into both catagories. The filter is adaptive in that as AIRS encounters words with low discrimination ability (which correspond to numerous instances of the word) the word is added to the .keyfilter file. The number of instances of a word which can exist without sending it to the .keyfilter file is the.keyfilter cutoff variable and may be adjusted if production is too high. Punctuation and contractions are also filtered out at this point.

Sample Non-discriminating Words									
English Usage									News Usage
if	while	the	Date	Subject	Organization	Message			
she	at	it	he	and	From	Re:	Lines	Keyword	Path

Table 2. Sample Non-discriminating Words

After an initial keyword list is developed, a weighting and elimination scheme is used to develop the profile of the user. In the initial prototype, a simple algorithm is used. Keywords from the .tindx file are given a weight of 1. Keywords from the .keykill file are given a weight of -2 and keywords from the .keysrceng file are given a weight of +2. The weights are given at each occurence in the file. The user profile is calculated by summing up the weights and ranking the keywords. A cutoff score is determined and keywords above the cutoff are sent off to the search engine.

Query generation and evaluation

The AIRS component prepares a query to send to the search engine after the user profile has been developed. This query is in the form of a single keyword search. Although the lqtext based search engine is capable of locating word combinations, this has been left to future iterations of the AIRS component. AIRS uses a round robin-scheduler to determine which computer on the network will receive the search for processing. The search query is bundled by the expert system into a remote procedure call (rpc) and sent to the appropriate network resource. When the query is received by a computer, the search is initiated and the results are returned to the platform running the expert system. The results are then collated, duplicate are articles deleted, and production is evaluated. If too few articles are returned, the total cutoff score will be lowered and new searches will be initiated. If too many articles are returned, the .keyfilter cutoff variable will be lowered, the.keyfilter file adjusted, and the articles returned will be re-evaluated. If an appropriate number of articles are returned, they will be examined to ensure that they have not already been read. Articles which the user has never seen will be ranked from most to least interesting and sent to the newsreader module for reading by the user.

RESULTS

Most of our initial objectives have been met in the prototype newsreader. Implementation time was exceptionally fast: a single-reader, single-machine configuration was prototyped in less than 1 programmer month. This is particularly significant given that the expert system programmer had no CLIPS or other rule-based programming

experience. As we expected, we have had no difficulty porting CLIPS to other computers. In fact, as is commonly done at Cal Poly, most of our rules were developed on IBM PC based machines, at home, and ran with no changes when transferred to the NeXT computers at school. Compiling a CLIPS client/server for the project was particularly easy, as was developing the interface to the Mach Operating System.

Use of *arn - adaptive read news* with the AIRS component is intended to supplement the normal reading habits of the user. Accordingly, an increase in news reader efficiency is predicted as readers will not only be able to continue their normal reading, but they will also benefit from the information provided by the expert system. Our newsreader/ expert system combination should provide a user with access to a significantly greater portion of the newsbase yet still reduce the amount of time a user spends reading the news. The second area of performance we consider is that of the traditional information retrieval measures of recall and precision. The actual performance of the expert system in terms of speed and efficiency are not considered in the evaluation of the prototype. As the prototype was designed to run in background during non-peak use periods, the performance of the expert system was not deemed critical to the prototype. In later production models, this will become an important issue.

Future Plans

Refinement of the Automated Information Retrieval System will continue using a stepwise development methodology. Current plans call for an upgrade in the search engine to full boolean search capability. Expansion and integration of more complex search tactics will continue, as will the effort to further distribute the system to improve efficiency. In its current form, the AIRS component relies solely on keyword-based heuristics. Additional user profiling heuristics will be added in later iterations. Distributed computing issues will be considered in hopes of improving performance. And as stated in our objectives, we intend to port the arn newsreader to other computer systems, including IBM PC's and Sun workstations.

The development of the Automated Information Retrieval System and *arn -adaptive read news* is the initial phase of an advanced information retrieval project, NewsClips, an intelligent system for the Automated Filtering of Time-Sensitive Information. NewsClips is envisioned as a distributed bulletin board system, running in a client/server configuration and written in CLIPS. The NewsClips system will evaluate incoming information in real time and, based on previously determined user profiles, will discard information of no value. It is intended as a testbed to determine whether sufficient confidence in an automated system can be achieved to allow it to discard information deemed of little value without human intervention.

Acknowledgements

This project is funded in part by the TRW Foundation. We would like to acknowledge the assistance of the Computer Systems Lab and the Advanced Workstations Lab at Cal Poly, and the following individuals for their assistance; Don Erickson, Ellen Clary, Peter Ogilvie, and Ding Yan.

References

(Bates 1979)

Bates, Marcia. "Information Search Tactics." *Journal of the American Society for Information Science* 30 (July 1979): 205-214.

(Etessami&Hura 1991)

Etessami, Farhad S. and Hura, Gurdeep S. "Knowledge Net Shell (KNS): Petri Net Based Development Tool for expert systems." *Microelectrtonics and Reliability* 31 4 (April 1991): 793-812.

(Gauch 1991)

Gauch, Susan Evalyn. "An expert system for Searching in Full-Text." Ph.D diss., University of North Carolina, 1991.

(Jacobs&Rau 1991)

Jacobs, Paul S. and Rau, Lisa F. "SCISOR: Extracting Information from On-line News." *Communications of the ACM* 33 (November 1990): 88 - 97.

(Mehrotra&Johnson 1990)

Mehrotra, Mala and Johnson, Sally C. "Rule Groupings in Expert Systems." *First CLIPS Conference Proceedings.* Houston: NASA, 1990, 274-285.

(Quam 1990)

Quam, Liam. "Lq-text"

(Reid 1991)

Reid, Brian. "Usenet Readership summary report for Jul 91." Unpublished electronic news article, August 1991.

(Robinson 1991)

Robinson, Mike. "Through a Lens Smartly." *BYTE* 16 (May 1991): 177-187.

(Skrenta 1990)

Skrenta, Rick. "Tass, rtass - Visual threaded Usenet news reader".

(Wyle&Frei 1991)

Wyle, M. F. and Frei, H. P. "Retrieving Highly Dynamic, Widely Distributed Information" *Proceedings of the Twelfth Annual International ACMSIGIR Conference on Research and Development in Information Retrieval.* Association for Computing Machinery: New York, 1989 pp 108-115.

N92-16600

Proposal for a CLIPS Software Library

Ken Porter

Harris Corporation
P. O. Box 98000
M/S R3-7257
Melbourne, FL 32902

Abstract. This paper is a proposal to create a software library for CLIPS, the C Language Integrated Production System expert system shell developed by NASA. Many innovative ideas for extending CLIPS were presented at the First CLIPS Users Conference, including useful user and database interfaces. CLIPS developers would benefit from a software library of re-usable code. The CLIPS Users Group should establish a software library--this paper proposes a course of action to make that happen. Open discussion to revise this library concept is essential, as only a group effort is likely to succeed. At the end of the paper is a response form intended to solicit opinions and support from the CLIPS community.

ANALYSIS OF THE CURRENT SITUATION

What kinds of software developed by one CLIPS programmer may be useful to others? How can they apply it to their own needs? Who would use a library, and who would support it? How hard is it to maintain and distribute software for a wide ranging user group, and what resources are available? Answers to these kinds of questions define the functional and operational requirements for a library facility.

Library Participants

NASA's open sharing of CLIPS with the public seems to have attracted interest primarily from university researchers, industry (IR&D) researchers, and small government-funded projects. There may also be a substantial number of PC and Macintosh users who have tinkered with CLIPS but are not really expert system developers. Together, these four overlapping groups are the "customers" or potential users of the software library; most of the good ideas and resources will come from them. There are several CLIPS-based products now available[1] or in prototype form, but commercially oriented developers are not likely to participate in a not-for-profit software library effort.

Resources

CLIPS users, NASA/JSC's Software Technology Branch (STB, the CLIPS development team), and expert systems research interests constitute the resources available to implement a software library.

The motivation for a CLIPS software library is the belief that a number of CLIPS users are willing to share their code, which would provide a great resource to other expert system application developers. The 82 papers presented at the First CLIPS Conference[2] provide evidence that such code is available. A survey of these papers shows about two-thirds have some re-use potential, either in the form of generic CLIPS extensions, improved user/developer environments for a particular hardware platform, or user-friendly "smart" front ends for specialized software packages. Several of these were in fact designed as generic interfaces, using formal or de-facto standards such as SQL, dBase III, and Hypercard. Most authors appeared willing to share their developments; at least seven authors made their code available at the Conference and several sent out softcopies shortly afterward.

If each author at the First CLIPS Conference represents ten others who did not contribute only for lack of time or money, then there are nearly a thousand potential contributors. This significant pool of talented people is the most important resource of the CLIPS software library. Creating a library could stimulate the CLIPS community to develop desirable re-usable functional extensions by encouraging them to consider generic requirements in designing and coding their applications.

Volunteers to maintain and distribute the software are a crucial resource. With sufficient motivation, the CLIPS Users Group should be able to staff a simple software library. Those most interested in obtaining library software might be harnessed to do most of the work. CLIPS users should be invited to participate, and the Library should exploit existing resources to the maximum practical extent.

Access to computers and data communications are key resources to implement a software library. Virtually all CLIPS users have access to computers of some kind, and usually there are few restrictions to using these computers after hours to support worthwhile activities. Access to data communications is less universal, but most users could at least find a PC with a modem so using electronic bulletin boards (BBSs) seems feasible. In addition to the existing JSC Software Support BBS, there may be other BBSs belonging to Group members.

NASA is interested in supporting CLIPS applications, but in these times of severe funding constraints it is difficult for public agencies to rationalize support for a small special interest group. Hopefully STB is already doing everything possible to further develop the CLIPS shell, in addition to providing COSMIC funding and sponsoring user conferences. COSMIC, in turn, has published the CLIPS newsletter for its first year to help foster an active users group. Although NASA and COSMIC participation is highly desirable, neither group should be expected to provide direct support beyond their current activities.

LIBRARY SPECIFICATION

Software Requirements

Ideally, the CLIPS software library would provide a programming resource packed with useful, portable C source code with concise but easily understood documentation. An excellent example is the blackboard extension BB_CLIPS developed by Orchard and Diaz[3], supported by the National Research Council of Canada. They responded to user requests at last year's Conference by distributing (at no charge) their C source for BB_CLIPS with detailed, well organized documentation and test files to support user compilation. Included with BB_CLIPS were files and documentation for FZ_CLIPS, a CLIPS extension for fuzzy logic reasoning. Even just an improved CLIPS executable with brief application notes holds potential for PC users; Marsh's graphical modeling shell ISA[4] provides an environment for graphically representing objects and linking to the CLIPS engine.

The CLIPS developer/user community is best served by establishing some requirements for submitting and reviewing candidate software. One possible set of software authorship goals is:

> Reliable operation in the intended operating environment
>
> Generic functionality - ease of use/adaptation
>
> Thorough documentation of user features
>
> Architecture discussion for other developers
>
> No license/restrictions imposed by author

Of course, submittals would not have to excel in every category to be useful to others and worth cataloguing. All submitted software should be reviewed, tested as appropriate, objectively assessed, and then posted for dissemination to the CLIPS community.

Software Assessment

Software assessment should include operational testing and evaluation of the software's potential for re-use or adaptation to new applications. Operational testing involves executing test cases, such as example knowledge bases or script files of new commands, to ensure the software performs as defined by the author. If the program code is supposed to be portable, test it on several common hardware platforms such as a PC, a UNIX workstation, and a VAX. If it provides a graphical user interface for a particular platform, test it on several different machines. For PCs, the normal variations in memory size, monitor, keyboard, and mouse interface can lead to unexpected problems. The assessment effort also should explore human factors, hardware and software limitations, and the range of potential applications or adaptation by other uses.

Feedback from the user community is valuable to both the library user and the author; the software review process should be responsive to the needs of both. This is vital to establishing a conduit of common interest that will support sustained library operation. The author should be provided two avenues of feedback: confidential technical feedback from the library's initial

technical review, and access to user comments after release. The reviewers should leave ample opportunity for the author to address their comments prior to software release, in respect to the author for his contribution. After release, the library should provide the users with a history of bug reports, work-arounds, and related developments. Authors should be provided names and addresses of people who have requested their software, unless the library takes responsibility for contacting people about support and revisions.

Distribution

Once CLIPS-related software has been received, reviewed, and posted for dissemination, there needs to be one or more mechanisms for distributing software. Requirements for distribution include:

Direct costs not exceed available funding

Controlled but easy access by the users

Support user feedback and author revisions

Containing costs is paramount. Many, if not most, CLIPS users seem to have considered low cost an important criteria in their initial selection of the CLIPS expert system shell. Later they probably found CLIPS's functionality, excellent documentation, portability, and extensibility to be important in building and eventually delivering their application. Still, the costs of a library facility should be in line with the "almost shareware" price of the CLIPS expert system, a big factor in its popularity. Minimizing user costs to make the library operation self-supporting greatly increases the feasibility of continued operations.

User should be able to access the library files easily, but this needs to be carefully controlled to prevent abuse and unauthorized use by non-members. Assigning a unique number or password for each user would provide a reasonable level of security, and is compatible with manual and electronic distribution methods. Some security provisions are also necessary to ensure the library facility can control and coordinate user comments, feedback to the authors, and dissemination of revisions to the right people. There would be little incentive for members to pay a usage fee for a software package if they knew subsequent revisions would be freely available.

User feedback is a most important benefits to the authors. Some mechanism should be established to collect comments which is simple and easy to use to encourage feedback. Distribution channels should be designed for two-way communication, whether on paper or in electronic form. If an author chooses to revise the software, the distribution system should support re-distribution with little additional effort on the part of the author.

PROPOSED CLIPS SOFTWARE LIBRARY

Strawman Approach

The proposed software library Strawman concept would create a self-supporting facility to collect, classify, and distribute CLIPS-related code for re-use by other expert system developers. However, the analysis and ideas presented in this paper are tentative at best. This "strawman" approach is intended to stimulate discussion and voluntary action within the CLIPS user community, especially members of the newly-formed CLIPS Users Group.

Organization

The CLIPS software library would be supported by four overlapping groups: a managing body, a team of technical reviewers, the general membership, and contributing authors. The managing body could be a standing CLIPS committee staffed by a few dedicated volunteers. Library Committee volunteers would do most of the work required to organize the overall effort; establish and maintain Library policies; solicit contributions and interface with authors; regulate the review process; and organize, store, and distribute the software.

Reporting to the Library Committee would be a pool of technical reviewers, experts with access to various computer systems. Hardware expertise should include Pcs, Macintoshes, workstations, VAXes, and possibly mainframes and supercomputers. Operating systems should include DOS (with and without Windows), Finder, UNIX, and VMS. Areas of functional expertise should include software design and testing, C language programming, human factors analysis, technical support and documentation, and of course expert system development. Initially, only some of the desired expertise will be volunteered; the Library Committee should strive to assemble a well-rounded pool of experts.

Since the Library Committee would be a part of the CLIPS Users Group, the baseline approach is to consider everyone in the CLIPS Users Group as a member of the Library. This provides a added benefit for being a member of the CLIPS Users Group, although it may discourage a few "free spirits" who don't join organizations as a matter of principle. Authorship should have no Library-imposed restrictions; there is already pressure from some employers to avoid complete disclosure of ideas. A positive incentive of one year free membership in the CLIPS Users Group could be offered to make authors' efforts worthwhile.

Operations and Coordination

To minimize costs for Library users, the costs of daily operations should exploit existing resources as much as possible, including telephones, existing networks, and dial-in PC/Mac-based BBSs.

Since the CLIPS Users Group is geographically scattered and financially limited, the Library Committee must employ electronic communications to organize and implement their objectives. Teleconferencing and facsimile transmission can provide the necessary interaction to coordinate

even complicated interactions. The Library Committee should use telephone conference calls, perhaps bi-monthly, to coordinate their efforts. A baseline approach is to use the NASA/JSC Software Support BBS as the backbone network for author uploads, reviewer interaction, and user downloads, under special access control per Library Committee policy. By assigning each CLIPS Users Group member a control number, their access privileges can be easily managed to meet Library guidelines. A NASA/STB representative should be actively involved in this process to ensure NASA consent in the use of this public-funded BBS.

Library listings should be available by mail or by modem from BBSs. A primary source would be the NASA/JSC board but other potential sources include boards/networks supported by CLIPS Users Group members. Access to Library software could be as simple as dialing the nearest BBS, providing a membership identification number, and downloading released source code and documentation. Privately supported BBSs should be persuaded to enforce Library policies.

Access to the software library is likely to increase the paid membership of the CLIPS Users Group, so perhaps some of the membership fees could be allocated to support Library Committee activities. However, the CLIPS Users Group is a fledgling organization that needs resources to grow; all direct costs should be factored into a usage fee charged for each Library software package.

Software Solicitation and Review

The Library Committee should exploit whatever opportunities are available to solicit material and membership. The CLIPS Users Group newsletter, newsCLIPS[5], is one avenue to reach potential authors. newsCLIPS may also be willing to publish information or short articles on new releases. Some professional trade magazines accept informational announcements and may prove otherwise helpful in reaching a wider audience (including those thousands of others who have "tinkered" with CLIPS).

When an author contacts the Library Committee, the Chair should assign one Committee member as the author's agent. The responsible agent would collect the material from the author and determine how it should be assessed. The Chair would assign one or more technical reviewers per the agent's recommendations. The review time should be as short as possible, and the agent would be responsible to coordinate among the technical reviewers and expedite the review process.

This software review process will benefit both the library user and the author. The rights of the authors are considered foremost--they should review all test findings, revise their submission if desired, and have the right to make a final decision on releasing the software. This means that during the review process the software must be handled in confidence. The author should be told who will test his software and the test team should be obligated to keep the software and its test results private. One means to implement this would be special BBS/network access privileges.

The agent should ensure that the author has reviewed the test results and any additional findings of the test and evaluation effort, then get the author's final consent on a standard form. The standard form should consent to a legal position on ownership and distribution rights, and

minimize liabilities. Finally, the agent should submit the material to the library for storage and distribution.

Ownership and Usage Rights

The degree to which the library owns the donated software should be thoroughly discussed with potential authors and the CLIPS community at large. One position would request the author to sign over his copyright to the Library. Some research supported by corporate or government sponsorship may stipulate certain kinds of restrictions on how the software is used (e.g. non-commercial use only). Library members could be asked to sign a limited distribution agreement to prevent secondary distribution. However, authors should act on the principal that the software is a donation to the CLIPS community, intended to empower others to freely exploit and further develop their code. The library effort would be improperly burdened to support any profit-oriented venture.

The Library Committee should establish the extent to which ownership and usage restrictions are acceptable. As one of its first duties, the Library Committee should draft a formal ownership document that spells out the rights of the author and library members; there may need to be several versions to meet the needs of individual, university, government, and corporate participants.

Direct Costs

If software media are provided by user and labor is provided by volunteers, the only remaining necessary distribution direct costs are reproduction and postage. For relatively small changes to the standard CLIPS syntax, user documentation could be furnished in ASCII soft copy (tape or floppy disk), which reduces reproduction costs to almost nothing. A distribution volume of perhaps two requests per user per year could require 20-100 copies per month, produced by perhaps 10-20 hours effort.

If the requestor supplies the required media (per the COSMIC model), distribution direct costs should be on the order of $1 per disk/Mbyte and 5-10 cents per page of paper documentation (if necessary for complex programs). These estimates include mailing a tape or several floppy disks (postage, mailer, and labels) and paper documentation (copier, additional postage, paper and envelopes). Distribution direct costs should be borne by the requestor.

Other Library Committee Tasks

The library should additionally be chartered to promote CLIPS and its applications in any aspect that seems feasible. As an adjunct to its primary tasks of review and distribution, the CLIPS software library should seek leverage from existing interest in CLIPS, expert systems, and general IR&D to promote the library concept and solicit new inputs. Exploiting opportunities to promote CLIPS will serve the common interest.

CONCLUSIONS

The analysis and discussions presented in this paper support the following conclusions:

A significant pool of re-usable software could be made available to CLIPS expert system developers at low cost. The software is out there, as demonstrated at the First CLIPS Conference. The new CLIPS Users Group might be willing to support a basic library facility.

Cooperative synergy between authors and users provide rewards to both. Feedback from the evaluation process and end users provides a valuable Beta-test to the author, and provides CLIPS developers/users with a source of documented and tested code.

Minimal additional support is required to implement a library facility. Modest usage fees, access to several existing BBS/networks, and a few active volunteers may be all that is needed to implement a software library.

RECOMMENDATIONS

The following recommendations are based on the presented analysis and conclusions:

The CLIPS Users Group should actively support a library facility. They should elect or appoint a Library Committee Chair and solicit volunteer from the membership. Issues such as library membership and software ownership should be resolved as soon as possible.

The library should minimize costs. The CLIPS user community is cost-sensitive. The library staff should minimize costs and leverage on existing interest in CLIPS, expert systems, and AI.

Those interested in participating in a software library should complete the form at the end of this paper. The survey results will be presented to the CLIPS Users Group if strong support is indicated.

REFERENCES

1. ECLIPSE, distributed by The Haley Enterprise, 413 Orchard St., Sewickley PA 15143. (412) 741-6420

2. Proceedings of the First CLIPS Users Conference, Vol. I and II, held at Johnson Space Center, Houston, Texas August 13-15, 1990. Published by COSMIC, Univ. of GA, 382 E. Broad St., Athens, GA 30602. NASA Conference Publication 10049.

3. Orchard. R. A., and A. C. Diaz, "BB_CLIPS: Blackboard Extensions to CLIPS", First CLIPS Conference Proceedings (Vol. II), pp. 581-591. Contact Orchard at Knowledge Systems Lab, Building M50, Montreal Rd., Ottawa, Ontario (Canada) K1A 0R6, orchard@ai.iit.nrc.ca

4. C.A. Marsh,"A PC Based Fault Diagnosis Expert System", First CLIPS Conference Proceedings (Vol. II), pp. 442-458.

5. newsCLIPS is the newsletter of the CLIPS Users Group, published by Dr. Linda K. Cook, Lockheed AI Center, PO Box 7732, Menlo Park CA 94026-7732; cook@netcom.com

CLIPS Software Library Response Form

This form is intended to solicit ideas, opinions, and voluntary support to create A CLIPS software library. Please indicate your degree of interest by answering the following questions.

Do you favor the creation of a CLIPS software library?
Yes No

Do you think some portion of the CLIPS Users Group membership fees should be allocated to support a software library, and if so, how much?
Yes, ____ % No

Do you feel library membership should include people who are not members of the CLIPS Users Group?
Yes No

Would you participate in a library as a member, with privileges including low-cost access to software?
Definitely would Might Would not

Would you participate as a contributing author, including donating your software and documentation?
Definitely would Might Would not

Would you volunteer some time as either a Library Committee member or technical reviewer?
Definitely would Might Would not

If interested in performing technical reviews, what operating systems and hardware platforms do you understand well and have access to?

____

Do you own/operate a network or electronic bulletin board service which may be used to support library efforts? Yes No

Other comments:

Name, address, and phone number (voluntary):

SESSION 6 B

PRECEDING PAGE BLANK NOT FILMED

N92-16601

IMPROVING NAVFAC's TOTAL QUALITY MANAGEMENT OF CONSTRUCTION DRAWINGS WITH CLIPS

by

Albert Antelman AIA
Shore Facilities Department
Naval Civil Engineering Laboratory
Port Hueneme, CA 93043

ABSTRACT This paper describes a diagnostic expert system to improve the quality of Naval Facilities Engineering Command (NAVFAC) construction drawings and specification. CLIPS and CAD layering standards are used in an expert system to check and coordinate construction drawings and specifications to eliminate errors and omissions.

INTRODUCTION

Designing and constructing naval shore facilities for the United States Navy is a complex process. The quality of construction documents is a major factor in this process. The review and coordination of construction drawings and specifications is one of the critical tasks performed by NAVFAC architects and engineers. Defective drawings and specifications can lead to change orders, time delays, and litigation.

Experience has shown that more than half of the errors and omissions found in construction drawings and specifications result from inadequate coordination between architectural and engineering disciplines (Nigro, 1984). A recent study by the U.S. Army Corps of Engineers found that more than 95 percent of all review comments addressed coordination issues (Kirby, 1989).

In response to the problem, NAVFAC implemented a quality assurance program in April of 1986. An interdisciplinary coordination review checklist was developed to check for in-consistencies, interferences, errors and omissions, both technically and graphically, that may exist in or between disciplines. A recent survey by Charles Markert, NAVFAC's Deputy Assistant Commander for Engineering and Design found that NAVFAC has discovered significant benefits from conducting interdisciplinary coordination checks at the final design stage of projects (DCQI, 1990).

The NAVFAC interdisciplinary coordination checklist contains over 500 review items. The checklist, when used conscientiously, can eliminate many of the design deficiencies which have occurred in past construction projects. Current procedures calls for each checklist item to be analyzed for applicability to the project's drawing and specification content. This is accomplished by manually reviewing the drawings and specifications with the checklist. If an item is found not applicable, the letters "NA" will be inserted adjacent to the checklist item. The remaining checklist items are used to perform the interdisciplinary coordination review.

PRECEDING PAGE BLANK NOT FILMED

THE PROBLEM

The development and application of quality control coordination checklists is a step in the right direction, but does not provide a production oriented solution to the problem. Often checklists contain several hundred items which may not be applicable to the drawing and specification content. Typically, due to quantity and nonapplicability, checklist items are often ignored during the review process. The process of editing, comparing, and coordinating checklist items with the drawings and specifications is time consuming considering it is not unusual for project drawings to exceed 50 sheets. Checklist editing also assumes a level of experience the reviewer may not possess and may well result in the non-prioritizing of the issues being checked.

The majority of NAVFAC's construction drawings are produced using manual drawing procedures, but this is rapidly changing. NAVFAC as well as architectural/engineering firms under contract to NAVFAC have made heavy investments in computer-aided design (CAD) hardware and software. Receiving construction drawings delivered in a CAD format is becoming common. Despite the self-coordinating aspects of CAD drawings, coordination and omission errors can still arise. No matter what process (manual drafting, systems drafting or CAD) is used to produce a set of construction drawings, all drawings need to be checked (Duggar, 1984).

OBJECTIVE

The objective of this project is to produce an easy-to-use, automated, expert system, capable of quickly analyzing project data (drawing and specification content), recognize potential coordination issues, establish review priorities and provide quality control guidance specific to the project being reviewed. The expert system must function as an intelligent assistant which provides the user with knowledge (advice) based on expert experience and lessons learned from past projects with similar drawing content.

SOLUTION

The solution to the problem of automating the quality review of construction drawings and specifications is to develop a rule-based diagnostic expert system capable of reading the drawing contents of CAD drawing database files. The C Language Integrated Production System (CLIPS) was selected as the expert system shell and AutoCad software running in conjunction with the CadPLUS Total Architectural/Engineering software was selected to produce the CAD drawings.

The CAD Data Base

The CadPLUS Total Architectural/Engineering System is a powerful facility design tool developed by the Naval Civil Engineering Laboratory and CadPLUS Products Company of Albuquerque, NM under a Cooperative Research and Development Agreement. The

software runs in conjunction with AutoCad and implements the CAD Layering Guidelines published by the American Institute of Architects (AIA).

In order to insure reusability of CAD drawings during a facility's life cycle, NAVFAC has adopted a standard approach for the use of CAD layers. Layering is "the basic method most CAD systems use to group information for display, editing, and plotting purposes" (Schley, 1990a). NAVFAC along with the American Institute of Architects, the American Consulting Engineers Council, the American Society of Civil Engineers, International Facility Management Association, United States Army Corp of Engineers and the Department of Veterans Affairs sponsored the development of a standard approach for the use and naming of CAD layers.

It was not the intention of the CAD Layering Guidelines to attempt to use layers to carry "drawing intelligence" (Schley, 1990a), however the CAD Layering Guideline's structure and format, see Figures 1 through 5, provide a detailed description of a project's drawing content. Drawing content is the key to determining the applicability of interdisciplinary coordination checklist items.

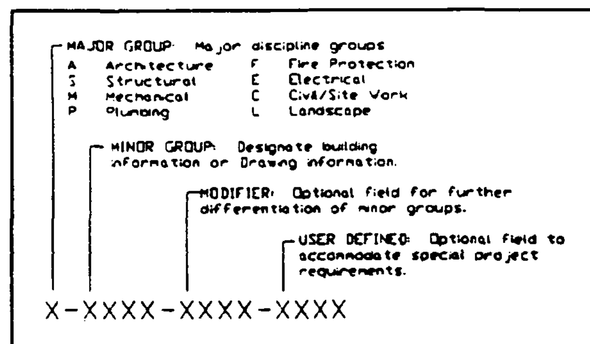


Figure 1. AIA layer name format (Schley, 1990b)

ARCHITECTURE	
A-WALL	WALL
A-DOOR	DOORS
A-GLAZ	WINDOWS
A-FLOR	FLOOR INFO
A-EQPM	EQUIPMENT
A-FURN	FURNITURE
A-CLNG	CEILING INFO
A-ROOF	ROOF
A-SHBD	SHT/BORDER
A-PFLR	FLOOR PLAN
A-PLGS	LG SCALE PLAN
A-PCLG	REFLECTED CEILING PLAN
A-PROF	ROOF PLAN
A-PEQM	EQUIP PLAN
A-ELEV	ELEVATIONS
A-SECT	SECTIONS
A-DETL	DETAILS
A-SCHD	SCHEDULES

PLUMBING	
P-ACID	ACID, ALKALINE AND OIL
P-DOMW	DOMESTIC HOT & COLD WATER SYS
P-SANR	SANITARY DRAINAGE
P-STRM	STORM DRAINAGE
P-EQPM	PLUMBING MISC EQUIPMENT
P-FIXT	PLUMBING FIXTURES
P-SHBD	SHT BORDER/TITLE
P-PPLM	PLUMBING PLAN
P-PDRA	STORM DRAINAGE
P-PSAN	SANITARY DRAINAGE
P-RISR	PLUMBING RISER DIAGRAM
P-ELEV	ELEVATIONS
P-SECT	SECTIONS
P-DETL	DETAILS
P-SCHD	SCHEDULES

STRUCTURAL	
S-GRID	COLUMN GRID
S-FNDN	FOUNDATION
S-SLAB	SLAB
S-ABLT	ANCHOR BOLT
S-COLS	COLUMN
S-WALL	BEARING/SHEAR WALL
S-FRAM	BEAMS/JOISTS
S-SHBD	SHT BORDER/TITLE
S-PFND	FOUNDATION PLAN
S-PSFR	FRAMING PLAN
S-PCOL	COLUMN PLAN
S-ELEV	ELEVATIONS
S-SECT	SECTIONS
S-DETL	DETAILS
S-SCHD	SCHEDULES

ELECTRICAL	
E-LITE	LIGHTING
E-POWR	POWER
E-CTRL	ELEC CONTOL SYSTEM
E-GRND	GROUND SYSTEM
E-AUXL	AUXILIARY SYS
E-LTNG	LIGHTNING PROTECTION SYS
E-FIRE	FIRE ALARM SYS
E-DATA	DATA SYSTEMS
E-SOUN	SOUND/PA SYSTEMS
E-TVAN	TV ANTENNA SYS
E-CCTV	CLOSED CRT TV
E-NURS	NURSE CALL SYS
E-SERT	SECURITY SYSTEM
E-SHBD	SHT BORDER/TITLE
E-PLIT	LIGHTING PLAN
E-PPOW	POWER PLAN
E-PCOM	COMMUNICATION SYSTEM PLAN
E-PROF	ELEC ROOF PLAN
E-LEGN	LEGEND
E-ILIN	ONE LINE DIAGRAM
E-RISR	RISER DIAGRAM
E-ELEV	ELEVATIONS
E-SECT	SECTIONS
E-DETL	DETAILS
E-SCHD	SCHEDULES

MECHANICAL	
M-BRIN	BRINE SYSTEM
M-CHIM	PREFAB CHIMNEY
M-CMPA	COMPRESSED AIR
M-CONT	CNTRL & INSTR
M-DUST	DUST & FUME COLLECTION
M-ENER	ENERGY MNGT
M-EXHS	EXHAUST SYSTEM
M-FUEL	FUEL SYS PIPING
M-HVAC	HVAC SYSTEM
M-HOTW	HOT WATER HEATING SYSTEM
M-CWTR	CHILLED WATER SYSTEM
M-MDGS	MEDICAL GAS SYS
M-PROC	PROCESS SYSTEM
M-REFG	REFRIG SYSTEM
M-SPCL	SPECIAL EQUIP
M-STEM	STEAM SYSTEM
M-SHBD	SHT BORDER/TITLE
M-PPIP	PIPING PLAN
M-PDUC	DUCT PLAN
M-PEXD	EXHST DUCT PLAN
M-PHVA	HVAC PLAN
M-PSTM	STEAM PIPING PLAN
M-PWCH	CHILLED WATER PIPING PLAN
M-ELEV	ELEVATIONS
M-SECT	SECTIONS
M-DETL	DETAILS
M-SCHD	SCHEDULES

LANDSCAPE	
L-PLNT	PLANTING & LANDSCAPE MATERIALS
L-IRRG	IRRIGATION SYSTEM
L-WALK	WALKS & STEPS
L-SITE	SITE IMPROVEMENTS
L-SHBD	SHT BORDER/TITLE
L-PSIT	SITE PLAN
L-PPLA	PLANTING PLAN
L-PIRR	IRRIGATION DRAWING
L-PWLK	WALKS/PAVING PLAN
L-ELEV	ELEVATIONS
L-SECT	SECTIONS
L-DETL	DETAILS
L-SCHD	SCHEDULES

CIVIL/SITE	
C-PROP	PROPERTY LINE & SURVEY BENCHMARKS
C-TOPO	PROPOSED CONTOUR LINES & ELEV
C-BLDG	PROPOSED BLDG FOOTPRINT
C-PKNG	PARKING LOTS
C-ROAD	ROADS
C-STRM	STORM DRAINAGE
C-COMM	SITE COMMUNICATIONS
C-WATR	DOMESTIC WATER
C-FIRE	HYDRANTS
C-NGAS	NAT GAS MANHOLES, METERS & TANKS
C-SSWR	SANITARY SEWER
C-SHBD	SHT BORDER/TITLE
C-PSIT	SITE PLAN
C-PELC	SITE ELEC SYS PLAN
C-PUTL	SITE UTILITY PLAN
C-PGRD	GRADING PLAN
C-PPAV	PAVING PLAN
C-ELEV	ELEVATIONS
C-SECT	SECTIONS
C-DETL	DETAILS
C-SCHD	SCHEDULES

FIRE PROTECTION	
F-CO2S	CO2 SYSTEM
F-HALN	HALON
F-SPRN	FIRE SPRINKLER SYSTEM
F-PROT	FIRE PROTECTION SYSTEM
F-SHBD	SHT BORDER/TITLE
F-PSPR	SPRINKLER PLAN
F-RISR	SPRINKLER RISER DIAGRAM
F-PFPE	FIRE PROTECTION EQUIPMENT PLAN
F-ELEV	ELEVATIONS
F-SECT	SECTIONS
F-DETL	DETAILS
F-SCHD	SCHEDULES

Figure 2. Typical building and drawing layers without modifiers (Schley, 1990b).

ORIGINAL PAGE IS
OF POOR QUALITY

The layer modifiers listed below may be used with any building iformation layers.

*-****-IDEN	IDENTIFICATION TAG
*-****-PATT	CROSS-HATCHING AND POCHE
*-****-ELEV	VERTICAL SURFACES (3D DRAWINGS)
*-****-EXST	EXISTING TO REMAIN
*-****-DEMO	EXISTING TO BE DEMOLISHED OR REMOVED
*-****-NEWW	NEW OR PROPOSED WORK (REMODELING PROJECTS)

EXAMPLE:
A-WALL-EXST USED TO DESIGNATE WALLS TO REMAIN

Figure 3. Building information layer (Schley, 1990c).

The layer modifiers listed below may be used with any drawing iformation layers.

*-****-NOTE	NOTES, CALL-OUTS AND KEY NOTES
*-****-TEXT	GENERAL NOTES AND SPECIFICATIONS
*-****-SYMB	SYMBOLS, BUBBLES, AND TARGETS
*-****-DIMS	DIMENSIONS
*-****-PATT	CROSS-HATCHING AND POCHE
*-****-TTLB	TITLE BLOCK, SHEET NAME AND NUMBER
*-****-NPLT	NONPLOT INFORMATION AND CONSTRUCTION LINES
*-****-PLOT	PLOTTING TARGETS AND WINDOWS

Figure 4. Drawing information layer modifiers (Schley, 1990c).

Modifiers may be added to layer names for further differentiation. For example, ceiling information (A-CLNG) may be categorized as:

A-CLNG-GRID	CEILING GRID
A-CLNG-OPEN	CEILING AND ROOF PENETRATIONS
A-CLNG-TEES	MAIN TEES
A-CLNG-SUSP	SUSPENDED ELEMENTS
A-CLNG-PATT	CEILING PATTERNS

Figure 5. Typical ceiling modifiers (Schley, 1990c).

ORIGINAL PAGE I
OF POOR QUALIT

CLIPS Expert System Shell

CLIPS is a forward chaining rule-based expert system shell, "designed at NASA/Johnson Space Center with the specific purpose of providing high portability, low cost and easy integration with external systems" (Giarratano 1989a). The three major components of the CLIPS expert system shown in Figure 6 are:

1. Fact-list: global memory for data
2. Knowledge-base: contains all the rules
3. Inference engine: controls overall execution

"In order to solve a problem, the CLIPS program must have data or information with which it can reason. A chunk of information in CLIPS is called a fact" (Giarratano 1989b). The programs fact-list is a product of the CAD drawing database. A LISP program within the CAD system is used to generate an ASCII file (layer.dat) listing all layers present within the CAD graphic database. The CLIPS load-facts function is used to input the facts into the program. The following are examples of facts :

Fact List	Description
(a-wall-new)	Architectural wall, new
(a-prof)	Roof Plan
(s-psfr)	Structural Framing Plan
(p-strm-rfdr)	Roof Drain
(e-prof)	Electrical Roof Plan

A rule is the method that CLIPS uses to represent knowledge. An example of a possible rule for checking drawing coordination is:

> **IF** the project drawings contain a Roof Plan and Roof Framing Plan.
> **THEN** coordinate the Roof Plan with the Roof Framing Plan and verify direction of roof slope.

The rule expressed in CLIPS format would appear as:

```
(defrule coordinate-roof-plan-and-roof-framing-plan
   (a-prof)
   (s-psfr)
   =>
   (fprintout t "Coordinate the Roof Plan with the Roof Framing
   Plan" crlf)
   (fprintout t "Verify direction of the roof slope."
   crlf))
```

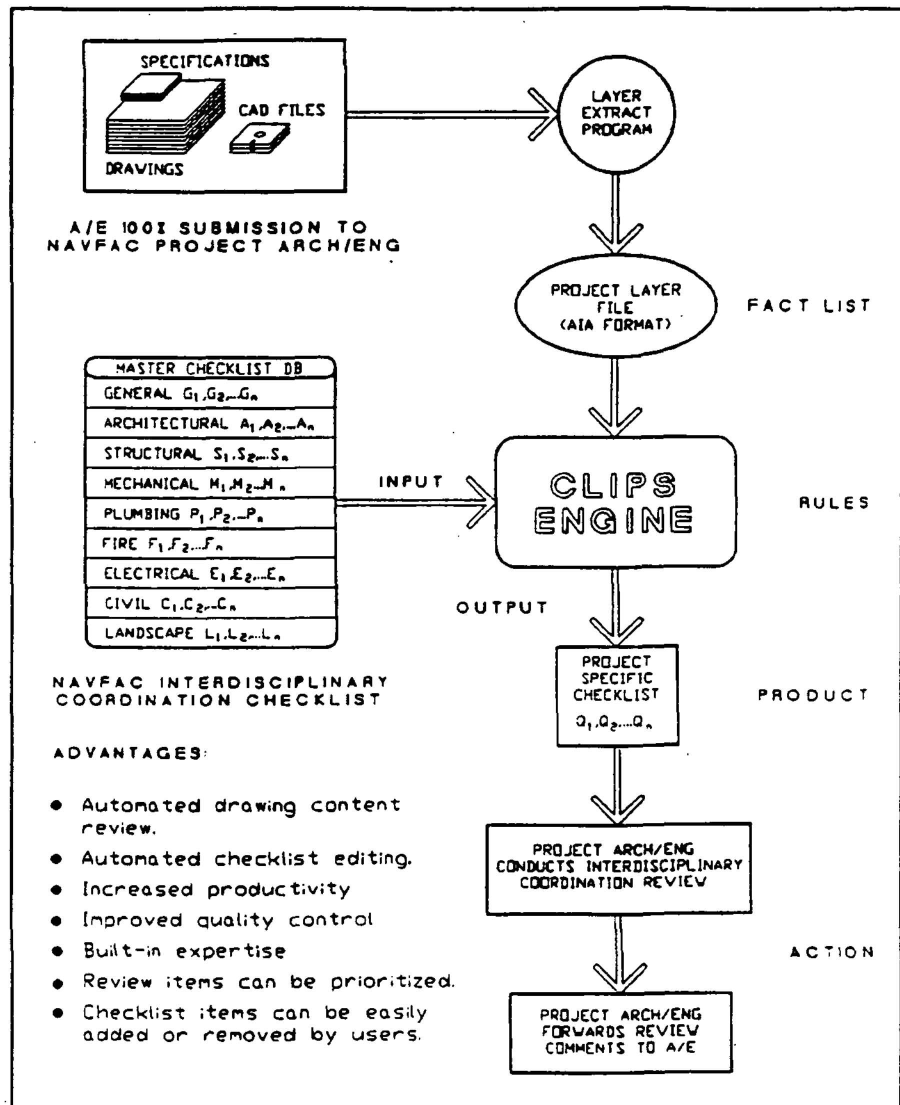


Figure 6. Prototype work model.

The knowledge-base rules are a product of the existing NAVFAC interdisciplinary coordination checklist and REDICHECK. REDICHECK, which was developed by LCDR William T. Nigro, CEC, USN (Ret) is a structured coordination review system that is also implemented by using a manual checklist.

The CLIPS inference engine makes inferences by deciding if a rule is satisfied by the facts. For example, if a project under review contained a Roof Plan (**layer a-prof**) and a Framing Plan (**layer s-psfr**), then pattern matching would occur in the previously defined defrule and the knowledge-base would consider the review comment as applicable. In this application, the CLIPS knowledge-base consists of rules that when activated by matching facts, outputs a project-specific quality control coordination checklist.

The rules required to generate a project-specific checklist are embedded in the CLIPS program. In order to reduce the size of the program, the master checklist items are stored outside the program and accessed by the CLIPS read function.

ORIGINAL PAG
OF POOR QUAL

CLIPS also has a feature to control the execution of rules called salience. Salience values are used to order the rules in terms of increasing priority and will activate rules to assemble a prioritized project specific checklist.

FUTURE WORK

To date, much progress has been made in understanding the problem domain and developing the knowledge-base. Future development plans include:

1. Development of a menu driven interface.

2. Development of rules that identify omissions, duplications, and inconsistencies between reference/identification symbols (detail bubbles, door reference symbols, equipment numbers, etc.) and details, sections, and schedules.

3. Development of rules that identify omissions, duplication, and inconsistencies between labels/keynotes and project specifications.

4. Development of an interface between the CAD geometric data base and the CLIPS knowledge-base.

CONCLUSIONS

At a recent Naval Sea Systems Command conference, Admiral Frank B. Kelso, II, Chief of Naval Operations commented that we have "to learn to do things more efficiently; with better quality than we had in the past." In this application, CLIPS provides NAVFAC with a powerful tool to improve the total quality management of the construction document review process.

ACKNOWLEDGMENTS

The author acknowledges the advice and assistance of Dr. Jens Pohl, CAD Research Unit, School of Architecture and Environmental Design, California Polytechnic State University, San Luis Obispo, California.

REFERENCES

DCQI, (1990). Design & Construction Quality Institute Newsletter, Spring 1990, Washington D.C., pg 1.

Duggar III, J.F. (1984). Checking and Coordinating Architectural and Engineering Working Drawings, McGraw-Hill Book Company, New York, pg. 4.

Giarratano, J. (1989a). Expert Systems, Principles and Programming, PWS-KENT Publishing Company, Boston, pg. 373-374.

Giarratano, J. (1989b). Expert Systems, Principles and Programming, PWS-KENT Publishing Company, Boston, pg. 379.

Kirby, J. G. (1989). Constructibility and Design Reviews: Analysis and Recommendations for Improvement, USACERL Technical Report P-89/15, US Army Corps of Engineers, Construction Engineering Research Laboratory, pg. 25.

Nigro, W.T. LCDR (1984). REDICHECK: A System of Interdisciplinary Coordination, DPIC Communique, Vol. 11, No. 6, Montery, pg. 3.

Schley, M. K. (ed) (1990a). CAD Layer Guidelines: Recommended Designations for Architecture, Engineering, and Facility Management Computer-Aided Design, The American Institute of Architects, Washington D.C., pg. 11.

Schley, M. K. (ed) (1990b). CAD Layer Guidelines: Recommended Designations for Architecture, Engineering, and Facility Management Computer-Aided Design, The American Institute of Architects, Washington D.C., pg. 13 and 16-20.

Schley, M. K. (ed) (1990c). CAD Layer Guidelines: Recommended Designations for Architecture, Engineering, and Facility Management Computer-Aided Design, The American Institute of Architects, Washington D.C., pg. 21 and 23.

N92-16602

VALIDATION OF AN EXPERT SYSTEM INTENDED FOR RESEARCH IN DISTRIBUTED ARTIFICIAL INTELLIGENCE

C. Grossner, J. Lyons, and T. Radhakrishnan

Concordia University
Deparment of Computer Science
Montreal, Quebec
Canada H3G 1M8

Abstract. The expert system discussed in this paper is designed to function as a testbed for research on cooperating expert systems. Cooperating expert systems are members of an organization which dictates the manner in which the expert systems will interact when solving a problem. The Blackbox Expert described in this paper has been constructed using CLIPS, C^{++}, and X windowing environment. Clips is embedded in a C^{++} program which provides objects that are used to maintain the state of the Blackbox puzzle. These objects are accessed by CLIPS rules through user-defined function calls. The performance of the Blackbox Expert is validated by experimentation. A group of people are asked to solve a set of test cases for the Blackbox puzzle. A metric has been devised which evaluates the "correctness" of a solution proposed for a test case of Blackbox. Using this metric and the solutions proposed by the humans, each person receives a rating for their ability to solve the Blackbox puzzle. The Blackbox Expert solves the same set of test cases and is assigned a rating for its ability. Then the rating obtained by the Blackbox Expert is compared with the ratings of the people, thus establishing the skill level of our expert system.

INTRODUCTION

Distributed Artificial Intelligence or DAI is the branch of AI that is concerned with the problems of coordinating the actions of multiple intelligent agents, in order to solve a large and complex problem (Gasser 1987). The agents could be expert systems or other types of AI programs. Two factors that could lead to the distribution of such agents are a geographic distribution required due to the intrinsic properties of the problem being solved and a functional distribution of the problem. The Distributed Vehicle Monitoring Testbed or DVMT distributes its agents based on a geographic distribution of sensors (Durfee 1987) whereas the Distributed Blackbox Testbed described in (Pitula 1980) is based on data partitioning. In this paper, we are concerned with the validation of an expert system (called Blackbox Expert) that solves the Blackbox puzzle. This expert system will be used as an agent in our DAI research. Our laboratory contains a Distributed Computing Facility that is composed of the MACH Distributed Operating System Kernel, a network of SUN workstations, C++, and CLIPS.

The resource constraints of a university environment will permit the development of low cost prototypes only. This normally has several conflicting requirements:

(a) The test environment should be "rich" in problems but not too complex to solve in a realistic time with the available resources.
(b) The test data required by the experiment should be easy to obtain but not trivial.
(c) Exercising and evaluating the prototype should reveal "more than obvious" behaviour of the modeled system.

The prototype systems used thus far in DAI research have been very complex and have required several man years of effort. For example, the manpower expended in the DVMT project is estimated to be 15 to 20 man years. In this context, the advantages and disadvantages of using Blackbox as an experimental test case are described in (Pitula 1990).

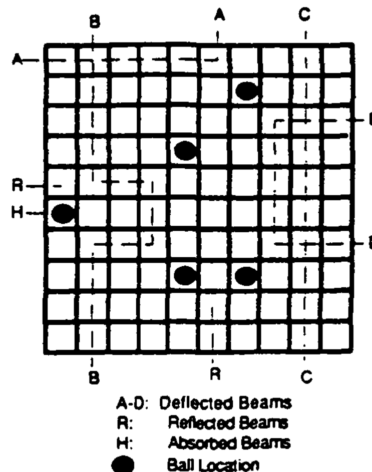


Figure 1: Beam Behaviour in Blackbox

Validation of a system substantiates that it performs with an acceptable level of accuracy (O'Keefe 1987). Developing a validated expert system for DAI research is a non-trivial task and must face several issues:

(a) Acquiring the knowledge and building the knowledge base.
(b) Arriving at an appropriate metric for performance evaluation.
(c) Finding an adequate number of human experts to participate in the validation experiment.
(d) Designing an experiment for statistical evaluation of the performance of the expert system compared with that of humans.

Unlike the case of other well known games and puzzles such as chess, there are no known ways of rating human problem-solvers of Blackbox. On the contrary, Blackbox is simple to learn, the correct solutions are known, and knowledge acquisition is not too expensive. One of the outcomes of our validation experiment is that we now have a set of test cases that are placed into multiple groups of increasing complexity. The performance of the expert system is compared with that of humans using this test set.

BLACKBOX AND DAI

The Blackbox puzzle[1] consists of an opaque square grid (box) with a number of balls hidden in the grid squares. The puzzle solver can fire beams into the box. These beams interact with the balls, allowing the puzzle solver to determine the contents of the box based on the entry and exit points of the beams. As illustrated in Figure 1, the beams may be fired from any of the four sides of the box (along one of the grid rows or columns) and follow four simple rules:

(a) If a beam hits a ball, it is absorbed. (Labelled by 'H')
(b) If a beam tries to pass next to a ball, it is reflected 90 degrees away from the ball in the square diagonally next to the ball. (Labelled Alphabetically except for 'H' and 'R')
(c) If a beam tries to enter the grid at a square adjacent to a border square that contains a ball, it is reflected back out the way it came in. (Labelled by 'R')
(d) If a beam tries to pass between two balls, it is reflected back 180 degrees. (Labelled by 'R')

[1] In previous publications Blackbox was referred to as a game. We have now decided to refer to it as a puzzle, because the word "game" implies an element of luck. The word "puzzle" implies that a skill is needed to find the solution to a problem, which is definitely the case with Blackbox.

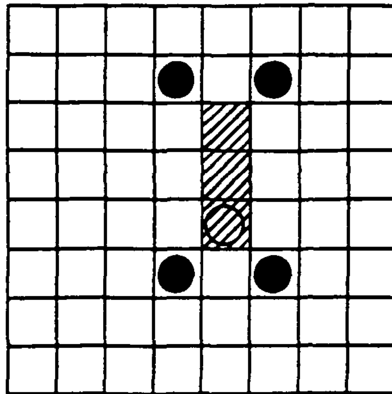


Figure 2: An Example of a Shielded Region.

The objective of the Blackbox puzzle-solver is to determine the contents of as many of the grid squares as possible, while minimizing the total *value* of beams fired. Beams that are absorbed or reflected have a value of one point, while deflections have a value of two points. The puzzle is solved iteratively by firing beams and observing their exit point from the grid. The information obtained from observing the exit points of the beams, and the problem solver's knowledge of how the beam can be affected by the balls within the box are used to create hypotheses about the contents of the box. The amount of information to be processed when solving the puzzle increases with the grid size and the total number of balls.(Pitula 1990).

A region is called a *shielded region* if it is a proper subset of the Blackbox square, it contains at least one ball, and is shielded by other balls so that no beams can penetrate into the region. An example of a shielded region is shown in Figure 2, wherein the shielded region is shaded. No beam can penetrate into the shaded region. The balls contained in this region are called "unmappable balls". In the case of a shielded region, a person can only state that the contents of each square in the region remains unknown.

According to (Parsaye and Chignell 1988), expert system tasks may be categorized into four well defined classes that require different approaches and methodologies: diagnosis and repair, monitoring and control, design and configuration, and intelligent tutoring. The tasks performed by the Blackbox Expert fall into the first category. Diagnosis may be viewed as the task of discovering the relationships between symptoms and faults (diseases). A single symptom, at a given level of granularity, may be the result of many different interacting faults (diseases). On the other hand, a single fault (disease) may produce many symptoms. These one-to-many non-independent relationships make the task of diagnosis quite difficult. Often the diagnosis expert is required to suggest a method of remedy or repair. As an analogy, consider the following pairs of subtasks performed by an expert system for medical diagnosis and the Blackbox Expert respectively:

(a) Observing the symptoms of a disease or the results of a medical test versus observing the exit points of the beams fired.
(b) Suggesting new medical tests versus selecting one or more beams to fire before hypothesising where the balls are located.
(c) Ruling out the possibility of certain diseases versus marking the squares as empty.
(d) Deciding how serious the potential errors in diagnosis are versus setting the weights for the terms in the SCORE function (described in the section on Validation of the Blackbox Expert).
(e) Taking into account the past treatment procedures followed for a disease versus considering the hypotheses generated from the sequence of beams that have been fired.

These types of analogies help in transferring the methodologies developed and the results of research from one problem domain to another.

Several areas that are of interest to the DAI community include negotiation protocols for expert systems (Smith 1980), blackboard architectures (Jagannathan et al. 1989), the sharing of

information among multiple expert systems (Durfee 1987), and coordination of multiple expert systems (Ginsberg 1987). When multiple experts are used to solve a single problem, they can be members of an *organization* (Grossner 1990). An organization defines the type of control flow and data flow permissible among the experts. When dealing with Ill-Structured problems (Simon 1973) there are two phases required for problem-solving, namely the planning phase and the execution phase (Durfee 1986). The control and data flow constraints may be applied in each of these phases as necessary. In (Grossner 1990), it is shown through three different example organizations that Blackbox is an interesting problem for DAI research.

THE BLACKBOX EXPERT

The primary goal in the design of the Blackbox Expert (Lyons 1990a, Lyons 1990b) was to provide an expert system that could later be used in our experiments with *organizations* of cooperating expert systems. Any system to be used for such a purpose requires the following features:

(a) It must be easy to modify the knowledge base.
(b) It must be possible to modify the data structures used by the Blackbox Expert without affecting the knowledge base.
(c) As the data structures used by the Blackbox Expert will be moved from its working memory to a blackboard when the Blackbox Expert becomes a member of an organization, they should be designed to minimize the effects of this move.
(d) It must be possible to monitor which rules are fired as the Blackbox puzzle is solved.
(e) It must be possible to monitor the number and type of accesses to the data structures used by the Blackbox Expert as it solves test cases of the Blackbox puzzle.

Easy modification of the knowledge base provides the flexibility we require for development of the Blackbox Expert and our DAI experiments. In the development stage of the Blackbox Expert, the knowledge needed to solve the Blackbox puzzle will be extracted from human experts. Human experts tend to disagree on the strategies that should be used to solve Blackbox. Thus, many of the rules in the knowledge base will be discovered incrementally by monitoring the performance of a prototype of the Blackbox Expert. When the Blackbox Expert is used in an organization, modifying its knowledge base would permit experimentation with the effects of updates or deficiencies in a single expert's knowledge base on the performance of the organization. The modularity of the knowledge base will also allow construction of different organizations.

Easy modification of the data structures used by the Blackbox Expert is required to support organizations and incremental development of the knowledge base. When the Blackbox Expert is a member of an organization and the data structures are put on the blackboard, the method required to access these data structures will change. Access to a data structure in the working memory requires a local access whereas access to a blackboard will require the Blackbox Expert to generate a request to an independent process, perhaps on a remote computer. Thus, the interface between the rules of the Blackbox Expert and the data structures must be consistent whether a data structure is located locally or on the blackboard. During the development phase of the Blackbox Expert, the functionality and implementation of the data structures will undergo many changes. These changes must be as transparent as possible to the rules in the knowledge base.

Our experiments using the Blackbox Expert for DAI research will depend on the ability to monitor the activities of individual experts. Experimental research conducted to evaluate the effectiveness of proposed solutions to the problems in DAI requires a comprehensive facility to monitor and record the pattern of rules fired and data structures accessed.

In order to meet the design requirements, we decided to use a combination of an expert system shell (Hayes-Roth et al. 1983) and object-oriented design techniques (Budd 1991). The flexibility that was required for incremental growth and modularity of the knowledge base was available as features of several expert system shells. Object-oriented design techniques were able to provide the data encapsulation required to permit the migration of the data structures used by the Blackbox Expert from its local working memory to the Blackboard. Object-oriented techniques also ensured that a modification to the data structures would not require global changes to the knowledge base.

The CLIPS expert system shell (Culbert 1989) and the C^{++} object-oriented programming language (Stroustrup 1987) were chosen for implementing the Blackbox Expert (Lyons 1990c,

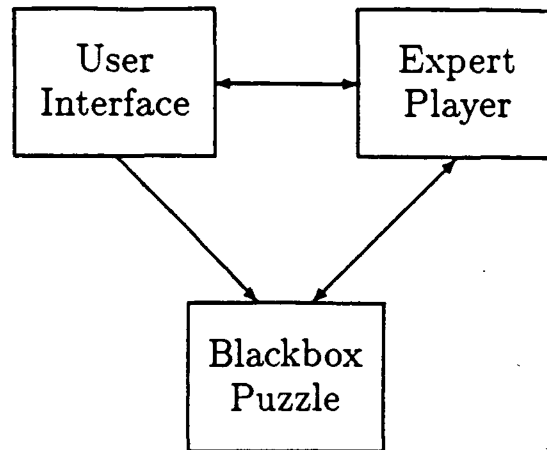


Figure 3: Components of the Blackbox Expert

Lyons 1990d). CLIPS and C^{++} were chosen together because CLIPS is written in C, the source code of CLIPS is available, CLIPS provides a flexible interface to external procedures, and C^{++} provides the data encapsulation facilities. CLIPS also provides the flexibility required for the development of the knowledge base and monitoring the rules that fire when the Blackbox Expert is operating.

As shown in Figure 3, the Blackbox Expert is composed of three major components: the User Interface, the Blackbox Puzzle, and the Expert Player. In this manner, each component of the problem solving system is encapsulated. The arrows depict the flow of data between the modules. The User Interface and the Blackbox Puzzle are implemented using C^{++} and the Expert Player is based on both C^{++} and CLIPS.

The Blackbox Puzzle is responsible for simulating the problem domain. It contains the rules describing the basic principles of the interactions that can occur between the beams and the balls in the Blackbox Puzzle. The Blackbox Puzzle maintains the data structure that contains the location of the balls within the Blackbox grid. It will receive the (X, Y) coordinates of the entry point for beams that are fired by the Expert Player. It determines the trajectory of beams and returns the (X, Y) coordinates of the exit point for the beams.

The Expert Player is responsible for solving the Blackbox puzzle. As shown in Figure 4, the Expert Player is composed of four modules, the Expert Manager, Working Memory, the Rule Base, and the CLIPS Inference Engine. These modules provide the rule base for solving the puzzle, the inferencing capability required to draw conclusions from the rule base, the data structures used by the rule base to maintain its current hypothesis as to the contents of the Blackbox, and an interface between the CLIPS Inference Engine and the other modules of the system.

The Expert Manager maintains control over the operation of the CLIPS Inference Engine and provides the interface between the components of the Expert Player, the Blackbox Puzzle, and the User Interface. It is responsible for the following: responding to commands from the User Interface, instructing the Rule Base to select a beam to fire, instructing the Blackbox Puzzle to fire the beam selected by the Rule Base, and instructing the Rule Base to analyze the result of a beam firing. The different operating modes for the Blackbox Expert, such as *single step*, are controlled by the Expert manager.

Working Memory stores the data structures that are required by the Expert Player. It contains the Expert Player's current hypothesis about the contents of the Blackbox, the number of balls hidden in the grid, the number of balls that have been found thus far, the size of the grid, and a list of the beams that have been fired. The Rule Base accesses Working Memory through a set of user-defined functions that have been added to the CLIPS shell. These functions allow the Rule Base to set the hypothesis for the contents of a square of the Blackbox, check the contents proposed for a grid square, or check a region of the grid to see if it is known to be empty, etc.

The Rule Base used for the Blackbox Expert is divided into 4 groups: Beam Selection, Beam

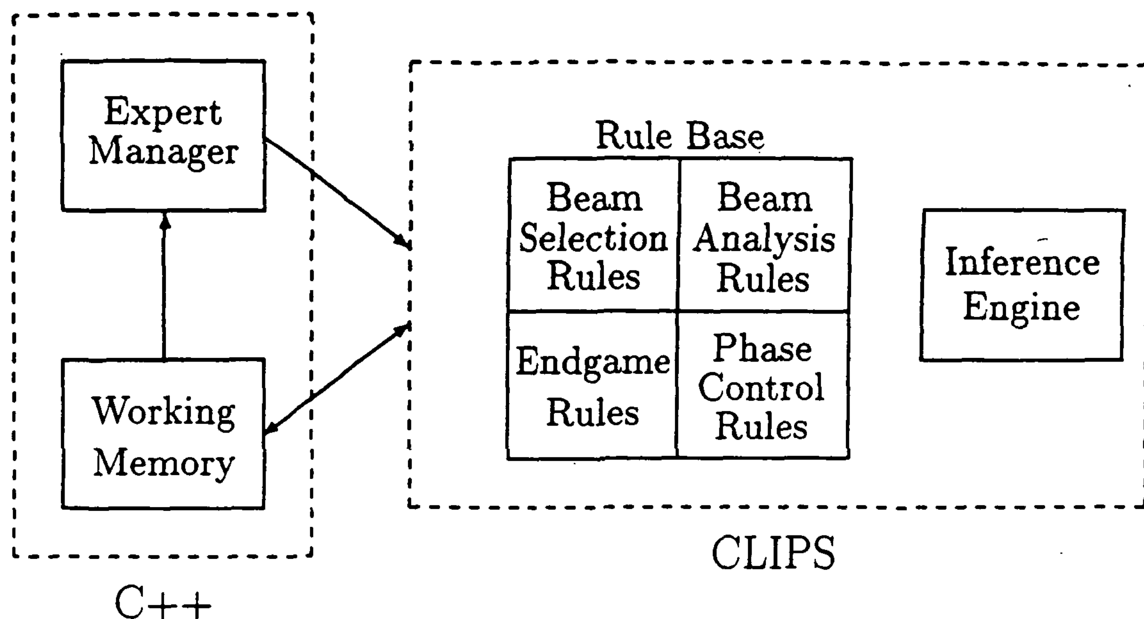


Figure 4: The Components of the Expert Player

Analysis, End Game, and Phase Control. These groupings reflect the different phases required for solving the Blackbox puzzle. The opening strategy for solving Blackbox requires a combination of both firing beams and analyzing the information gained from observing their exit points. The ending strategy for solving Blackbox requires different criteria for determining the contents of the remaining grid squares because there may not be any additional beams that can be fired which will provide more information.

The Beam Selection rules are responsible for determining the best beams for the Blackbox Expert to fire. The Expert Player maintains a list of beams that have yet to be fired along with a rating for each beam. The beam selection rules adjust these ratings based on the entry/exit locations observed for previous beams and the current hypothesis about the contents of the Blackbox grid.

Figure 5 shows a sample beam selection rule. This rule will fire when there is a beam that has not been fired which would potentially pass between two balls and the area of the grid that the beam would pass through is known to be empty. Once the rule fires, it asserts a fact that will cause an adjustment to the beam's rating. The user-defined function *isempty* is used by this rule to determine if the area of the hypothesis grid where the beam would pass before it reaches the ball is empty.

Beam analysis rules are responsible for determining the contents of the grid. Their findings are based on the data observed from the beams that have been fired, and the current contents of the hypothesis grid. Rules can determine that a grid square is empty or that it contains a ball. There are rules that can determine that the position of a ball placed on the hypothesis grid is known to be certain. When two or more rules present contradictory hypotheses, a *conflict* is detected. Thus, conflict resolution rules are required to determine which of the contradictory hypotheses is correct.

The Phase Control rules determine when the Blackbox Expert has finished the current phase. For example, when all the beam selection rules that can fire have fired, the phase control rules will take the highest rated beam and return it to the Expert Manager. The Expert Manager will then fire that beam, place the results in Working Memory and have the Expert Player enter the Beam Analysis phase. The Phase Control rules can also detect the start of a new phase such as the End Game.

The End Game rules are activated when there are very few beams left that the Blackbox Expert will fire. These rules will cause the Blackbox Expert to draw more "risky" conclusions as to the contents of the puzzle. This is justified because during this phase the Blackbox Expert is aware that it has observed most of the beams and it is unlikely to obtain any additional information by firing more beams. The End Game rules will also decide when the Blackbox Expert has finished

```
(defrule W91-24-left
  (phase selection)                  ; fire during beam selection phase
  (poss-ball-found ? ?row1 ?col $?)
  (poss-ball-found ? ?row2&:
          (>= ?row2 (+ ?row1 2)) ; the balls should be in the same
             ?col $?)            ; column, at least two rows apart
  (SHOTLEFT =(+ ?row1 1) 0 $?)       ; a beam has not been fired that
                                     ; will pass next to the upper ball
  (not (ADJUSTED-SHOT =(+ row?1 1)
                  0 W91-24-left) ; only adjust the beam once for
  (test (isempty ?row1 1 ?row2
                   (-?col 1)))   ; Is grid empty between the balls
                                 ; and the edge of the box
=>
  (assert (ADJUST-SHOT =(+ ?row1 1) 0 50 W91-24-left 0))
                                 ; then adjust the value of the beam
)
```

Figure 5: Sample Beam Selection Rule

solving the puzzle.

The User Interface is responsible for handling the interaction between humans and the Blackbox Expert. It allows a person to monitor and assert control over the Expert's progress as it solves the Blackbox puzzle. The User Interface is written in a combination of both C and C^{++} and runs under the X windowing environment. Both C and C^{++} are required because the widgets used in the X windowing environment are written in C while the procedural modules of the Blackbox Expert which carry out the functions requested by the humans are written in C^{++}. Figure 6 shows the Blackbox Expert's User Interface, with a sample game in progress.

The User Interface consists of four areas: the Real Grid, the Hypothesis Grid, the Dialog Window, and the Command buttons. These four areas allow a human to view the contents of the Blackbox grid as well as the current hypothesis of the Blackbox Expert as it solves the puzzle, allow the Blackbox Expert to annotate the actions it takes to solve the puzzle, and allow a human to issue commands to the Blackbox Expert. Both the Real Grid and Hypothesis Grid are custom made widgets designed specifically for the Blackbox Expert. The Dialog Window and the command buttons are implemented using the ATHENA widget set (O'Reilly and Assoc. 1990).

The Real Grid serves two purposes: it displays the contents of the Blackbox grid, and it allows humans to enter test cases for the Blackbox Expert to solve. Entering test cases is as easy as pointing the mouse cursor at the grid square where a ball is to be placed and pressing the left button. The right button of the mouse is used to remove a ball. Once a new test case has been entered into the Real Grid, the Save command button can be used to store the placement of the balls in a file.

The Hypothesis Grid displays the Blackbox Expert's current hypothesis about the contents of the Blackbox. It also displays the beams that have been fired by the Blackbox Expert. Areas of the grid about which the expert has not made any conclusions are marked with the letter U, for Unknown. Grid squares that the Blackbox Expert concludes contain a ball are marked with the letter B, while squares that the expert concludes are empty are marked blank. When the Blackbox Expert cannot definitely determine the contents of a grid square because two or more rules are in conflict over it, that square is marked with the letter C.

The Dialog Window is used by the Blackbox Expert to display messages that indicate the actions it is taking to solve the current test case of Blackbox. Messages are displayed in a variety of situations such as when beams are fired, and when conflicts are detected or resolved.

ORIGINAL PAGE IS
OF POOR QUALITY

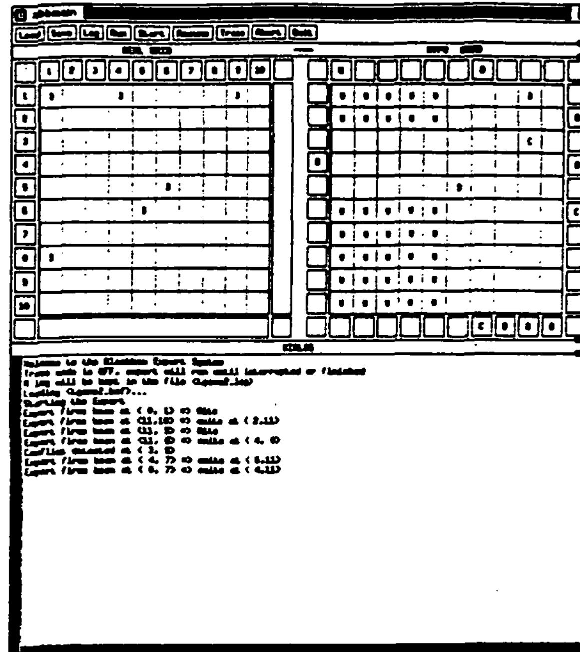


Figure 6: The Blackbox Expert's User Interface

The Command buttons perform the following functions:

Load: Loads a Blackbox test case from a file. A prompt is given to get the name of the file to load.
Save: Saves the current Blackbox test case to a file. A prompt is given to obtain the file in which to save the location of the balls.
Log: Toggles the logging function. If logging is turned on, a list of all the actions performed by the Blackbox Expert while solving the current test case of Blackbox is saved in a file.
Run: Allows the identification of a batch file from which the interface will take all its commands.
Start: Causes the Blackbox Expert to start solving the current test case of Blackbox.
Resume: When in trace mode, the Resume button causes the expert to fire the next beam and then analyze the results.
Trace: Toggles trace mode. When tracing is turned ON, the Blackbox Expert will stop after each beam is fired and analyzed; execution will resume upon the receipt of a Resume Command.
Abort: Aborts a game in progress.
Quit: Terminates the Blackbox Expert.

VALIDATION OF THE BLACKBOX EXPERT

The validation of computer software systems refers to the process of determining if the system satisfies the need for which it was designed (Boehm 1979). Validation of software systems is concerned with the factors that determine the system's usefulness such as: correctness of the results the system produces, its speed, its efficiency, its cost effectiveness, and many human factors. For our purposes, we will be concerned *only* with the correctness of the results produced by the system.
The validation of expert systems is analogous to the validation of general software systems but there are additional problems. The types of problems that are tackled by expert systems, called

Ill-Structured Problems (Simon 1973), do not have a complete specification. Thus, while the problem can seem to be intuitively clear, the criteria to be used for determining the accuracy of the solutions produced by an expert system, or even what constitutes an acceptable solution, are not clear. The lack of a "complete" specification for the problems solved by expert systems is a serious obstacle for the validation process.

The validation of expert systems strongly relies on the opinions of human experts. This has its problems (O'Keefe 1987). Achieving a consensus among a group of human experts is difficult. Generally, a human expert's time is limited and expensive. Producing a "complete" set of test cases for validating the expert system is typically not feasible.

Several models have been proposed for the development lifecycle of an expert system which include validation phases. The spiral model used for general software systems (Boehm 1988) is adapted to include phases that set the acceptable level of performance that is expected from the expert system at different stages of development(O'Keefe and Lee 1990). The heuristic testing approach of (Miller 1990) defines ten prioritized classes of fault types that can occur in an expert system and proposes a method for automatic generation of test cases. These test cases will test the expert system for the types of faults defined by each fault class. Generic tasks are used to decompose a knowledge base into conceptual units (Harrison and Ratcliffe 1991) in order to derive a standard methodology for knowledge base validation.

The development of the Blackbox Expert follows the modified spiral model of (O'Keefe and Lee 1990). Currently we have a "working" research prototype. The initial requirements analysis, requirements verification, and setting of acceptable levels of performance phases are complete. The following describes the validation phase we performed for our research prototype.

The Blackbox puzzle includes several features that facilitate the validation of the Blackbox Expert. Any solution that is proposed by a person for a test case of the Blackbox puzzle can be evaluated because the correct solution to the puzzle is known to another person. The time consumed by a computer or humans to solve each test case of Blackbox is between 10 minutes for someone with a lot of experience, and 30 minutes for a beginner. Therefore validation of the Blackbox Expert is not costly. Developing a population of human experts against whose performance the Blackbox Expert can be validated is simple because the effort required by a human to become skilled at solving Blackbox is not too large.

In consultation with a group of Blackbox experts, a metric has been devised to evaluate the *correctness* of a solution that is proposed for any test case of the Blackbox puzzle. The factors that were chosen to determine the correctness of a solution are: the number of balls that were correctly located, the number of locations of the grid (other than those which contain balls) whose contents are correctly identified, and the total value of the beams fired to solve the puzzle. As stated in the objectives of Blackbox, the best solution would have all the balls and locations correctly identified as well as a minimum total value for the beams that were fired.

The metric that was adopted is as follows:

$$SCORE = \left(2 - \frac{B^C - B^W}{B^M} - \frac{L^C - L^W}{L^T} + \frac{b^V}{b^T}\right) \times 100$$

where:

B^C: The number of correctly located mappable balls.
B^W: The number of incorrectly positioned balls.
B^M: The total number of mappable balls.
L^C: The number of locations of the grid which do not contain a mappable ball that are correctly identified.
L^W: The number of locations of the grid which do not contain a mappable ball that are incorrectly identified.
L^T: The total number of locations of the grid which do not contain mappable balls.
b^V: The total value of the beams fired to solve the puzzle.
b^T: The total number of entry/exit positions of the Blackbox.

This metric assigns a numerical value to a proposed solution of any test case of Blackbox and is used as a measure of the degree of its correctness. A solution with a lower $SCORE$ is considered

to be more correct than a solution with a higher score. This metric examines each of the factors th human experts identified as being relevant to assessing the correctness of a proposed solution o Blackbox. The SCORE function is sensitive both to correct and incorrect responses. The weight placed on each factor rank the number of balls correctly identified as the most important facto followed by the value of the beams fired and the number of correctly identified locations (thos not containing balls) is the least important. The minimum *SCORE* possible is zero and it occur when all balls are found, no beams are fired, and all locations are correctly identified. The maximun *SCORE* possible is 500 and it occurs when no balls are found, all shots are fired, and all location are incorrectly identified.

The Blackbox Expert is validated by comparing the correctness of the solutions it produce with human performance. A group of people, whose familiarity with Blackbox ranges from a fev hours of exposure to several years of exposure, are asked to solve a set of test cases for Blackbox Using the metric for evaluating the correctness of the solutions proposed by these people, eacl problem-solver receives a rating for their ability to solve Blackbox puzzles. The Blackbox Exper solves the same set of test cases and is assigned a rating for its ability. Then the rating obtained b the Blackbox Expert is compared with the ratings of the people, thus establishing the skill level o the Blackbox Expert.

Two people with several years of experience in solving Blackbox developed the set of tes cases to be used for validating the Blackbox Expert. The puzzles in the test set were given a rating o easy, medium, or hard. The two people who developed the test set participated in a group discussior with several other people who also had a lot of experience with Blackbox. The group focused it discussion on the factors that would determine the degree of difficulty of a test case of the Blackbo puzzle. Using the input from this discussion, the two people responsible for the test set determinec the criteria used to develop the test set and place each test case that was developed into one of the three categories.

The people who created the test set decided that the following features would contribute tc the complexity of a test case:

(a) The presence of unmappable grid squares.
(b) Beam entry and exit points that can be accounted for by many different trajectories through the grid.
(c) The presence of balls in the corners of the box.
(d) A positioning of balls that results in a large number of Hits and Reflections.

The presence of unmappable grid squares increases the complexity of a test case because it makes it difficult to decide when a solution has been found. Many people seem to have an aversion to leaving parts of the Blackbox grid unknown. They will actually convince themselves that they are able to pinpoint the locations of balls which actually are unmappable. People tend to always choose the simplest solution. Thus, when there are many possible trajectories that can account fo the entry and exit points of a beam people tend to make errors. If the puzzle-solvers do not confirm their choices for the locations of the balls by firing more beams, they risk making mistakes. The strategy used by many people when solving Blackbox is to work from the edges of the grid towards the center. Balls in the corners of the grid prevent a person from following this strategy thereby increasing the difficulty in solving the puzzle. Deflections provide a lot of information to the puzzle-solver because they often pinpoint the location of a ball and indicate many empty grid squares. A positioning of the balls that results in many hits and reflections is very difficult to solve as there is very little information with which one can determine the contents of the grid.

The rating of each person who participated in our validation experiment was done in two stages. The first stage was designed to be a learning phase and the second stage was the solution of the test set. The learning stage included a set of instructions explaining the basic principles of the Blackbox puzzle as well as the metric used for assessing the correctness of a proposed solution, a demonstration of how a person would solve the puzzle, and a set of sample games designed to demonstrate the principles and the metric described in the instructions. The test phase required each subject to solve the test cases which were presented to them in a random order. Even the subjects with a lot of previous exposure to Blackbox were required to go through the learning phase in orde to ensure that they fully understood the metric.

RESULTS: VALIDATION EXPERIMENT

Fifteen people participated in our validation experiment. They solved the 17 test cases in ou

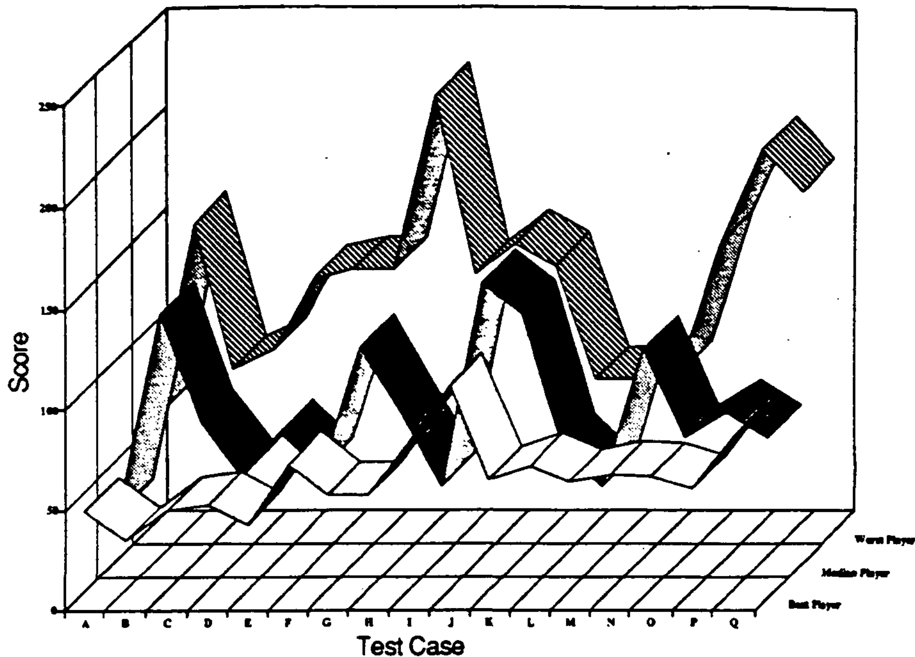


Figure 7: Best, Median, and Worst Player

Blackbox test set. Figure 7 shows the scores obtained by the people who had the best (lowest), median (middle), and worst (highest) average score for all seventeen test cases. Test cases A, B, and C were placed in the easy category, cases D, E, F, and G were placed in the medium category, and the other test cases were placed in the hard category. The person with the best score was also found to be the most consistent puzzle-solver. This consistency is evident from the narrow fluctuation in the scores for the person. The scores for the best person tend to rise slowly from easy to hard test cases. The person at the median has scores that fluctuate more widely than the best player. The person with the worst average score also experiences the largest variation in score.

The best, worst, and median scores for each test case are shown in Figure 8. Again, the lowest scores obtained by any person exhibit the least variation. The median scores vary more than the best scores and the worst scores have the largest variation. These scores also exhibit an upward trend when the easy, medium, and hard test cases are compared.

The average scores and total number of errors made in placing balls in the Blackbox grid by the people who solved the test set are shown in Table 1. Both the average score and total errors made in placing the balls increase when comparing the easy, medium, and hard test cases. As expected, this trend seems to suggest that the performance of the people when solving the test cases from each of the three groups in our test set is different. In order to validate this assumption, we performed an analysis of variance to determine if the difference that is observed in the mean scores of the three groups can be accounted for by the variance in the scores of all the test cases solved. The ANOVA table is shown in Table 2. The F ratio obtained with 2 and 252 degrees of freedom is 13.42. An F ratio of 4.69 or greater is needed for significance with a confidence level of 99%, thus we can reject the null hypothesis that $\mu_{easy} = \mu_{medium} = \mu_{hard}$.

Having determined that the average scores for the three groups are statistically different, we must now examine the individual differences between the groups. Table 2 shows the confidence intervals for the pairwise comparisons of the means of the groups in the test set. These comparisons are done using an F value of $F_{2,252,.95} = 3.035$. As the confidence intervals for $\mu_{easy} - \mu_{medium}$ and $\mu_{easy} - \mu_{hard}$ do not contain zero we can reject the null hypotheses $(\mu_{easy} - \mu_{medium}) = 0$ and $(\mu_{easy} - \mu_{hard}) = 0$. Thus, $\mu_{easy} < \mu_{medium}$ and $\mu_{easy} < \mu_{hard}$. However, the confidence interval for $\mu_{medium} - \mu_{hard}$ does contain zero, which does not permit us to reject the null hypothesis $(\mu_{medium} - \mu_{hard}) = 0$. Thus, there is a statistical difference between the easy and medium groups,

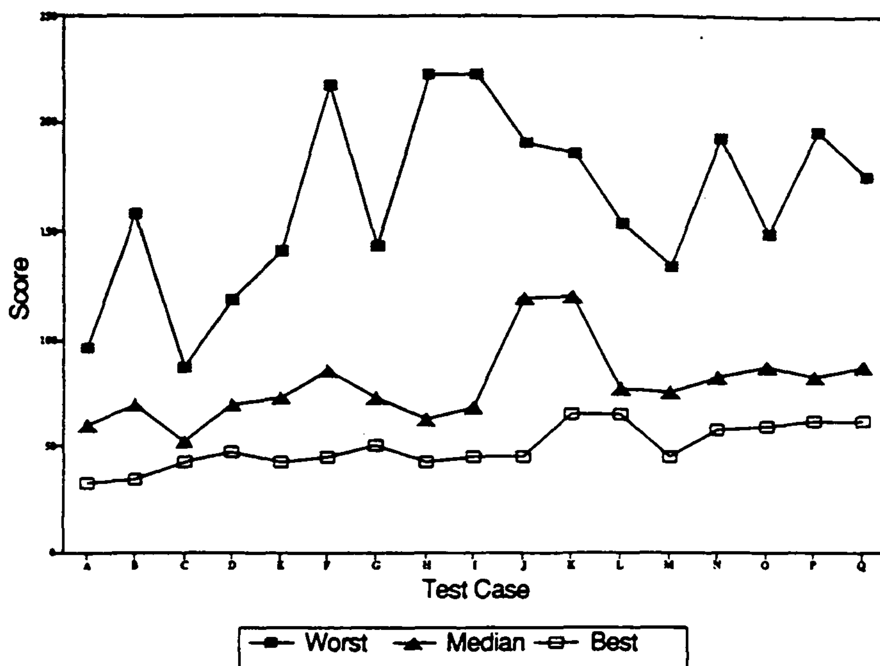


Figure 8: Best, Median, and Worst Scores

	Total Ball Errors			
SUBJECT	EASY	MED	HARD	ALL
1	8.0	4.0	18.0	30.0
2	0.0	0.0	2.0	2.0
3	0.0	0.0	9.0	9.0
4	0.0	0.0	4.0	4.0
5	2.0	1.0	9.0	12.0
6	2.0	2.0	23.0	27.0
7	0.0	0.0	6.0	6.0
8	0.0	6.0	8.0	14.0
9	0.0	0.0	7.0	7.0
10	0.0	6.0	21.0	27.0
11	0.0	8.0	17.0	25.0
12	0.0	4.0	13.0	17.0
13	2.0	20.0	33.0	55.0
14	8.0	15.0	30.0	53.0
15	10.0	22.0	27.0	59.0
AVG	2.1	5.9	15.1	23.1

	AVG SCORE			
SUBJECT	EASY	MED	HARD	ALL
1	80.0	72.2	84.7	80.9
2	60.0	66.3	69.3	66.9
3	45.0	56.3	73.5	64.4
4	69.2	74.4	79.1	76.2
5	64.7	80.1	82.7	78.9
6	50.0	65.2	97.8	81.7
7	57.5	62.5	78.2	70.8
8	61.7	107.0	88.1	87.9
9	59.2	62.5	78.6	71.4
10	59.2	80.8	93.3	84.3
11	66.7	93.9	114.7	101.4
12	54.2	76.6	93.5	82.6
13	46.7	137.4	130.4	117.3
14	86.1	118.2	124.4	116.2
15	102.6	125.2	141.1	130.6
AVG	64.2	85.2	95.3	87.4

Table 1: Average Score and Total Errors in Placing Balls

Source of Variation	Sum of Squares	df	Mean Square	F
Between group	33915.97	2.00	16957.99	13.42
Within group	318506.48	252.00	1263.91	
Total	352422.45	254.00		

	Confidence	Interval
Pairwise Comparison	Lower Limit	Upper Limit
Easy-Medium	-38.34	-3.79
Medium-Hard	-23.44	3.32
Easy-Hard	-46.01	-16.24

	Confidence	Interval
Group Comparison	Lower Limit	Upper Limit
Easy-(Medium,Hard)	-40.77	-11.42
Hard-(Easy,Medium)	9.38	31.81

Table 2: Analyses of Variance

	Average Score			
	EASY	MEDIUM	HARD	ALL
PEOPLE	64.2	85.2	95.3	87.4
EXPERT	64.2	73.6	88.9	80.9
	Average Ball Error			
PEOPLE	2.1	5.9	15.1	23.1
EXPERT	0.0	2.0	12.0	14.0

Table 3: Average Score and Ball Errors

the easy and hard groups, but not the medium and hard groups. The factors we used to place the different test cases into the groups are valid. However, the difference between the medium and hard groups is not confirmed.

The last set of tests to be performed on our test set are the group comparisons shown in Table 2. The confidence intervals are given for $\mu_{easy} - (\frac{\mu_{medium}+\mu_{hard}}{2})$ and $\mu_{hard} - (\frac{\mu_{easy}+\mu_{medium}}{2})$ using an F value of $F_{2,252,.95} = 3.035$. In both cases, we can reject the null hypothesis that $(\mu_{easy} - (\frac{\mu_{medium}+\mu_{hard}}{2})) = 0$ and $(\mu_{hard} - (\frac{\mu_{easy}+\mu_{medium}}{2})) = 0$. Thus, the easy group is different from the average of the medium and hard groups and the hard group is different from the average of the easy and medium groups.

The performance of the Blackbox Expert on the test cases is shown in Figure 9. The Blackbox Expert is compared to that of the people who were rated as the best, median, and worst players. Except for one test case (I), the best player performed better than the Blackbox Expert. The Blackbox Expert performed better than the worst player in 15 of the 17 test cases. The Blackbox Expert performed better than the median player in 7 of the 17 test cases. The two test cases (C and P) where the Blackbox Expert performed poorly compared to the worst player indicate a deficiency in the knowledge base. Both test cases C and P have balls located near the corners of the Blackbox grid. The knowledge base of the Blackbox Expert is lacking rules to determine when balls are located near the corners of the Blackbox.

The average score and the total number of errors made placing balls by the humans and the Blackbox Expert are shown in Table 3. The Blackbox Expert on average made fewer errors locating balls than the humans. The lowest total number of errors (2 errors in 17 test cases) was made by a person with several years of experience solving the Blackbox puzzle as seen in Table 1. Also, the Blackbox Expert had a better average score on each group of test cases in the test set except on the easy test cases where it had the same average score. When the total number of errors in locating balls is considered, the Blackbox Expert ranks 7th compared to the people.

The average score obtained by the Blackbox Expert for each group of test cases as well as the entire test set is compared to the average score obtained by the people in Figure 10. The Blackbox Expert ranks 10th on the easy test cases, 7th on the medium and hard test cases, and 7th

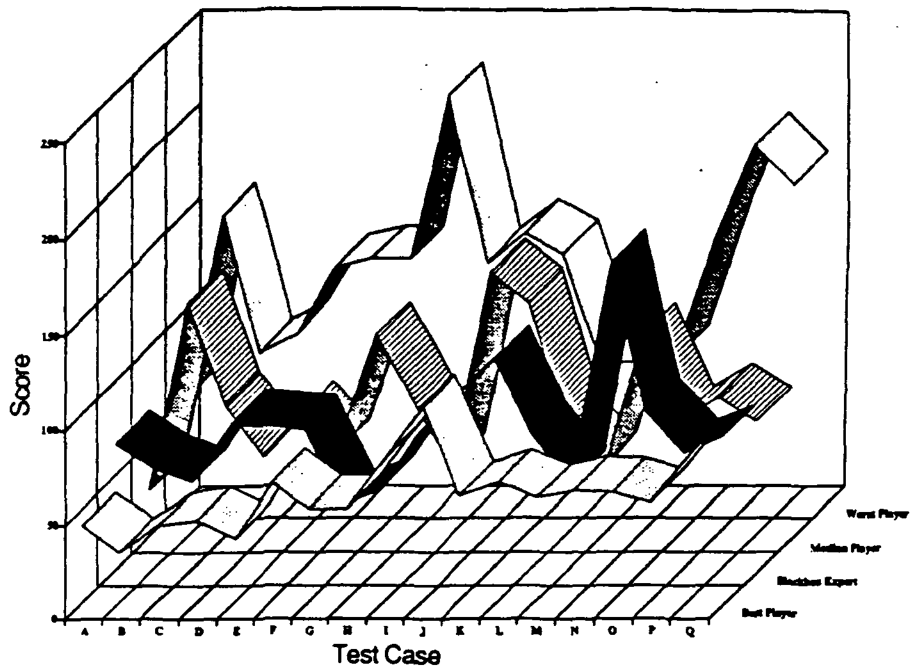


Figure 9: Scores of the Blackbox Expert

on the entire test set. The improvement observed in the Blackbox Expert's ranking on the medium and hard test cases occurs because even though the Blackbox Expert and the humans can find all the balls in the easy test cases, the Blackbox Expert requires more beams to solve the test cases. In the case of the medium and hard test cases, the Blackbox Expert still tends to fire more beams than the humans. However, on average it makes fewer errors which allows it to improve its position.

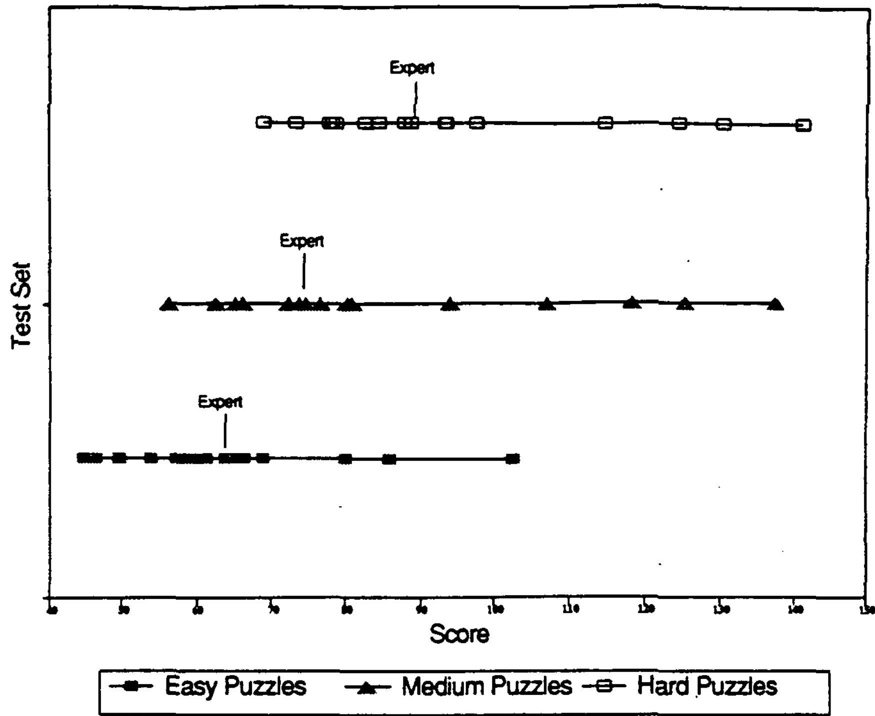
CONCLUSIONS AND FURTHER RESEARCH

In order to validate the expert system described in this paper, we created 17 test cases. Using a set of criteria extracted from human Blackbox experts, the test cases were placed into three groups easy, medium, and hard. Based on a metric defined in this paper and the results of the 15 people solving each of the 17 test cases, the groups in the test set were statistically tested to determine if the mean scores for the test cases in each group were different. It was found that the means of the easy and hard groups are statistically different as are the means of the easy and medium groups. However, the difference between the means of the medium and hard groups is not significant. Further statistical tests have shown that the medium group can be combined either with the hard or with the easy group to form two statistically different groups of test cases.

We have briefly described the structure and design of the Blackbox Expert in this paper. It consists of 300 rules and 8000 lines of source code of which approximately 25% is comments. Its performance is compared with that of the 15 people. This comparison included both the SCORE metric developed in the paper and the total number of errors committed in placing balls. Overall the Blackbox Expert ranks 7th with respect to SCORE and 7th with respect to total errors placing balls.

By analyzing the games in which the people performed better than the Blackbox Expert, we notice that the expert system can be improved in three ways: the rule base can be enhanced to account for the cases where balls are placed in corner squares; the beam selection rules can be improved; and the conflict resolution rules can be improved by studying the ball placement errors committed by the expert system. The additional knowledge required for the improvements to the rule base of the Blackbox Expert can now be obtained from the people who solved the test set. During the next phases of the modified spiral model used for the development of expert systems, this information will be useful.

The Blackbox Expert, the test cases, and the details of the humans solving the test cases



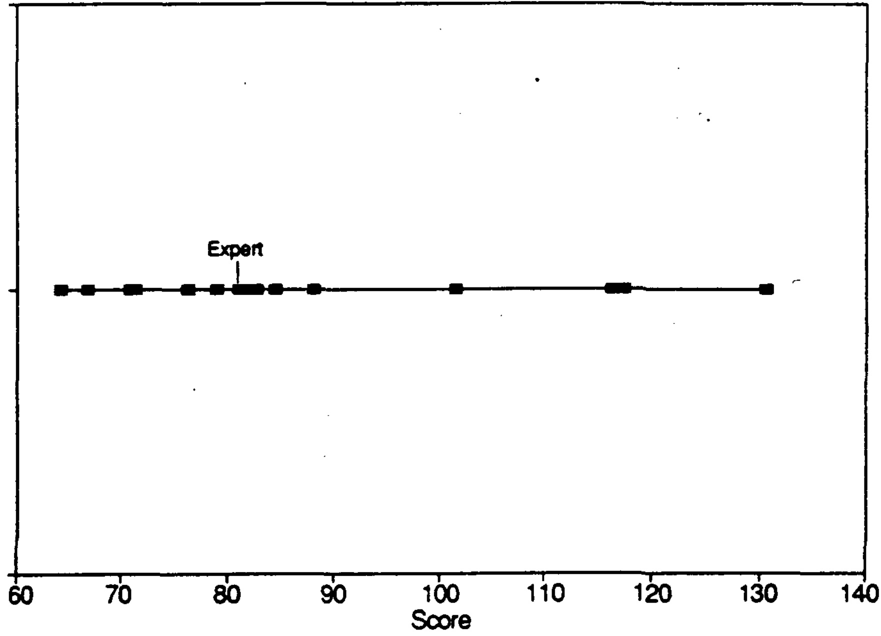


Figure 10: Blackbox Expert versus Humans

ORIGINAL PAGE IS
OF POOR QUALITY

will all be useful for several experimental research projects. In particular, we are interested in co-operative problem solving with multiple expert systems. The optimality of different organizations effects of "Information Deficit", and the effectiveness of different planning and communication protocols using the blackboard architecture are some of the problems that our group of two faculty members and five graduate students are investigating.

ACKNOWLEDGEMENTS

There are many people who helped us to collect the data for this experiment by solving the test cases of Blackbox. We thank them for their participation. The indispensable help of Le Hoc Duong in the development of the rule base for the Blackbox Expert is gratefully acknowledged. We thank Kristina Pitula who made her Blackbox expertise available to us for developing the test set. We are grateful to Dr. Alun Preece for his valuable suggestions and comments concerning our validation procedure.

REFERENCES

Boehm, B.W. (1979). Software Engineering: R & D trends and defense needs, *Research Directions in Software Technology*, Wegener, P. (Ed), M. I. T. Press, Cambridge Mass.

Boehm, B.W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, May, pp. 61-72.

Budd, T. (1991). *Object-Oriented Programming*, Addison Wesley.

Culbert, C. (1989). CLIPS Reference Manual, *Artificial Intelligence Section, Johnson Space Center*, Houston.

Durfee, E.H. and Lesser, V.R. (1986). Incremental Planning to Control a Blackboard Based Problem-Solver, *AAAI-86*, pp. 58-64.

Durfee, E.H., Lesser, V.R. and Corkill D.C. (1987). Cooperation Through Communication in a Distributed Problem Solving Network, *in Distributed Artificial Intelligence*, Morgan Kaufmann.

Gasser, L. (1987) The 1985 Workshop on Distributed Artificial Intelligence, *AI Magazine*, Vol. 8, No. 2.

Ginsberg, M.L. (1987). Decision Procedures, *in Distributed Artificial Intelligence*, Morgan Kaufmann, California.

Grossner, C. and Radhakrishnan (1990). Organizations for Cooperating Expert Systems, *22nd Southeast Symposium on System Theory*, Tenn.

Harrison, R.P., and Ratcliffe, P.A. (1991). Towards Standards for the Validation of Expert Systems, *Expert Systems with Applications*, Vol. 2, No. 4, pp. 251-258.

Hayes-Roth, F., Waterman, D.A. and Lenat, D.B. (Eds.) (1983). *Building Expert Systems*, Addison Wesley.

Jagannathan, V., Dodhiawala, R. and Baum, L.S. (1989). *Blackboard Architectures and Applications*, Academic Press.

Lyons, J. and Grossner, C (1990). A Blackbox Expert System: User Requirements, *Technical Report*, Computer Science Dept., Concordia University.

Lyons, J. and Grossner, C (1990). A Blackbox Expert System: Software Requirements Specification, *Technical Report*, Computer Science Dept., Concordia University.

Lyons, J. and Grossner, C (1990). A Blackbox Expert System: Preliminary Design, *Technical Report*, Computer Science Dept., Concordia University.

Lyons, J. and Grossner, C (1990). A Blackbox Expert System: Detailed Design, *Technical Report*, Computer Science Dept., Concordia University.

Miller, L.A. (1990). Dynamic Testing of Knowledge Bases Using the Heuristic Testing Approach, *Expert Systems with Applications*, Vol. 1, No. 3, pp. 249-270.

O'Keefe, R.M., Balci, O. and Smith, P.E. (1987). Validating Expert System Performance, *IEEE Expert*.

O'Keefe, R.M. and Lee, S. (1990). An Integrative Model of Expert System Verification and Validation, *Expert Systems with Applications*, Vol. 1, No. 3, pp. 231-236.

O'Reilly-Staff (1990). X Toolkit Intrinsics Reference Manual. *O'Reilly and Associates*, Vol. 5.

Parsaye, K. and Chignell, M. (1988). *Expert Systems for Experts*, John Wiley.

Pitula, K., Radhakrishnan, T. and Grossner, C. (1990). Distributed Blackbox: A Test Bed for Distributed Problem Solving, *Int. Phoenix Conf. on Computers and Communications*, Phoenix.

Simon, H. (1973). The Structure of Ill-Structured Problems, *Artificial Intelligence* , Vol. 4, 1973.

Smith, R.G. (1980). The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver, *IEEE Transactions on Computers*, Vol. C-29, No. 12, pp. 1104-1113.

Stroustrup, B. (1987). *The C^{++} Programming Language*, Addison Wesley.

N92-16603

TESTING VALIDATION TOOLS ON CLIPS-BASED EXPERT SYSTEMS

C.L. Chang, R.A. Stachowitz and J.B. Combs

Lockheed Missiles & Space Company, Inc.
Software Technology Center, B/254E, O/96-10
3251 Hanover Street
Palo Alto, Ca 94304

Abstract. The Expert Systems Validation Associate (EVA) is a validation system which was developed at the Lockheed Software Technology Center and Artificial Intelligence Center between 1986 and 1990. EVA is an integrated set of generic tools to validate any knowledge-based system written in any expert system shell such as CLIPS, ART, OPS5, KEE and others. Many validation tools have been built in the EVA system. In this paper, we describe the testing results of applying the EVA validation tools to the Manned Maneuvering Unit (MMU) Fault Diagnosis, Isolation, and Reconfiguration (FDIR) expert system, written in C Language Integrated Production System (CLIPS), obtained from the NASA Johnson Space Center.

INTRODUCTION

The Expert Systems Validation Associate (EVA) is a validation system developed at the Lockheed Software Technology Center and Artificial Intelligence Center. Its goal was to develop generic and flexible tools to validate any knowledge-based system written in any expert system shell such as CLIPS, ART, OPS5, KEE and others. We achieved this goal by using the *metaknowledge approach*. Metaknowledge, or knowledge about knowledge, describes constraints on the knowledge that can be used for checking redundancy, completeness, consistency, and correctness of a knowledge base. We realized that we can not standardize validation criteria for all applications. Each application has its own statements to be validated, but they can be represented by means of the metapredicates.

From 1986 to 1990, we implemented many different checkers (Stachowitz et al. 1987,1991, Chang et al. 1988,1989,1990, McGuire et al. 1990) for checking different types of errors in a knowledge base. In order to see if the checkers were sufficient to detect anomalies and errors, we tested them on a real expert system, the Manned Maneuvering Unit (MMU) Fault Diagnosis, Isolation, and Reconfiguration (FDIR) expert system, written in C Language Integrated Production System (CLIPS), we obtained from the NASA Johnson Space Center. To validate the FDIR expert system, we implemented a CLIPS translator to convert FDIR's CLIPS rules into the internal knowledge representations in Prolog used by our checkers. EVA only requires a new translator for a new shell because its checkers were built to be independent of any particular expert system shell. This paper describes the testing results of applying the EVA validation tools to the FDIR expert system.

MMU FDIR AUTOMATION TASK

The MMU FDIR automation task is described in (Lawler and Williams 1988). The primary objectives of the task were to investigate the potential of automating the fault diagnosis, isolation and reconfiguration (FDIR) function of the MMU currently performed by the MMU pilot.

The problem of this task is to determine the types of all possible failure scenarios, appropriate instrumentation for monitoring the MMU state and supporting diagnosis, and appropriate actuators for executing the failure recovery.

The MMU system is represented by a three-level architectural diagram. At level 0, MMU is a black box whose inputs and outputs are:

- Inputs of MMU:
  - MMU Torques
  - Automatic Attitude Hold (AAH) Select
  - Gyro Power
  - Control-Electronic-Assembly-A (CEA-A) Power
  - Control-Electronic-Assembly-B (CEA-B) Power
  - CEA-A Isolate
  - CEA-B Isolate
  - Translation Commands
  - Rotation Commands
  - Xfeed Valves
- Outputs of MMU:
  - Tank-A Pressure (Gas Flow Rate)
  - Tank-B Pressure (Gas Flow Rate)
  - Thruster-A Firings
  - Thruster-B Firings

At level 1, MMU is decomposed into 3 components, namely, Gyros, CEA Assembly (CEA-A and CEA-B), and Tank/Thruster Assembly (A and B Thrusters). Their inputs and outputs are:

- Inputs of Gyros:
  - MMU Torques
  - Gyro Power
- Outputs of Gyros:
  - Gyro Rotation Commands

- Inputs of CEA Assembly:
  - Automatic Attitude Hold (AAH) Select
  - Control-Electronic-Assembly-A (CEA-A) Power
  - Control-Electronic-Assembly-B (CEA-B) Power
  - Translation Commands
  - Rotation Commands
  - Gyro Rotation Commands
- Outputs of CEA Assembly:
  - Valve-Drive-Amplifier-A Commands
  - Valve-Drive-Amplifier-B Commands

- Inputs of Tank/Thruster Assembly:
  - CEA-A Isolate
  - CEA-B Isolate
  - Xfeed Valves
  - Valve-Drive-Amplifier-A Commands
  - Valve-Drive-Amplifier-B Commands
- Outputs of Tank/Thruster Assembly:
  - Tank-A Pressure (Gas Flow Rate)
  - Tank-B Pressure (Gas Flow Rate)

Thruster-A Firings
Thruster-B Firings

At level 2, the CEA Assembly is further decomposed into CEA-A and CEA-B each of which knows about the power status of the other. Similarly, the Tank/Thruster Assembly is broken down into the components of side A and side B coupled through the crossfeed valves.

The architectural diagram provides a simple description of the causal effect of MMU. To implement the recovery procedures, MMU provides one or more of the recovery options: do nothing and continue mission, turn on/off CEA-A or CEA-B, turn on/off gyro power, turn on/off AAH, open/close crossfeed valves, or abort mission (return to orbiter or call for rescue).

Measurements and switch settings of different parts/components in MMU are taken by instruments to assess the state of the MMU system and components. Each measurement is analyzed for the amount of diagnostic capacity it contributes to the FDIR system. The measurements and switch settings are processed and converted into symbolic information represented by a set of initial facts in CLIPS. The initial facts are processed by CLIPS rules in the FDIR expert system to generate diagnosis and recovery actions.

CLIPS TRANSLATOR

We implemented a CLIPS translator to convert FDIR's CLIPS rules into the internal Prolog representations used by the EVA checkers. The CLIPS translator needs to handle many special CLIPS constructs such as multifield variables, nested AND/OR, variable assignments, if...then...else, while...do. Since Prolog does not support these constructs, the CLIPS translator needs to perform normalizations and semantic translations. Therefore, it is possible that a CLIPS rule may be translated into many Prolog Horn clauses. For example, the CLIPS rule:

```
(defrule pull-block-c
    ?s <- (stack $?stack-1 c $?stack-2)
    =>
    (assert (stack $?stack-1 $?stack-2))
    (retract $s))
```

where $?stack-1 and $?stack-2 are multifield variables is translated into

```
rule('pull-block-c',
    [stack(X),
     append(Stack_1, [c | Stack_2], X),
     append(Stack_1, Stack_2, Z)],
    [assert(stack(Z)),
     retract(stack(X))]).
rule('append-0',
    [ ],
    [append([ ],L,L)]).
rule('append-1',
    [append(T,L,R)],
    [append([H | T],L,[H | R])]).
```

VALIDATION OF THE FDIR EXPERT SYSTEM

The FDIR expert system consists of 104 CLIPS rules. The CLIPS translator converted these CLIPS rules into 357 normalized internal rules which were then checked by the EVA checkers. The goal of our experiments was to detect if redundancy, incompleteness, inconsistency, or incorrectness is present in FDIR's CLIPS rules. The following were the results we found:

Redundancy

There are indeed redundancies in FDIR's CLIPS rules. The redundancy checker detected redundancies in the following 5 CLIPS rules:

```
;1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;improper CEA behavior
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;logic for (no aah) or (no gyro movement)and(aah on) - prime mode
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule cea-test-input-null
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
        (vda ?side ?thrust on)
=>
        (assert (failure cea))
        (printout crlf "failure - vda signal was sent to " crlf)
        (printout "thrusters without intended command "crlf)
)


;2
;pos roll gyro input
(defrule cea-a-gyro-input-roll-pos-6
       (aah off) (gyro on)
       (gyro movement roll pos)
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
        (vda a ?m on)

=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "aah failed to correct pos roll")
)


;3
(defrule cea-b-gyro-input-roll-pos-6
       (aah off) (gyro on)
       (gyro movement roll pos)
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
        (or
        (vda b r4 off)
        (vda b l1 off)
        (vda b ?n&~r4&~l1 on)
        )
```

```
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "aah failed to correct pos roll")
)


;4
;pos roll gyro input
(defrule gyro-input-roll-pos-backup-a-6
       (aah off) (gyro on)
       (gyro movement roll pos)
        (not (checking thrusters))
        (side a on)
        (side b off)
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
        (or
        (vda a r4 off)
        (vda a l1 off)
        (vda a ?n&~r4&~l1 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "cea failure on side a")
)


;5
(defrule gyro-input-roll-pos-backup-b-6
       (aah off) (gyro on)
       (gyro movement roll pos)
        (not (checking thrusters))
        (side a off)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
        (or
        (vda b r4 off)
        (vda b l1 off)
        (vda b ?n&~r4&~l1 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "cea failure on side b")
)
```

where the last 4 rules contain parts *subsumed* by the first rule. For example, in the second rule, *cea-a-gyro-input-roll-pos-6*, the conditions

```
        (aah off)
        (rhc roll none pitch none yaw none)
        (thc x none y none z none)
```

```
(vda a ?m on)
```

are subsumed by the left hand side (LHS) of the first rule, and both rules have the same action, (assert (failure cea)), in the right hand side (RHS). This type of redundancy, which is not obvious to the naked eye, can be detected only when the rules are normalized.

Incompleteness

An input to the FDIR expert system is a set of initial facts. The requirements that can be used for checking incompleteness should describe what are all the legal inputs. That is, we want to verify if every input can be processed by the FDIR expert system. Unfortunately, these requirements are not specified in (Lawler and Williams 1988). We could only manually deduce from the 6 test cases given in Lawler and Williams' report and from the MMU model given in the previous section that an input contains one or more instances of each of the following CLIPS relations:

```
(defrelation aah
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type WORD) (allowed-words on off)))

(defrelation fuel-used-a
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type NUMBER) (range 0 ?VARIABLE)))

(defrelation fuel-used-b
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type NUMBER) (range 0 ?VARIABLE)))

(defrelation side
  (min-number-of-fields 3)
  (max-number-of-fields 3)
  (field 2 (type WORD) (allowed-words a b))
  (field 3 (type WORD) (allowed-words on off)))

(defrelation xfeed-a
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type WORD) (allowed-words open closed)))

(defrelation xfeed-b
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type WORD) (allowed-words open closed)))

(defrelation tank-pressure-current
  (min-number-of-fields 3)
  (max-number-of-fields 3)
  (field 2 (type WORD) (allowed-words a b ab))
  (field 3 (type NUMBER) (range 0 ?VARIABLE)))

(defrelation tank-pressure-was
  (min-number-of-fields 3)
```

```
  (max-number-of-fields 3)
  (field 2 (type WORD) (allowed-words a b ab))
  (field 3 (type NUMBER) (range 0 ?VARIABLE)))

(defrelation gyro
  (min-number-of-fields 2)
  (max-number-of-fields 2)
  (field 2 (type WORD) (allowed-words on off)))

(defrelation gyro
  (min-number-of-fields 4)
  (max-number-of-fields 4)
  (field 2 (type WORD) (allowed-words movement))
  (field 3 (type WORD) (allowed-words none pitch roll yaw))
  (field 4 (type WORD) (allowed-words none pos neg)))

(defrelation vda
  (min-number-of-fields 4)
  (max-number-of-fields 4)
  (field 2 (type WORD) (allowed-words a b))
  (field 3 (type WORD) (allowed-words b1 b2 b3 b4
                                      f1 f2 f3 f4
                                      l1 l2 l3 l4
                                      r1 r2 r3 r4
                                      u1 u2 u3 u4
                                      d1 d2 d3 d4))
  (field 4 (type WORD) (allowed-words on off)))

(defrelation rhc
  (min-number-of-fields 7)
  (max-number-of-fields 7)
  (field 2 (type WORD) (allowed-words roll))
  (field 3 (type WORD) (allowed-words none pos neg))
  (field 4 (type WORD) (allowed-words pitch))
  (field 5 (type WORD) (allowed-words none pos neg))
  (field 6 (type WORD) (allowed-words yaw))
  (field 7 (type WORD) (allowed-words none pos neg)))

(defrelation thc
  (min-number-of-fields 7)
  (max-number-of-fields 7)
  (field 2 (type WORD) (allowed-words x))
  (field 3 (type WORD) (allowed-words none pos neg))
  (field 4 (type WORD) (allowed-words y))
  (field 5 (type WORD) (allowed-words none pos neg))
  (field 6 (type WORD) (allowed-words z))
  (field 7 (type WORD) (allowed-words none pos neg)))
```

The types of incompleteness we checked were deadend and unreachable literals. A deadend literal is a LHS literal of a rule that does not match with all input facts or all RHS literals in the knowledge base. Obviously, any rule containing a deadend literal will not fire. An unreachable literal is a RHS literal of a rule that never appears in the LHS of a rule. In our experiment, we did not find deadend literals. However, we did find 2 unreachable literals, (assert (failure-thrusters-with-xfeed)), and (assert (checking thruster)) in the following rule:

```
(defrule xfeed-fuel-reading-test-general
        (declare (salience -10))
        ?x <- (xfeed-a open)
        ?y <- (xfeed-b open)
        (fuel-used-a ?fuel-a)
        (fuel-used-b ?fuel-b)
        (tank-pressure-was ab ?p-old)
        (tank-pressure-current ab ?p-new)
        (test (!= (- ?p-old (+ ?fuel-a ?fuel-b)) ?p-new))
        (side b on)
        (side a on)
=>
        (retract ?x ?y)
        (assert (xfeed-a closed))
        (assert (xfeed-b closed))
        (assert (failure-thrusters-with-xfeed))
        (printout crlf "failure occurred while executing thruster commands")
        (printout crlf crlf "xfeed is open, testing sides after closing xfeed")
        (printout crlf crlf)
        (assert (checking thruster))
)
```

It turned out that (assert (checking thruster)) is a typo. The correct one should be (assert (checking thrusters)).

Inconsistency

In the EVA system, we provide the flexibility of letting the user define his own inconsistency criteria for his expert system. This is done through specifications of constraints on input facts and asserted facts. We introduced some new constructs such as *defambiguity*, *defincompatibility* and *defimply* for this purpose. For example, if we state

```
(defambiguity (assert ?fact) (retract ?fact))
```

then an ambiguity will occur when there is a set of input facts that can lead to a situation where a rule may fire to assert ?fact and another rule may fire to retract ?fact. If we state

```
(defincompatibility (xfeed-a closed) (xfeed-a open))
```

then we have an inconsistency when both facts, (xfeed-a closed) and (xfeed-a open), simultaneously exist in the short term memory. If we state

```
(defimply
    (gyro movement ~none ?v)
=>
    (!= ?v none))
```

then we have to make sure whenever a fact about gyro is asserted, if its third field is not 'none' then its fourth field must not be 'none'. Adhering to EVA standards, we use the same syntax of CLIPS in this case for constraint specifications so that the user does not have to learn a new syntax.

Based upon some of the above constraint specifications, our logic checker detected anomalies where one rule added a fact, followed by the deletion of the same fact by another rule, and in between these two operations no new operations were performed, as specified by the above *defambiguity* statement.

Incorrectness

There are two major methods, namely, verification and testing (Chang et al. 1990), for check-

ing correctness of an expert system. Both these methods assume that requirements on functional behaviors of the system exist. Behavior requirements specify input-output relationships of the system. Unfortunately, such requirements are not specified in (Lawler and Williams 1988). The approach given in their report is to perform testing of the FDIR expert system with a few manually created test cases, and then manually evaluate the output produced by the system.

The EVA system has a prototype tool called test case generator which automatically generates test cases that satisfy the input specifications given in the previous sections. However, we are not in a position to tell what the correct outputs would be for the generated test cases. The performance for them has to be evaluated by experts in the field.

CONCLUDING REMARKS

We have described the testing results of the validation experiments we performed for the FDIR expert system. The experiments are very useful. They not only tell us that EVA is a useful validation system to detect errors/anomalies in a knowledge base, but also reveal to us what requirements and information EVA needs in order to perform thorough validation of an expert system. Because of these experiments, we gained better insight on how to build an expert system development environment that can nicely integrate requirement specification, system prototyping/coding, and system verification, validation and testing.

REFERENCES

Chang, C.L., and Stachowitz, R.A. [1988] "Testing expert systems," *Proc. Space Operations Automation and Robotics (SOAR 88) Workshop*, Wright State University, Dayton, OH.

Chang, C.L., Stachowitz, R.A., and Combs, J.B. [1989] "Testing integrated knowledge-based systems," *Proc. of IEEE International Workshop on Tools for Artificial Intelligence*, 1989, IEEE Computer Society Press.

Chang, C.L., Combs, J.B., and Stachowitz, R.A. [1990] "A report on the expert systems validation associate (EVA)," *Journal of Expert Systems With Applications*, Vol. 1, pp.217-230, 1990, Pergamon Press, Maxwell House, Fairview Park, Elmsford, New York 10523.

Chang, C.L., Stachowitz, R.A., and Combs, J.B. [1990] "Validation of nonmonotonic knowledge-based systems," *Proc. of the IEEE Second International Conference on Tools for Artificial Intelligence*, 1990, IEEE Computer Society Press.

Lawler, D.G., and Williams, L.J.F. [1988] *MMU FDIR Automation Task --- Final Report*, Contract NAS9-17650, Task Order EC87044, McDonnell Douglas Astronautics Company - Engineering Services, 16055 Space Center Blvd., Houston, TX 77062.

McGuire, J. [1990] "Uncovering redundancy and rule-inconsistency in knowledge bases via deduction," *Proc. of Fifth Annual Conference on Computer Assurance: Systems Integrity and Software Safety (IEEE COMPASS-90)*, June, 1990, IEEE Computer Society Press.

McGuire, J., and Stiles, R. [1990] "Detecting interference in knowledge-based systems," *Proc. of the 5th Knowledge-Based Software Assistant (KBSA) Conference*, Rome Laboratory, Griffiss Air Force Base, New York 13441-5700.

Stachowitz, R.A., and Combs, J.B. [1987] "Validation of expert systems," *Proc. of the Twentieth Hawaii International Conference on System Sciences*, Kona, Hawaii, January,

1987.

Stachowitz, R.A., Combs, J.B., and Chang, C.L. [1987] "Validation of knowledge-based systems," *Proc. AIAA/NASA/USAF Symp. on Automation, Robotics and Advanced Computing*, Arlington, VA, Mar 1987; published also as: Chapter 7 in *Machine Intelligence for Aerospace Systems*, eds. H. Lum and E. Heer, *Progress Series in Astronautics and Aeronautics*, Vol. 101, ed. Martin Summerfield, Am. Institute of Astronautics and Aeronautics, 1989.

Stachowitz, R.A., and Chang, C.L. [1991] "Completeness checking of knowledge-based systems," *Proc. 24th Hawaii International Conference on System Sciences*, Kona, HI, Jan 1991.

SESSION 7 A

PRECEDING PAGE BLANK NOT FILMED

N92-16604

Design Concepts for Integrating the IMKA Technology with CLIPS

David Scarola

Carnegie Group, Inc

Abstract. This presentation will share our experiences in evaluating the technical alternatives for integrating the IMKA frame-based knowledge representation system with CLIPS. The Initiative for Managing Knowledge Assets (IMKA), consisting of Carnegie Group, Inc., Digital Equipment Corporation, Ford Motor Company, Texas Instruments Incorporated and U S WEST, Inc., was formed to foster the cooperative funding and development of a software technology that will meet each company's and their clients' needs for capturing and managing complex, corporate-wide knowledge. The IMKA Technology is a frame-based knowledge representation system for developing knowledge-based applications.

We've explored various models for integrating the IMKA technology with CLIPS. Integrating the IMKA technology with CLIPS allows application knowledge to be encoded naturally using both frames and rules, and allows the knowledge stored in frames to be reasoned about using rules. Integrating a frame-based system with a RETE-based rule system is a challenging task because the approach to accessing data is very different in each system. We found three integration models that can be used to address the different data access methods of IMKA and CLIPS. This presentation provides an overview of the features of the IMKA technology, describes the challenges of integrating the IMKA technology with CLIPS, and discusses the three integration models and the circumstances under which each is appropriate.

NOTE: I apologize for not having a paper prepared for the conference proceedings. I just ran out of time. However, my slides are included and they contain a fair amount of detail. If anyone would like more information on either the IMKA Technology or the integration models discussed in the slides, feel free to contact me at either **scarola@cgi.com** or **(412) 642-6900**.

PRECEDING PAGE BLANK NOT FILMED

Overview

1. History and background on IMKA.
2. The IMKA Technology.
3. The design goals for integrating the IMKA Technology and CLIPS.
4. The challenges involved in the integration.
5. The models that we've used.
6. The circumstances under which each model is appropriate.
7. Open issues and technical challenges.

History and Background

- IMKA: Initiative for Managing Knowledge Assests
- Participants:
  - U S WEST, Inc.
  - Digital Equipment Corporation
  - Ford Motor Company
  - Texas Instruments
  - Carnegie Group, Inc.
- Purpose: Foster the cooperative funding and development of a software technology to meet each company's needs for capturing and managing complex, corporate-wide knowledge.
- The IMKA Technology: A frame-based knowledge representation system.
- Industry participation: Includes Quintus Corporation and AICorp.
- IEEE Working Group P1252. For more information, contact Mitch Smith at (303) 541-6133

The IMKA Technology

The feature set:

- Basic entity is a frame.
- Frames contain attributes, relations, and messages.
- Class and instance frames.
- Class/subclass hierarchy.
- High degree of dynamicity in the frame base.
- Users can define their own relations.
- Slot and value inheritance.
- Users can define customized inheritance.
- Context mechanism.
- Demon mechanism.
- Meta-Knowledge.
- Path following.
- Persistent frame storage mechanism.
- RDB interface.
- Designed to be used with C++ & C.
- User-defined C++ types.

Design Goals

- IMKA allows powerful reasoning systems to be developed.
- IMKA plans to integrate with multiple rule languages.
- Integrating IMKA and CLIPS would provide a powerful hybrid system that contains both frames and rules.
  - Enhances CLIPS by allowing rules to match against frames.
  - Enhances IMKA by providing it with rule-based reasoning.
  - Allows the appropriate representation of knowledge.
  - Maintains the uniqueness and power of each system.
  - Supports the external data storage mechanisms that IMKA provides.
  - Is highly-integratable and portable.

Frame/Rete Data Access Models

Frame Data Access Model

- Setting a value in a frame is basically a passive operation.
- When a frame value is read, dynamic frame mechanisms are invoked to return the correct value.
- For subsequent reads, these mechanisms are re-invoked to return the correct value.
- Due to the high degree of dynamicity in the frame system.

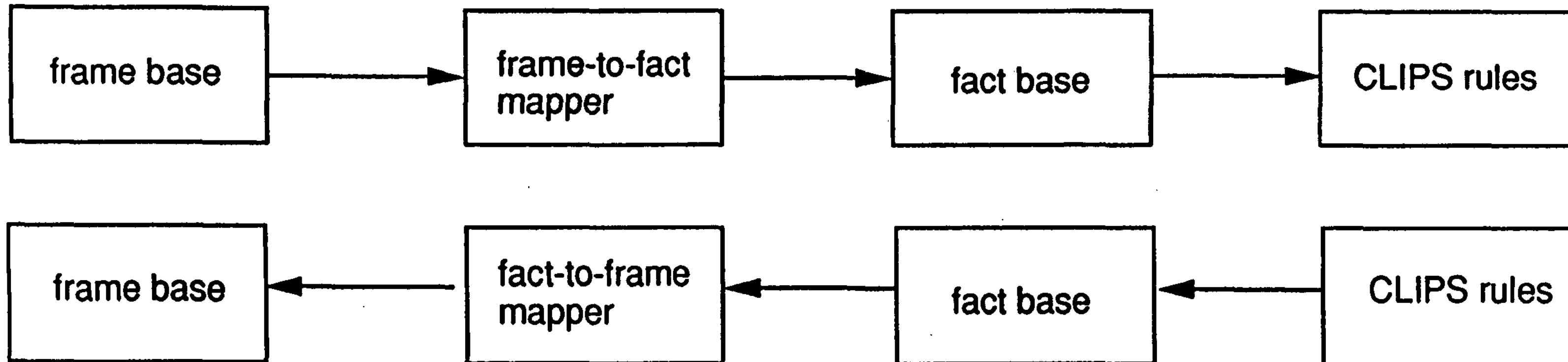
CLIPS (Rete) Data Access Model

- Reading a value from an object is a passive operation.
- When a value is written to an object, the modified object is propagated down the RETE network.
- The propagation requires that the correct values be available.
- Due to the caching mechanism for fast matches that RETE requires.

- Challenge: Matching rules against frames requires that the dynamic frame system features be invoked at write time so that the correct values can be matched by RETE.

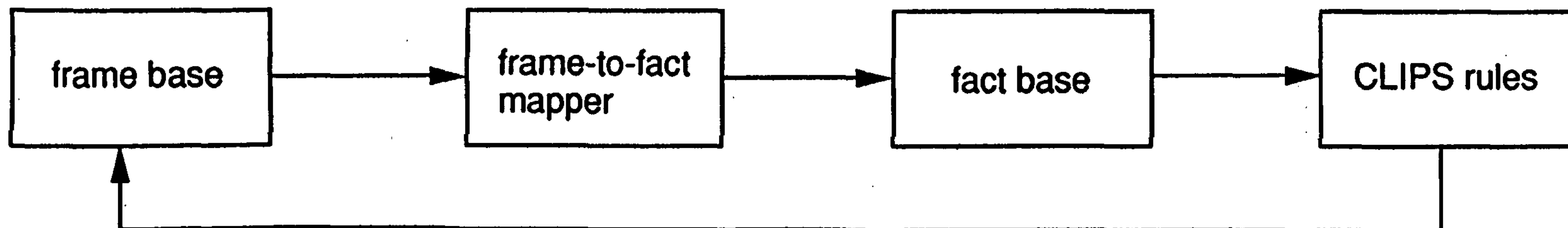
Three Models of Integration

Batch:



Direct:

Mixed:



Batch Model

- Approach:
  - Build an IMKA frame base.
  - Build CLIPS rules to match against facts.
  - Run frame base until it is ready for rule-based reasoning.
  - Execute a function which maps the frame base into the fact base.
  - Run the rules to match against the resulting fact base.
  - Modify facts on the RHS of the rules.
  - Execute a function which maps fact base into frame base.
- Matching dynamic frame features in RETE: The dynamic features of the frame system are invoked at map time and cached in the fact base.
- Limitations: None -- allows all dynamic frame features to be matched by CLIPS rules

Direct Model

- Differences from BATCH model:
  - No facts -- frames are matched directly by CLIPS.
  - Interface is on-going instead of a one-time mapping.
- Approach:
  - Build IMKA frame base.
  - Build CLIPS rules to match frames using Deftemplate syntax.
  - RETE network matches frames directly instead of matching facts.
  - When frames are updated to contain new value, RETE must be informed of which frames are updated and need to be re-matched.
- Matching dynamic frame features in RETE: An interface must be supported by which the frame system informs the RETE network which frames need to be repropagated.
- Limitations: Certain dynamic frame features cause other frames to be updated which can't always be identified.

Mixed Model

- A combination of BATCH & DIRECT:
  - Like BATCH, CLIPS matches facts instead of matching frames.
  - Like DIRECT, the interface is on-going instead of a one-time map.
- Approach:
  - Build and IMKA frame base.
  - Build CLIPS rules to match against facts.
  - Each frame is assigned a corresponding fact (internally). Whenever the user modifies a frame, the corresponding fact is also updated.
  - User invokes CLIPS to match the facts which correspond to frames.
  - Modify frames on the RHS of the rules.
- Matching dynamic frame features in RETE: When a frame is modified, its corresponding fact and all other facts which correspond to frames which are effected by that modification must be updated.
- Limitations: Certain dynamic frame features cause other frames to be updated which can't always be identified.

Appropriateness of Each Model

Batch:

- Easiest model to Implement.
- No modifications to the internals of either system.
- Rules are run a single time.
- All dynamic frame mechanisms available for matching.

Direct:

- Hardest model to implement
- Modification to rule language and frame language internals.
- Memory is a premium.

Mixed:

- Modification to frame system internals.
- Rules are run multiple times.

Open Issues and Technical Challenges

- Dynamic IMKA features which are difficult to integrate with CLIPS:
  - User-defined inheritance
  - Demons
  - Contexts
  - Dynamic classes and slots
- Limitations in the CLIPS Deftemplate external interface.
- C++ name mangling and overload types.
- CLIPS object system 5.0 is not integrated with rules.
- Backward chaining.
- Common syntax between frames and rules.
- Non-RETE inferencing system.

For additional information contact Dave Scarola at scarola@cgi.com or call (412) 642-6900.

A CLIPS-BASED TOOL FOR AIRCRAFT PILOT-VEHICLE INTERFACE DESIGN*

N92-16605

Thomas D. Fowler
California State Polytechnic University, San Luis Obispo

Steven P. Rogers, Ph.D.
Anacapa Sciences, Inc., Santa Barbara, California

Abstract

The Pilot-Vehicle Interface of modern aircraft is the cognitive, sensory, and psychomotor link between the pilot, the avionics modules, and all other systems on board the aircraft. To assist Pilot-Vehicle Interface designers a CLIPS-Based tool has been developed that allows design information to be stored in a table that can be modified by rules representing design knowledge. Developed for the Apple Macintosh®, the tool allows users without any CLIPS programming experience to form simple rules using a point-and-click interface.

INTRODUCTION

Development of the Pilot-Vehicle Interface (PVI) is one of the most challenging aspects of advanced aircraft programs. The PVI is the cognitive, sensory, and psychomotor link between the pilot, the avionics modules, and all other systems on board the aircraft. To reduce the pilot's workload while maximizing his situation awareness, the PVI must optimize the flow of information between the pilot and the hardware/software systems, respond smoothly to his immediate needs, and present the required information elements using the most readily comprehensible methods.

It is tempting to believe that advancing technology must have made the PVI of modern aircraft better than ever before. In fact, according to many pilots, the opposite is true. Although modern military aircraft have better weapons, better sensors, better computers, and better displays than in years past, the barrage of information from all the cockpit systems is nearly overwhelming, the data from various systems are not integrated to support the pilot's tasks, and the display types and formats are often ill-chosen. To acquire the information necessary for a given task, the pilot may have to extract data from several different systems and mentally translate or manipulate the data, while simultaneously dealing with the unrelenting workload imposed by aviation, navigation, and communication tasks.

The general requirements for an intelligent PVI for an advanced aircraft are that it be capable of filtering out information that is not currently useful, prioritizing the most useful information, prompting the pilot to examine the most important information, and integrating the functionally related data into an easily comprehended presentation. In order to determine the best presentation, the PVI must rapidly perform a layered series of

*This work is supported by an Army Small Business Innovation Research Contract (No. NAS213391) administered by the Aeroflightdynamics Directorate, NASA-Ames Research Center, Moffet Field, California. Dr. Edward M. Huff is the Contracting Officer's Technical Representative.

information presentation choices to select the best display modality, location, and format for the pilot's current activity.

In order to meet these general requirements for the PVI, it is necessary to base the system design on the pilot's functions and information requirements, rather than focusing on the technology of the aircraft systems. Taking such an information requirements approach, however, requires that thorough, systematic analyses be performed to identify the specific information elements necessary to perform each pilot function and to determine the various attributes of these elements. Next a set of rules must be developed to select the best display modalities, locations, and formats for integrating and presenting the information given the specific information elements and their attributes.

The necessity of a database of information elements, and rules linking the information elements to optimal presentation methods, strongly suggested that an expert system be developed to manage and guide the display method selection process. Such an expert system must be designed to organize the extensive collections of analytical information, document the controlled growth of the rules for information display, and manage the application of the rules to the database for selection of display methods in a methodical, consistent, and easily understood manner.

To assist in the development of such an expert system, a CLIPS-Based tool has been developed that allows the information elements and their attributes to be stored in a table that can be modified by rules representing design knowledge. The tool allows the user to create rules incrementally using a simple point-and-click interface. The following sections describe the tool from the user's perspective and discuss the use of CLIPS (NASA 1991) on the Macintosh for its implementation.

USER LEVEL VIEW

The tool presents the user with a table contained in a window that has horizontal and vertical scrolling capabilities. Rows and columns can be manipulated using cut, copy, and paste commands available from the menu bar. Entries in the table can be selected by clicking on them with the mouse or by using the arrow keys on the keyboard.

PVI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red			
Flight_Plan	Altitudes	High	Green			
Weather	Wind_Speed	Medium	Blue			
Weather	Visibility	High	Blue			
Aircraft	Fuel_Remaining	High	Red			
Aircraft	Oil_Pressure	Low	Red			

Figure 1. Information Element Table

The first row in the table is used to identify the type, name, and attributes of each information element. Figure 1 shows a table consisting of six information elements each having the attributes X, Y, Modality, Location, and Format.

Columns entries that are set by the user are tagged as "input" columns, while those set as the result of rule execution are designated "output" columns. In figure 1, columns X and Y are input columns with entries that the user is responsible for setting. The Modality, Location, and Format columns are output columns. Associated with each column is a list of value options from which each entry is selected. Figure 2 shows the input of a list of value options for column X and its designation as an input column.

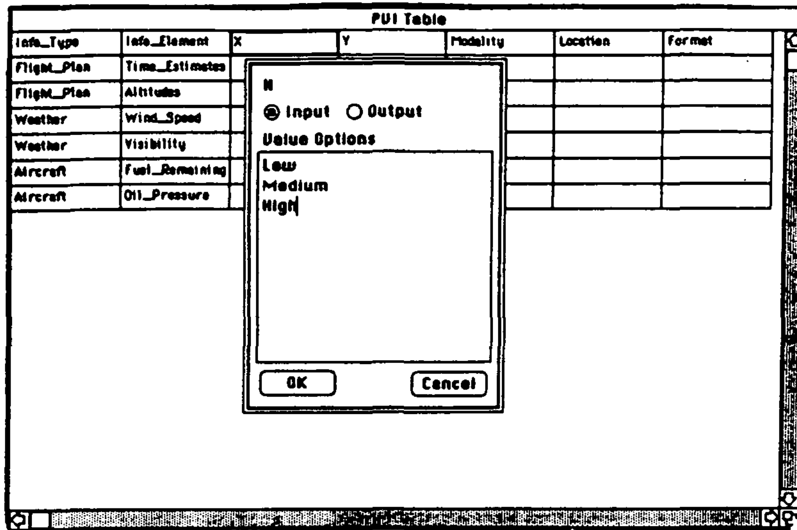


Figure 2. Column settings dialog

Once the list of value options has been set for a column, the dialog shown in Figure 3 is used to set the values of its entries. In this case, the value of the attribute X for the Wind_Speed information element is being set to Medium. The information type and attribute for the entry is indicated at the top of the dialog. If an entry already contains a value, then it is displayed in the box below the type and attribute indicators.

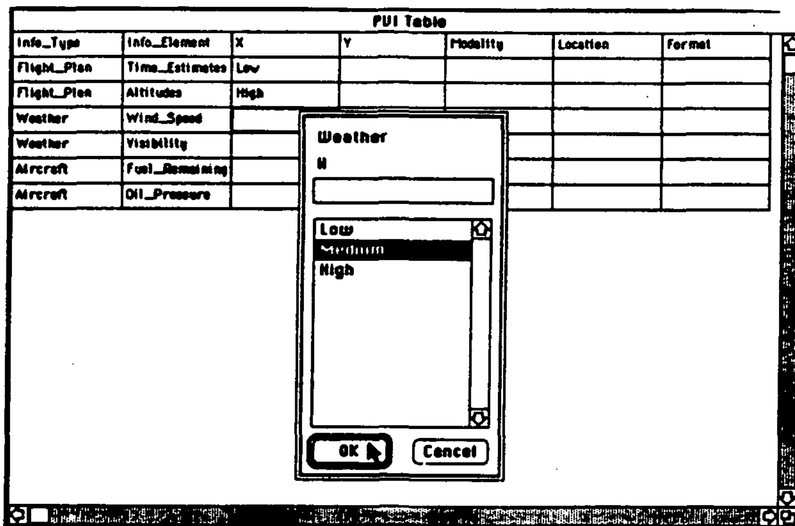


Figure 3. Attribute entry dialog

A list of value options must also specified for the Modality, Location, and Format columns for use by the rules which set them. Restricting the values of entries in this manner guarantees correctness of input and insures the consistent usage of symbols by user-created rules.

ORIGINAL PAGE IS
OF POOR QUALITY

Adding Rules To The Table

Rules can be formed that have left-hand side (LHS) patterns that match values in input and output columns, and right-hand side (RHS) clauses that set values in output columns. Rule creation is facilitated by the dialog shown in Figure 4(a). Here, a rule with name "Rule1" has been formed that will apply to information elements that have attribute X set to Low and attribute Y set to Red. The consequence of Rule1's execution is to set the attributes Modality, Location, and Format to Visual, Display1, and Iconic, respectively. Figures 4(b) and 4(c) show the formation of additional rules.

Figure 4(a). Rule1

Figure 4(b). Rule2

ORIGINAL PAGE IS
OF POOR QUALITY

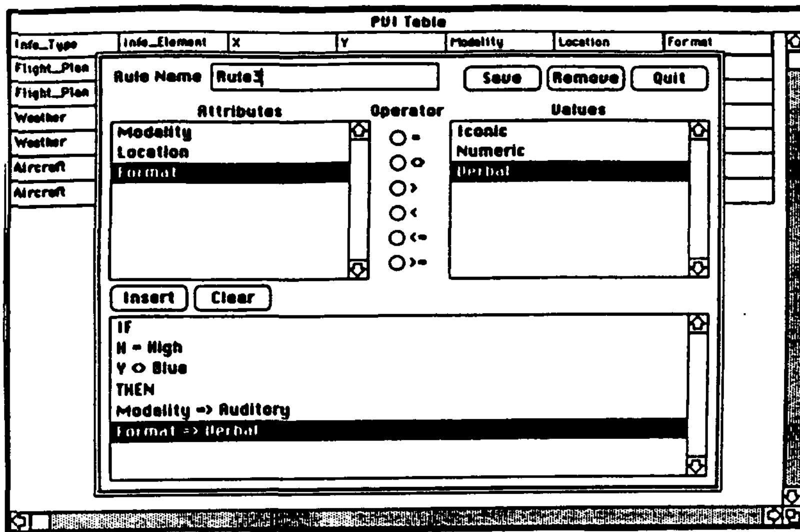


Figure 4(c). Rule3

Figures 4(a)-(c). Rule input dialog

LHS rule patterns are composed of an attribute, an operator, and a value. The list of attributes, displayed in the Attributes list-box, consists of the names of all input and output columns from the first row of the table. Displayed in the Values list-box is the list of value options that correspond to the selected attribute. LHS side patterns are formed by selecting an attribute, an operator (via the Operator radio buttons), and a value. The resulting pattern is displayed at the selected line in the rule display.

RHS rule clauses are composed of an attribute, the assertion operator (=>) and a value. When a RHS clause is being formed, only the names of output columns appear in the Attributes list-box and the Operator radio buttons are not applicable.

New LHS patterns and RHS clauses can be inserted into the rule by pressing the Insert button and deleted by pressing the Clear button. The Save button allows newly created rules to be saved while the Remove button deletes pre-existing ones.

Rule Execution

A single rule can fire multiple times to set attribute values for different information elements. However, rules are applied to the table in a row-wise manner. That is, the LHS and RHS parts of rule can only apply to one row at a time. Figure 5(a) shows the state of table after Rule1 [Figure 4(a)] has executed. Rule1's LHS patterns are satisfied by the values of attributes X and Y for information elements Time_Estimates and Oil_Pressure. Rule1 will fire once for each of these information elements, resulting in their Modality, Location, and Format attributes being set to Visual, Display1, and Iconic, respectively. Rule2 [Figure 4(b)] includes a LHS pattern that matches if "X >= Medium." This is permissible because an order relationship is established on the list of value options. Thus, the relationship (Low < Medium < High) holds for the value options for the attribute X.

Figure 5(b) shows the state of the table after the Rule2 and Rule3 have executed. Notice that because Rule3's RHS does not set the Location attribute, it has not been set for the information elements Altitudes and Fuel_Remaining.

ORIGINAL PAGE IS
OF POOR QUALITY

PUI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red	Visual	Display1	Iconic
Flight_Plan	Altitudes	High	Green			
Weather	Wind_Speed	Medium	Blue			
Weather	Visibility	High	Blue			
Aircraft	Fuel_Remaining	High	Red			
Aircraft	Oil_Pressure	Low	Red	Visual	Display1	Iconic

Figure 5(a). Table after Rule1 executes

PUI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red	Visual	Display1	Iconic
Flight_Plan	Altitudes	High	Green	Auditory		Verbal
Weather	Wind_Speed	Medium	Blue	Visual	Display2	Numeric
Weather	Visibility	High	Blue	Visual	Display2	Numeric
Aircraft	Fuel_Remaining	High	Red	Auditory		Verbal
Aircraft	Oil_Pressure	Low	Red	Visual	Display1	Iconic

Figure 5(b). Table after Rule2 executes

Figures 5(a) and (b). Effects of rule execution

Once a rule has been entered it will apply to any new information elements that are added to the table. Figure 6 shows the addition of the information element Planned_Speeds in the third row of the table and the setting of its Y attribute to Red. After the Y attribute has been set to Red, Rule3 [Figure 4(c)] will execute and the Modality, and Format attributes will be set accordingly (Figure 7).

The effects of modifying an existing rule are immediately reflected in the state of the table. Figure 8 shows Rule2 modified so that it sets the attribute Location to Display3 instead of Display2. After the modified Rule2 has executed, the Location attribute for information elements Wind_Speed and Visibility will be set to Display3 (Figure 9).

ORIGINAL PAGE IS
OF POOR QUALITY

PVI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red	Visual	Display1	Iconic
Flight_Plan	Planned_Speeds	High				
Flight_Plan	Altitude	High				Verbal
Weather	Wind_Speed	Medium			Display2	Numeric
Weather	Visibility	High			Display2	Numeric
Aircraft	Fuel_Remaining	High				Verbal
Aircraft	Oil_Pressure	Low			Display1	Iconic

Flight_Plan

Y

Red
Green
Blue

OK Cancel

Figure 6. Addition of new information element

PVI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red	Visual	Display1	Iconic
Flight_Plan	Planned_Speeds	High	Red	Auditory		Verbal
Flight_Plan	Altitude	High	Green	Auditory		Verbal
Weather	Wind_Speed	Medium	Blue	Visual	Display2	Numeric
Weather	Visibility	High	Blue	Visual	Display2	Numeric
Aircraft	Fuel_Remaining	High	Red	Auditory		Verbal
Aircraft	Oil_Pressure	Low	Red	Visual	Display1	Iconic

Figure 7. Table after addition of a new information element

Conflicting Rules

A rule conflict is defined as two or more rules attempting to set the same table entry. A conflict situation arises when the PVI designer has entered two or more rules that apply to the same output attribute(s) of an information element. A more subtle conflict situation presents itself when a change in the value of a table entry brings two or more previously non-conflicting rules into conflict.

The tool allows conflicting rules to be entered. However, when multiple rules attempt to set the value of the same table entry, the symbol "CONFLICT" is placed into the entry. The user can then request that a list of all rules that caused the conflict be generated and modify the appropriate rule(s) to resolve the conflict. The resolution of conflicting rules is an important feature of the tool that insures the consistency of the design knowledge.

ORIGINAL PAGE IS
OF POOR QUALITY

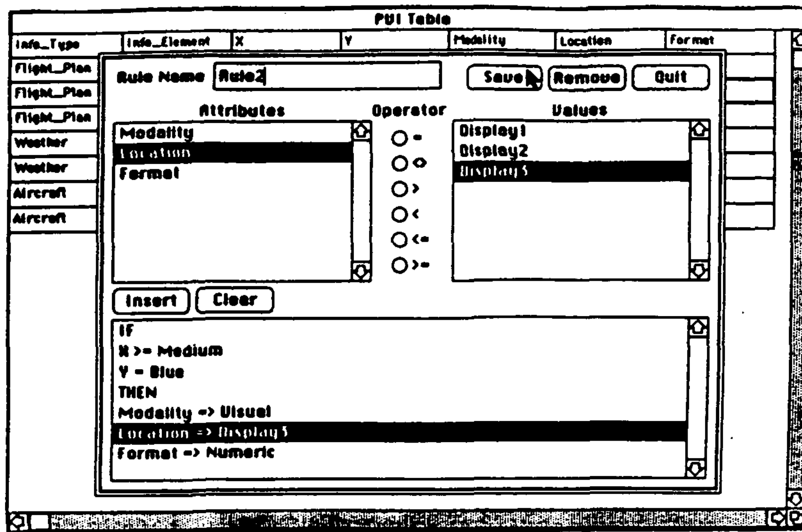


Figure 8. Modification of Rule2

PUI Table

Info_Type	Info_Element	X	Y	Modality	Location	Format
Flight_Plan	Time_Estimates	Low	Red	Visual	Display1	Iconic
Flight_Plan	Planned_Speeds	High	Red	Auditory		Verbal
Flight_Plan	Altitudes	High	Green	Auditory		Verbal
Weather	Wind_Speed	Medium	Blue	Visual	Display3	Numeric
Weather	Visibility	High	Blue	Visual	Display3	Numeric
Aircraft	Fuel_Remaining	High	Red	Auditory		Verbal
Aircraft	Oil_Pressure	Low	Red	Visual	Display1	Iconic

Figure 9. Table after modified Rule2 has executed

Column and Value Option Deletion

The deletion of a column or one of its value options is problematic in that rules that reference the column or value will no longer apply. For example, if the column representing attribute X were deleted from the table, then any rules that reference X will no longer fire. Similarly, if a value option for a column is deleted, any rules referencing that value will no longer execute.

There are several alternatives for handling this situation. One is to remove all rules that make reference to deleted columns or values. This has the dangerous consequence that a large number rules could potentially be removed due to a single operation by the user. A second alternative is to modify the rule set by removing all clauses that make reference to the deleted column or value option. While less severe than the first alternative, a large scale modification of the rule set could occur. A third alternative is to simply alert the user that a number of rules have been invalidated. A query can then be performed over the rule set to locate all invalidated rules. The user then has the option of modifying or deleting these rules as appropriate.

Since Invalidated rules are no longer applicable they are "unfired" by clearing all table entries that were set as a consequence of their execution. When a value option is

ORIGINAL PAGE IS
OF POOR QUALITY

C-3

removed from a column, all entries in the column with that value are cleared. The clearing of table entries in this manner can result in the LHS patterns of previously executed rules to longer be satisfied. These rules must also be "unfired" to maintain the consistency of the information in the table. The resolution of conflicts, as described above, has not yet been fully implemented. Currently, The truth-maintenance facility of CLIPS Version 5.0 is being considered for use.

IMPLEMENTATION

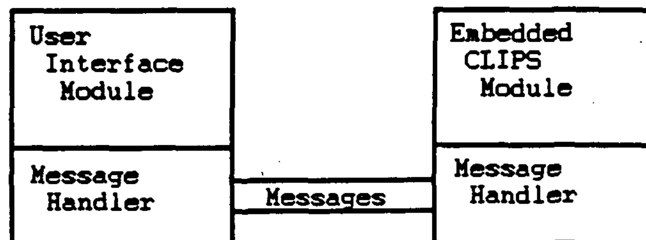


Figure 10. Tool Architecture

Architecture

The tool is comprised of three main components; a user-interface module, a message handling facility and an embedded CLIPS module (Figure 10). The user interface module is responsible for handling all events generated by the user and operating system. The embedded CLIPS component is responsible for maintaining the table and rule-base. The message handling component facilitates communication between the user-interface and CLIPS modules. Whenever the table is modified by the user, a message is sent to CLIPS by the user-interface. CLIPS handles this message by updating its fact-list and executing any rules that have become activated. If rules that result in the modification of the table are executed, then the CLIPS module sends a message to the user-interface module - which responds by updating the display accordingly. When the user creates rules, the user-interface module sends a message to CLIPS, which responds by adding the rule to its environment using the CLIPS Build function.

Developing With CLIPS On The Macintosh

Programming the Macintosh family of computers requires a substantial investment of time to become familiar with the toolbox routines implemented in ROM. Indispensable to any serious development effort for the Macintosh is the *Inside Macintosh* series (Apple 1985). Knowledge of the basics of Pascal records, procedures and functions is helpful because the toolbox interface is defined in Pascal. Apple's *Programmer's Introduction to the Macintosh Family* (Apple 1988) is a good source of information for those considering a developing for the Macintosh.

Symantec's THINK C compiler was chosen to implement this tool. Since CLIPS for the Macintosh was developed using THINK C, the task of embedding it into an application is made easier by using THINK C. In addition, THINK C comes with a class library that provides an interface to the Macintosh toolbox that can release the programmer from being concerned with many low-level toolbox details. Because Symantec's documentation for the THINK C Class Library (Symantec 1989) is extremely

terse and includes few coding examples, Dave Mark's *Macintosh C Programming Primer* series (Mark and Reed 1989, Mark 1990) is strongly recommended.

CONCLUSIONS AND FUTURE DIRECTIONS

The problem of transferring human knowledge into an expert system is known as the knowledge acquisition bottleneck (Giarratano 1989) . This tool helps reduce this bottleneck by facilitating the incremental acquisition of design knowledge from its users. It accomplishes this by allowing users without formal programming experience to easily create rules while assessing their effects. Using CLIPS to represent these rules provides a measure of power that is not attainable using other tools such as a spreadsheet. CLIPS' use of the Rete Algorithm (Forgy 1979) results in the execution of only those rules affected by incremental changes to the table. The pattern matching capabilities of CLIPS allows rules to be formed that reference table entries via user-defined symbols rather than positional row/column addresses imposed by the implementation of the table. This allows rows and columns to be added or rearranged without having to modify any rules.

Future directions for the tool include allowing the formation of more sophisticated rules, incorporating meta-rules that can be used to evaluate and examine the design rules entered by the user, and providing a rule explanation facility.

REFERENCES

Apple Computer, Inc. *Inside Macintosh.* 6 vols. Addison-Wesley, 1985 - 1991.

Apple Computer, Inc. *Programmer's Introduction to the Macintosh Family.* Addison-Wesley, 1988.

Forgy, Charles L. *On the Efficient Implementation of Production Systems.* Ph.D. Thesis, Carnegie-Mellon University, 1979.

Giarratano, Joseph, Gary Riley. *Expert Systems Principles and Programming.* Boston: PWS-KENT, 1989.

Mark, Dave, Cartwright Reed. *Macintosh C Programming Primer Volume I.* Addison-Wesley, 1989.

Mark, Dave. *Macintosh C Programming Primer Volume II.* Addison-Wesley, 1990.

NASA. *CLIPS Reference Manual (Version 5.0).* Software Technology Branch, Lyndon B. Johnson Space Center, 1991.

Symantec. *THINK C User's Manual.* Symantec, 1989.

N92-16606

ON THE GENERATION OF GRAPHICAL OBJECTS AND IMAGES FROM WITHIN CLIPS USING XVIEW

Terry Feagin

University of Houston - Clear Lake

Abstract. A variety of features that support the generation and manipulation of graphical objects and images in CLIPS are described. These features provide the CLIPS programmer with the ability to develop an enhanced user interface. Windows and objects within the windows (such as buttons, sliders, text, etc.) can be created, hidden, redisplayed, given new values, properties, or positions, and manipulated in various other ways. Menus can be generated for any window or object and displayed when desired. Interaction with the user is primarily via mouse movements and mouse button selections made over windows, items, or menus. User input is recorded in the form of fact assertions. Limited animation of objects is also supported.

INTRODUCTION

In the development of heavily interactive expert systems, it is important to provide a user interface that will enhance and accelerate the the overall process. Meaningful dialog should be accomplished with minimal effort. Clear and unambiguous communication that expedites the handling of sensitive or emergency situations and that provides intuitive mechanisms for giving commands and for receiving responses should be supported.

Computer graphics has always been recognized as a valuable aid for facilitating the flow of information between the user and the application. Modern computer graphics allows users to interact with their applications via two-dimensional color images that can move or be influenced by the user with a mouse, light pen, keyboard, joystick, or other graphic input device.

XView was developed by Sun Microsystems (both XView and Sun are trademarks) as a high level toolkit for developing graphic applications for the X Window System. It also facilitates the conversion of SunView applications to X Window applications because many of the features are similar. However, the display of graphic images in SunView is usually accomplished using "pixwins" and the conversion of these features to XView requires the direct use of the lower level Xlib routines. Also, the use of implicit dispatching (which allows for the concurrent input of text and mouse selections) in XView is more complicated than in SunView because of the subtle timing and synchronization issues of the X Window System. Nevertheless, the use of XView simplified considerably the development of the graphical extensions to CLIPS described below. In the current version, digitized color images or simple colored shapes or text may be displayed in the background of a window and the CLIPS programmer may elect to display buttons, sliders, text, or other items (with which the user may interact) on top of the background image. In addition, the user may interact with menus for an item or for the window itself if the CLIPS programmer has made provision for it.

The use of graphical input and output for an expert system conveys much more information to the user. The more traditional command-line interface is restricted to character input and output and is more susceptible to errors and misunderstandings. In a graphics-oriented interface, one can use a more natural vehicle for conveying ideas such as providing an image or icon that is easily recognized and indicating state changes via changes in the image. In serious or emergency situations, one could sound alarms and flash the images to gain the user's attention. Also, with the extensive use of menus and help windows, one can provide special explanations, diagrams, and assistance for novice users.

GRAPHICS PRIMITIVES

A number of primitive graphics commands are provided for the CLIPS programmer that facilitate the creation of windows, objects within windows, images within windows, etc. Most of these primitives are summarized below:

Command		Description
create-window	-	causes a new window to be created and its attributes to be specified
remove-window	-	causes an existing window to be destroyed
remove-all-windows	-	causes all existing windows to be destroyed
hide-window	-	causes an existing window to be hidden from view
show-window	-	causes an existing window to be displayed
open-window	-	causes an existing window to be opened or de-iconified
open-all-windows	-	causes all existing windows to be opened
close-window	-	causes an existing window to be closed to an icon
close-all-windows	-	causes all existing windows to be closed
set-window	-	allows various attributes of an existing window to be altered
get-window	-	retrieves the value of a given attribute of a given window
load-window-image	-	places a specified raster image into the window's background
draw-rectangle	-	causes a rectangle with prescribed attributes to be drawn in window
draw-circle	-	causes a circle with prescribed attributes to be drawn in window
draw-line	-	causes a line with prescribed attributes to be drawn in window
draw-polygon	-	causes a polygon with prescribed attributes to be drawn in window
create-item	-	causes a new item of specified type and attributes to be created
remove-item	-	causes an existing item to be destroyed
remove-all-items	-	causes all existing items within a window to be destroyed
remove-all-items-in-all-windows	-	destroys all items in all windows
hide-item	-	causes an existing item to be hidden from view
show-item	-	causes an existing item to be displayed (if the window is displayed)
set-item	-	allows various attributes of an existing item to be altered
get-item	-	retrieves the value of a given attribute of a given item
animate-item	-	allows an existing item to be moved within a window at a given rate
create-item-menu	-	causes a menu and selections to be created for a given item
remove-item-menu	-	destroys the menu for a given item
get-alert-window	-	displays an alert window and blocks user until he/she responds

In addition to these primitives, there are a number of functions and commands that permit one to reset the default values for various attributes (such as the size and position of a window). The use of defaults makes the system much easier to use. Also, there are commands that permit the user to list the items and the windows currently defined. This feature is especially useful for debugging purposes.

GRAPHICS INPUT

In addition to the graphics output primitives described above, there are a number of ways to provide graphics input to the system. The simplest mechanism is to select a button item within a window by depressing the mouse select button (usually the leftmost button on the mouse) while the mouse cursor is directly over the item. The result of this action by the user is the automatic assertion of a fact into the CLIPS database of facts. In this case the fact asserted would be:

(selected-item "item-name")

If the item were a text item with space where the user could enter text (e.g. NAME: __________), then when the user selected this item and typed in the name "username", the fact asserted would be:

(selected-item-text "item-name" "username")

The adjustment or selection of other items results in similar facts being asserted. The net effect is that, if the CLIPS programmer has provided rules that activate and fire when a particular item is selected or adjusted, appropriate actions can be taken that will allow the user to interact with the executing CLIPS program.

Another mechanism for interacting with the user involves the use of menus. The CLIPS programmer creates the menu using the create-window-menu or the create-item-menu commands described above. If the user depresses the menu-select button (usually the rightmost button on the mouse), while the mouse cursor is over an item (for which a menu has been created earlier) then the menu is displayed. If the user releases the menu-select-button while the cursor is over a particular menu selection, then a fact is asserted according to the selection made. For example, if at some point in time a menu were created via the command:

(create-item-menu "item-name" "load" "save" "quit")

then the item "item-name" would have a menu associated with it. Later, if the user depressed the menu-select button while the cursor was over the item, the menu:

load
save
quit

would be displayed and if the user selected the first selection, the following fact would be asserted:

(selected-item-menu "item-name" "load")

which could then be used to cause further actions to take place.

It is also possible to create menus that are not associated with any particular item, but that are associated with a window (or its background). If the user depresses the menu-select button while the cursor is over a window, but not over an item, then the menu associated with the window (if such a menu has been created earlier) would be displayed.

THE ISSUE OF CONTROL

In developing a graphics extension to CLIPS, one invariably faces the question of control. A graphics interface requires that events be accepted, queued up somehow, and handled appropriately in a timely fashion. Systems like SunView and XView generally require that the programmer define all the graphics entities (windows, items, menus, etc.), their interrelationships, any events that may interest the programmer, and callback routines which should be invoked whenever particular events occur, and *then* turn over the control of the program to a main loop. The reason for this is the desire to be able to deal effectively with multiple streams of input (mouse, keyboard, etc.) The execution of the program then proceeds according to the events that occur and the appropriate, related callback routines. In such a case, the programmer has given control of the program to the central main loop.

For the present extension of CLIPS, it was decided to avoid (to the greatest extent possible) changing CLIPS in any major fashion and to maintain the central control loop of CLIPS. Therefore, the dispatching mechanism of XView is called explicitly after each rule firing (to dispatch any events that might occur) and implicitly during any blocking or non-blocking read. This allows the user to obtain good response to graphics input events while CLIPS is firing rules and also when the user is entering commands directly to the CLIPS> prompt.

CONCLUSIONS

A graphics extension to CLIPS has been developed using the XView toolkit. The project is now complete and over fifty new, user-defined functions have been added to CLIPS. Most of the new functions and commands are graphics-oriented. Work is now underway to enhance these capabilities and examine their usefulness within various applications.

SESSION 7 B

PRECEDING PAGE BLANK NOT FILMED

N92-16607

PASSIVE ACQUISITION OF CLIPS RULES*

Vincent J. Kovarik Jr.

Software Productivity Solutions, Inc.
122 North 4th Avenue
Indialantic, FL 32903
(407) 984-3370

Abstract The automated acquisition of knowledge by machine has not lived up to expectations and knowledge engineering remains a human intensive task. Part of the reason for the lack of success is the difference in cognitive focus of the expert. The expert must shift his or her focus from the subject domain to that of the representation environment. In doing so, this cognitive shift introduces opportunity for errors and omissions. This paper presents work which observes the expert interact with a simulation of the domain. The system logs changes in the simulation objects and the expert's actions in response to those changes. This is followed by the application of inductive reasoning to move the domain-specific rules observed to general domain rules.

INTRODUCTION

The acquisition and application of complex domain knowledge remains a human intensive task. Efforts to directly elicit such knowledge from experts has not lived up to expectations and current knowledge acquisition tools fall short of direct acquisition from the expert. To a large degree, this is because tools focus on providing the knowledge engineer with powerful abstractions for building the declarative structures for the domain but do not encompass an understanding of the acquisition process. Consequently, human knowledge engineers must still be relied upon to develop a comprehensive knowledge base. Even those domain experts that learn to use knowledge engineering tools, their success is limited because of the necessity to shift their cognitive focus between the domain to the knowledge engineering environment. This shift in context results in errors and omissions.

The goal of this effort is to develop an approach to the direct extraction of knowledge from experts by emulating the unobtrusive observation and elicitation activities of a human knowledge engineer, i.e. a Passive Knowledge Acquisition System (PaKAS). This effort is not, as one might first assume, a knowledge engineering environment. PaKAS is intended to be a proactive system which "observes" the expert in the performance of a task, elicits rationale and background knowledge from the expert regarding the reasoning and decision behind the actions

* This work was supported by NASA, Johnson Space Center, under the Small Business Innovative Research (SBIR) program, contract NAS9-1129.

PRECEDING PAGE BLANK NOT FILMED

taken, and directly generates a knowledge base from this information. This generated knowledge base could then be modified and edited using an existing knowledge engineering toolset.

The first phase of PaKAS was developed in CLOVER, an object-oriented knowledge representation system developed by the author. CLOVER is a hybrid system, borrowing concepts from (Levesque 1984). It provides the usual capabilities of a traditional frame system in the areas of default reasoning, inheritance, attribute daemons. It also supports method definition, a message passing protocol, method daemons, and other programming constructs.

Acquisition by Observation

The "observation" of an expert by an automated system must make some assumptions and concessions regarding the actual mode of observation. Observation of expert behavior is not a totally new concept (Wilkins et al 1987) but the approach used by PaKAS is different. Of course, technology is not at a state where a machine can "visually" observe an expert and identify the actions performed by the individual, the objects acted upon, and the state changes resulting from the actions. Consequently, PaKAS works with a simulated environment. The expert works with the environment interactively, without any knowledge of the observation activity. The domain simulation logs each action performed by the expert and state changes resulting from the action.

Rather than forcing the expert to think both as an expert and as a knowledge engineer, the goal of PaKAS is to allow the expert to perform within their domain without external distraction. During their performance, the passive acquisition system watches, observes, and "take notes" regarding the actions performed by the expert. Then, as a human knowledge engineer does, after the expert has completed the task, the system queries the expert for rationale, goals, and anticipated results of their actions during the simulation execution. This allows the expert to focus only on the task at hand, thereby reducing the potential of omissions and oversights.

PaKAS Domain

Various options were discussed for the domain for the Phase I effort. It was decided that the Vacuum Vent Line / Intelligent Computer Aided Training tool (VVL/ICAT) would be used as the domain simulation. VVL/ICAT was developed to help train individuals in the concepts and troubleshooting of a vacuum system. It is constructed in Turbo C[1] and CLIPS.[2] Turbo C provides the basic user interface capabilities, along with the MetaWindows and Icon Tools support packages, and CLIPS provides the simulation capability. VVL/ICAT runs on an IBM PC or clone with a color graphics adapter and mouse.

After some experimentation with the VVL/ICAT tool, it became apparent that, although it was an excellent tool for building a vacuum vent line model and helping the user develop an understanding of the domain, it was not a simulation model which could be used as a front end for PaKAS. In order to propagate changes along the VVL model, the user needed to explicitly select a menu item and choose the proper option. For PaKAS, automatic propagation of changes whenever the user made some change in the model's state was necessary.

---

[1]Turbo C is a trademark of Borland International, Inc.

[2]CLIPS is a forward chaining inference engine developed by NASA, Johnson Space Center.

To support the type of model interaction desired for PaKAS, an object-oriented model was developed within the knowledge representation system This model provides model-based reasoning via discrete-event qualitative simulation. Since PaKAS would have had the model objects and a representation of the domain for acquisition purposes anyway, the only additional capabilities developed were the model execution and user interface. VVL/ICAT was used as the method for model construction and validation. PaKAS then imports models developed using VVL/ICAT and assumes that they have been checked for correctness by VVL/ICAT prior to being loaded into PaKAS.

Figure 1 shows the architecture of the Phase I PaKAS system. The VVL/ICAT system is used to construct and debug a vacuum vent line model and save that model to disk. PaKAS then reads and interprets the VVL/ICAT file and instantiates the corresponding knowledge classes representing the model.

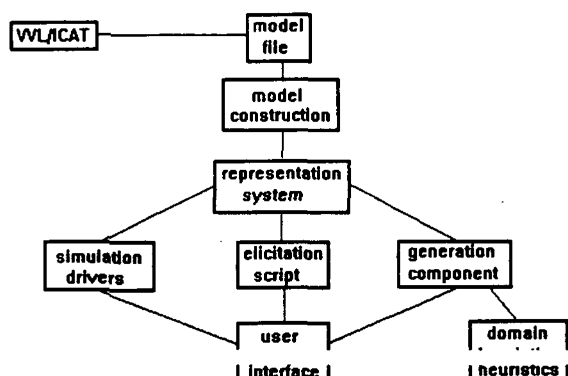


Figure 1. PaKAS Phase I Architecture

Domain Knowledge

Vacuum propagation begins by setting the state of the objects to a pre-vacuum condition. A vacuum is then propagated starting at the vacuum source(s) and continuing through the network of objects until a leaf node (i.e. a gauge or apparatus) or a closed valve is encountered. After the vacuum has been propagated, the leak is back-propagated to update objects affected by the leak condition. PaKAS uses a set of connection relations specified in the representation system to build a directed graph of the network topology from the vacuum sources outward. This represents the direction of vacuum propagation from a vacuum source.[3] Effectively, this collapses the connecting pipes into a single object between two or more model objects. The dependencies between the VVL simulation objects are shown in figure 2.

[3]Leaks are propagated in **both** directions along the network.

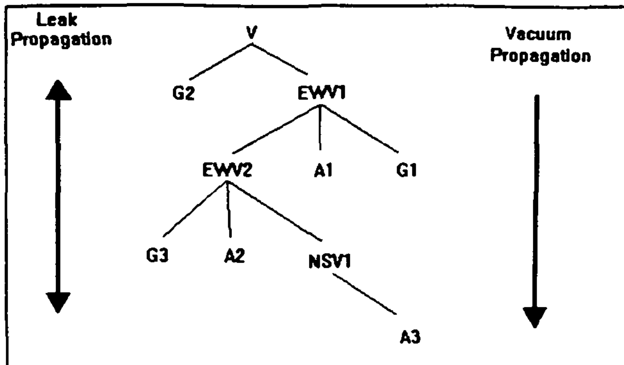


Figure 2. Model Dependencies

"V" is a vacuum source, "G*n*" represents some numbered gauge, "A*n*" represent some equipment or apparatus that uses the vacuum, and the "EWV*n*" and "NSV1" represent "east-west" and "north-south" valves, respectively.

The prototypical descriptions of the components are the description of the classes that make up the model. Each of the components is a descendent of the abstract class **model-component**. The model-component class defines basic methods and attributes common to all model objects. The lower-level classes (valves, gauges, apparatus, and vacuum) specialize this abstract class. The model-component object definition is shown below.

```
(defobject model-component physical-object
  :instantiable nil
  :attributes ((x-coord :cardinality 1 :default 0)
               (y-coord :cardinality 1 :default 0)
               (name :cardinality 1)
               (display-code :cardinality 1)
               (foreground-color :cardinality 1 :default 1)
               (background-color :cardinality 1 :default 0))
  :relations  ((connected-to :range model-component)))
```

The abstract class also defines basic simulation actions and value propagation routines for the simulation. These capabilities are defined as a collection of methods and daemons at the abstract class level. Examples of these are shown below.

```
;;;
;;;  After an instance of any model-component is created, it
is
;;;  asserted as a part of the current model instance.
;;;
(defmethod (model-component :after :new) ()
  (tell (pack* self '- (tell self :get instance-number))
        :assert part-of (tell model :get current-model)))

;;;
;;;   This method writes the current state of a model
component
```


ORIGINAL PAG
OF POOR QUA

```
;;;  to the specified output stream.  It is written as a
LISP
;;;  expression
;;;  so that when it is loaded back in, it will be evaluated
;;;  and set
;;;  the state of the object to the value specified.
;;;
(defmethod (model-component :write-state) (stream)
   (princ (pack* "$(tell " self " :set state "
                             (?attr self 'state) ")")
          stream)
   (terpri stream))
```

Coupled with the physical model is a representation of the events or actions that can be performed by the expert and the states that can be present for each of the equipment components. Event knowledge is an understanding of the potential actions that the expert may perform on the instantiated objects in the model, the effects of the action to the object acted upon, and the propagation of changes through the model along the relationships defined. In the VVL domain there are three actions that may be performed. Two affect the valve object, **open** and **close**. These actions change the state of the simulation and are used by the expert to identify the leak by isolating parts of the system. The third action is **inspecting** a gauge. This action does not affect the state of the underlying simulation but is used by the expert to guide the diagnosis process. This distinction between inspection actions and state-change actions on the part of the expert plays a key role in the induction of rules.

ACQUISITION PROCESS

The passive acquisition process is comprised of three phases,

1) Observation,
2) Playback/Elicitation, and
3) Rule Generation.

The next sections describe these phases in more detail.

Observation

In the observation phase, the system emulates a human knowledge-engineer. There are two sub-processes in this phase. The first is a domain analysis process. This process addresses the construction of domain knowledge for the qualitative simulation model. Contextual knowledge regarding the types and structures of the objects involved in the model, potential connectivity, relationships, and states are analyzed.

The domain analysis is performed as a precursor to observing the expert's actions during a simulation execution. In the example domain for this effort, the basic structural analysis of the model to be executed is constructed. General domain knowledge such as descriptions of valves, vacuum sources, etc. and their properties and behaviors have been "hand crafted" for this effort. Subsequent efforts could build on existing domain knowledge bases or previous efforts in

constructing models and, as PaKAS is repeatedly applied, the corpus of reusable world knowledge will grow correspondingly.

The second sub-process is the observation of the expert's diagnosis action's during the simulation. This involves specifying the set of initial states and starting the simulation. The simulation technique used for the phase I efforts is a qualitative, discreet-event simulation technique. It is constructed via a collection of methods and daemons triggered by state changes in the model components.

For example, as the user opens or closes a valve, the effect of the vacuum source is propagated throughout the system based on the state of the valves. As the vacuum is propagated, the gauges' state are reset to nominal. In a second pass, the effect of the leak is propagated throughout the network staring at the leak source. As the leak is propagated, gauges are set to abnormal for those gauges affected by the leak.

Procedural knowledge is then gathered by observing the expert in action. The system accomplishes this by initializing a log file containing the initial state of all objects within the model. PaKAS then captures each action performed by the expert and saves them in the log file as well. Actions are stored are a triplet consisting of the actions performed, the object on which the expert performed the action, and the state of the object. In the case of inspection actions, the state information is the state of the object inspected returned by the system. For state-change actions, it is the new state of the object as set by the expert. As the system collects the actions performed by the expert, it builds a temporal network of action instances. These form a procedural network which provides an initial ordering of the tests and actions performed by the expert.

Playback/Elicitation

The elicitation phase involves playing back the actions performed by the expert and querying for goals, rationale, anticipated changes, etc. This directly parallels the behavior of a human knowledge engineer. In this phase, commonsense knowledge would be advantageous. As described by (Lenat 1986), it is our common corpus of knowledge that allows us to master new information and situations. The human knowledge engineer may rely on a basic understanding of vacuums in order to ascertain the interdependencies between the various components of a model and provide insight into what the expert may have been thinking when performing a particular action.

The PaKAS elicitation process is built around the playback of the simulation run performed by the expert. Playback is performed through the interpretation of the log file entries for the simulation execution which is being analyzed. The playback follows the process used by a human knowledge engineer when eliciting knowledge from an expert. After watching the expert perform their task, the knowledge engineer will "walk through" the expert's actions.

During the playback of the simulation, PaKAS asks the expert for information regarding the actions performed. These questions can be thought of as addressing several areas. These include goal information, "Why was an action performed," strategy assessment, "When an action is performed," rationale, "What was the purpose of the action," anticipations of the expert, "What was the expected outcome (if any) of the action," and others.

Rule Generation

The first step is the development of instantiation-specific rules. These rules are collections of

inspection and action patterns which were performed for the specific model instantiation. These are represented as the literal pairs of inspections and actions. The system assumes that a series of state inspection actions followed by state-change actions constitutes a specific rule instance.

The rule developed is specific, however, not only to the model being investigated, but also specific to the state of the model at that point in time. Because of this, the model specific rules developed are limited in scope and not very useful as a general rule that could be applied to other model instances in the domain. They do, however, form a solid base which can be analyzed and provide a base to develop more general-purpose rules for the domain.

Induction is then applied to build general rules which would be applicable across multiple model instantiations within the same domain. The approach used for the induction process was to develop a set of generalized tests from the specific rules. These generalized tests were developed using an interactive, stepwise refinement. The initial approach utilized a four step process described below.

1) *Replace instance references with the class id and a variable name* - This step takes the specific instance inspections such as (CHECK GAUGE-1 ABNORMAL) and transforms it into (GAUGE ?X STATE ABNORMAL). The variable name X is associated with GAUGE-1 so that any further references to GAUGE-1 encountered will result in the use of X as the instance handle variable.

2) *Add equality test for the state observed* - This step inserts the boolean predicate "=" into the antecedent clause, yielding (GAUGE ?X STATE ABNORMAL). This example was adequately addressed using the "=" test. This will require expansion of the elicitation process in phase II to check for value ranges.

3) *Build relationship tests using both the inspected objects and the state-change objects* - This step adds antecedent clauses which relate the inspection objects and state-change objects. For example, (GAUGE ?X IS-CONNECTED-TO VALVE ?Y), is a generated relationship test for the fact that a gauge is connected to a valve. This step begins to refine the capabilities of the rule by defining configurations in which the rule is valid.

4) *Add equality test for the previous state of the object(s) acted upon by the user* - Finally, tests are added which test for the prior state of the objects acted upon by the user. For example, if the user opens VALVE ?Y, the test that would be added is (VALVE ?Y STATE CLOSED).

EXECUTION EXAMPLE

This section walks through an interaction with the PaKAS system. The PaKAS user interface is shown in figure 3. The simulation is started when the expert clicks on the "Simulation" menu choice at the top of the screen. At this point, a new simulation history file is opened for the model, a pipe section is chosen at random for failure,[4] the vacuum and leak effects are propagated through the network, all initial states are then logged to the simulation history file,

[4]Although devices as well as pipes can fail causing leaks, the phase I prototype was limited to pipes due to the funding and time constraints of a phase I SBIR.

PaKAS enters observation mode, and control is turned over to the expert.

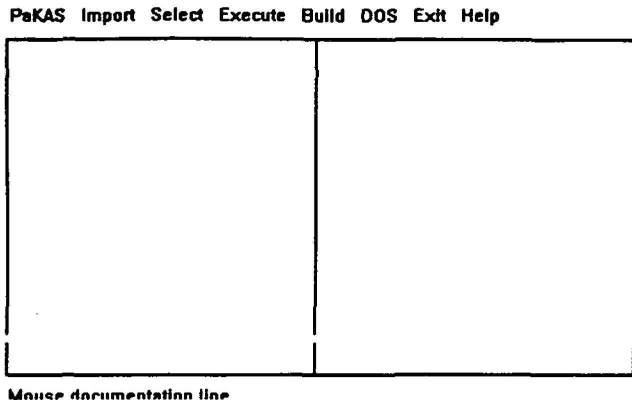


Figure 3. PaKAS User Interface

Once control is turned over to the expert, the expert then begins inspecting gauges and opening or closing valves until she believes the leak point has been identified. At this point, the simulation is completed by clicking again on the "Simulation" menu choice.

In our example, all valves are open at the start of the simulation. The expert clicks on gauges 1 and 3, finds both to be abnormal, then closes valve 2. Then gauge 1 is re-inspected and found to be nominal. At this point, the expert closes valve 3 and opens valve 2. Gauge 3 is re-inspected and found to be abnormal. The expert then click on "Simulation" to conclude this run having identified the leak to be in the pipe section connecting apparatus-2, valve-2, valve-3, and gauge-3.

At this point, PaKAS closes the simulation log file and returns to a command mode. An example of the simulation log file contents is shown in figure 4.

```
$(set-window 'history)
$(clear-screen)
$(center-line nil)
$(auto-crlf t)
Simulation run for: demo1
Date: 7/10/1990  --  16:25:26
Initializing model state...
$(tell GAUGE-1 :set state ABNORMAL)
$(tell GAUGE-2 :set state ABNORMAL)
$(tell GAUGE-2 :set state ABNORMAL)
$(tell EW-VALVE-1 :set state OPEN)
$(tell EW-VALVE-2 :set state OPEN)
$(tell NS-VALVE-1 :set state OPEN)
$(set-window 'model)
$(tell *current-model* :display)
$(set-window 'history)
Beginning elicitation playback...
$(CHECK GAUGE-1 ABNORMAL)
$(CHECK GAUGE-3 ABNORMAL)
$(CLOSE EW-VALVE-2)
$(CHECK GAUGE-1 NOMINAL)
$(CLOSE NS-VALVE-1)
```

```
$(OPEN EW-VALVE-2)
$(CHECK GAUGE-3 ABNORMAL)
Completion of elicitation playback...
```

Figure 4. Simulation Log File

In the elicitation phase, PaKAS opens the simulation log file and begins playing back the steps taken by the expert. The simulation log file is script of functions to be evaluated and text strings to be played back on the screen. The first few lines of the log file are initialization functions which are generated for all log files.

```
$(set-window 'history)
$(clear-screen)
$(center-line nil)
$(auto-crlf t)
```

These are followed by several text strings identifying the VVL/ICAT model instance for which the log file applies, the date and time of the simulation, and a descriptive message describing the initialization process.

```
Simulation run for: demo1
Date: 7/10/1990  --  16:25:26
Initializing model state...
```

The next few lines reset the state of all the objects within the model to their state at the start of the simulation.

```
$(tell GAUGE-1 :set state ABNORMAL)
$(tell GAUGE-2 :set state ABNORMAL)
$(tell GAUGE-2 :set state ABNORMAL)
$(tell EW-VALVE-1 :set state OPEN)
$(tell EW-VALVE-2 :set state OPEN)
$(tell NS-VALVE-1 :set state OPEN)
```

The system then redisplays the model on the screen and tells the expert that it is initiating the elicitation sequence.

```
$(set-window 'model)
$(tell *current-model* :display)
$(set-window 'history)
Beginning elicitation playback...
```

The remainder of the actions consist of the inspection or state change actions performed by the expert. The actions are logical broken into collections of inspection/action pairs. This forms a general equivalency with the production rule formalism.

```
$(CHECK GAUGE-1 ABNORMAL)
$(CHECK GAUGE-3 ABNORMAL)
$(CLOSE EW-VALVE-2)
$(CHECK GAUGE-1 NOMINAL)
```

```
$(CLOSE NS-VALVE-1)
$(OPEN EW-VALVE-2)
$(CHECK GAUGE-3 ABNORMAL)
Completion of elicitation playback...
```

The system restates the inspections and actions performed by the expert using template driven natural language. These are executed in sequence with the system eliciting responses from the expert at each set of inspection/action pairs. The expert is asked several basic questions regarding the intentions, goals, and anticipated outcome (if any) of the actions performed. An example of this interaction is shown in figure 5.

PaKAS> First you checked GAUGE-1 which was ABNORMAL.
PaKAS> Then you inspected GAUGE-3 which was ABNORMAL.
PaKAS> Next you CLOSED EW-VALVE-2.

PaKAS> Why did you CLOSE EW-VALVE-2?
Vince> The valve is roughly in the middle of the network.

PaKAS> What did you hope to find after closing EW-VALVE-2?
Vince> That GAUGE-1 had a NOMINAL reading.

PaKAS> Why would that be significant?
Vince> Because it would mean that the leak is after EW-VALVE-2.

Figure 5. Elicitation of Rationale

The generation phase then takes the collection of inspection/action pairs and develops model-specific rules. These rules are a literal conversion of the expert's actions into a model-specific production rule. Figure 6 shows an example of model-specific production rule instantiations..

```
RULE-1
     If
          (GAUGE-1 STATE EQ ABNORMAL)
          (GAUGE-3 STATE EQ ABNORMAL)
     Then
          (CLOSE EW-VALVE-2)

RULE-2
     If
          (GAUGE-1 STATE EQ NOMINAL)
     Then
          (CLOSE NS-VALVE-1)
          (OPEN EW-VALVE-2)

RULE-3
     If
          (GAUGE-3 STATE EQ ABNORMAL)
     Then
          (DONE)
```

Figure 6. Model-specific Production Rules

In viewing the model-specific rules it immediately becomes evident that, although a literal translation of expert actions into production rules for a specific model is certainly feasible, it is of little value. Firstly, the rules would be valid only for that particular model instantiation. This limitation alone would be enough to require further work. A second reason is the implicit reasoning and assumptions of the expert that are not evident by their actions alone.

An example can be found in figure 9 between RULE-1 and RULE-2. In the first rule, the expert checks two gauges and closes a single valve. In looking at the dependency relationship between the gauges and valve, we see that the valve is in between the two gauges on the dependency chain. When RULE-2 is initiated, the only inspection action is performed on GAUGE-1. Implicitly, what the expert has done is to bisect the dependency graph at EW-VALVE-2 by closing it. By then inspecting GAUGE-1, the expert immediately knows if the leak is before or after EW-VALVE-2.

Since GAUGE-1 had a NOMINAL reading, the leak is somewhere after EW-VALVE-2. So the expert has implicitly "marked" EW-VALVE-2 as the furthest known node in the network with no leak. This means that the leak is further along the dependency tree. Since there are two more pipe groups, the expert opens EW-VALVE-2 to allow the vacuum to propagate further down the dependency tree and closes NS-VALVE-1 to bisect the remaining pipe network.

At this point all that needs to be inspected is GAUGE-3. If it is NOMINAL then the leak is between NS-VALVE-1 and APPARATUS-3. Otherwise, it is in the pipes connecting EW-VALVE-2, NS-VALVE-1, GAUGE-3, and APPARATUS-2. The gauge does read nominal and the expert concludes the interaction.

Figure 7 below shows the CLIPS rules generated after applying the induction strategy to the instance rules.

```
(DEFRULE RULE-3
;;;  It was on the pipe network
       (GAUGE ?V1 ABNORMAL)
```

```
        (VALVE ?V2 OPEN)
        (?V2 CONNECTED-TO ?V1)
=>
        (ASSERT (VALVE ?V2 CLOSED))
;;; It bisected the network
)  ;;; END DEFRULE

(DEFRULE RULE-6
;;; It was between the closed valve and the vacuum source
        (GAUGE ?V4 NOMINAL)
        (VALVE ?V5 OPEN)
        (VALVE ?V2 CLOSED)
        (?V2 CONNECTED-FROM ?V4)
        (?V5 CONNECTED-FROM ?V4)
=>
        (ASSERT (VALVE ?V5 CLOSED))
        (ASSERT (VALVE ?V2 OPEN))
;;; to propagate the vacuum further
;;; to bisect the remaining section of pipe
)  ;;; END DEFRULE

(DEFRULE RULE-7
;;; to finalize the procedure
        (GAUGE ?V1 ABNORMAL)
=>
)  ;;; END DEFRULE
```

Figure 7. Generated CLIPS Rules

CONCLUSIONS

The use of a hybrid object-oriented knowledge representation system greatly simplified the representation and execution of both the domain model and the expert actions. Using a common representation allows the acquisition system to access domain information, object structures, expert events, and any related knowledge that may be entered into the system.

A common knowledge representation system for the individual components also allows the construction of corporate or commonsense knowledge that may be applied to subsequent domains. This capability eases the construction of new domains by being able to copy and modify structures from existing, related domains.

The representation of the domain within the knowledge representation system needs is being expanded in the phase II effort to encompass a wider range of world processes and events. The basic engine for representing the various inspection actions and state change actions should be extended to be similar to the event manager concept used in simulation languages and systems.

Finally, although CLIPS is the primary target, the promising results thus far makes this technology useful for other production languages and expert system shells. This aspect of the technology will be investigated as part of the commercialization effort of Phase III.

REFERENCES

Buchanan, Bruce G. *Some Approaches to Knowledge Acquisition.* Report No. KSL-85-38, Knowledge Systems Laboratory, Stanford University: Stanford, Calif., October, 1985.

Hayes-Roth, Frederick, and John McDermott. *Knowledge Acquisition from Structural Descriptions.* Report No. P-5910, Rand Corp.: Santa Monica, Calif., 1976.

Levesque, Hector J. "Foundations of a Functional Approach to Knowledge Representation." *Artificial Intelligence* **23**, Elsevier Science Publishers B.V.: New York, 1984. 155-212.

Lenat, Doug, Mayank Prakash, and Mary Shepherd. "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks." The AI Magazine, (volume, number, and date unknown). 65-85.

Wilkins, David C., William J. Clancey, and Bruce G. Buchanan. *Knowledge Base Refinement by Monitoring Abstract Control Knowledge.* Report No. KSL-87-01, Knowledge Systems Laboratory, Stanford University: Stanford, Calif., 1987.

N92-16608

YUCSA: A CLIPS EXPERT DATABASE SYSTEM TO MONITOR ACADEMIC PERFORMANCE

Anestis A. Toptsis, Frankie Ho[1], Milton Leindekar[2], Debra Low Foon, and Mike Carbonaro

Dept. of Computer Science and Mathematics
York University, Atkinson College
Toronto, Ontario M3J 1P3, Canada
Phone: (416) 736-5232
Fax: (416) 736-5773
Email: anestis@cs.yorku.ca

Abstract. This paper presents YUCSA (York University CLIPS Student Administrator), an expert database system implemented in CLIPS, for the monitoring of the academic performance of undergraduate students at York University[3]. The expert system component of the system has been already implemented for two major Departments and it is under testing and enhancement for more Departments. Also, more elaborate user interfaces are under development. We describe the design and implementation of the system and also discuss the problems encountered, as well as our immediate future plans. The system has excellent maintainability and it is very efficient (assessment of one student takes less than one minute).

---

[1] Also with Goodfellow Consultants, Inc., Toronto, Canada.
[2] Also with Royal Bank of Canada, Division of Management Information Systems, Toronto, Canada.
[3] York University is one of the largest in Canada, having a current enrollment of about 50,000 students.

INTRODUCTION

Student enrolment and degree requirement satisfaction checks are time consuming tasks for the University registration staff. Moreover, subjective judgements of the registration personnel may cause further conflicts in degree audits. On the other hand the rules posed for graduation and registration are numerous, frequently changing, and many of them quite complex. In the absence of an expert system, at least one human expert is necessary to assess a student's academic record for graduation and registration requirements satisfaction (as an example, a rule is "a student can only enroll in a fourth year course if he/she has completed at least 12 courses from which at least 4 in the third year level, and has obtained a grade of C or better in courses C1 and C2". Moreover, several prerequisites usually apply for any upper level course that a student wishes to enroll). As a result, a computerized academic performance monitoring package is always desirable for obvious reasons (speed, minimization of number of errors caused during the assessment process, ability for mass production of reports to be sent to students, etc).

Past experience has taught us that employment of "conventional" tools (such as operating system utilities, relational database management systems, and programming in a 3GL) is inadequate for the implementation of such a system (Gale 1990). The main drawback of such conventional tools is that the resulting system is very difficult to be maintained due to the constant changing of University curriculum, registration, and graduation rules. Therefore, even if all the rules are captured correctly during the system's development (which is not a typical case), it is very likely that the resulting product will be almost obsolete even by the time of its release. This is because some of the rules would have been already updated and/or deleted, while new rules would have been added. On the other hand, after a fairly short time development effort (less than one academic year, including problem setup, learning the details of the University rules, design, implementation, and limited testing) we have a *working* expert system which performs the above discussed task, *correctly*.

The system is designed to,

1. Replace manual effort and time spent in checking course enrolment pre-requisites and restrictions satisfaction;

2. Eliminate subjective judgements of the registration personnel;

3. Quickly determine the achievements of a student and give advice on the outstanding course works to complete his/her program.

The system is designed to perform various needs for different users. A registration personnel will use it to determine whether a student is allowed to enrol into a particular course or to determine whether a student fulfils all his/her requirements of the program. On the other hand, a student uses it to check the progress of his/her achievements and get advice from the system the outstanding course works he/she has to complete for graduation. Other staff may use it to update the student's record etc.

DESIGN AND IMPLEMENTATION

In order to understand the design concerns of the system, it is better to discuss the activities a student may face in his/her life in the University. Those activities are illustrated in Figure 1. Although the diagram is tailored for York University, it is generally true for any other institution if some rules has been changed to reflect different flow in the diagram. A new student registered into York will be evaluated for whether he/she has already completed some advanced standing courses from other institution, and if so, some advanced credits will be granted. Then, the student will take courses up to a maximum number of course that is allowable to take simultaneously. After the school term, he/she may pass or fail the courses. Credits will be given to the passing courses, and he/she will repeat those courses he/she fails. When a student fails too many courses or when he/she meets the debarment conditions, he/she will be debarred from York. Of course, a student has an option to repeat the same course once, in order to upgrade his/her marks even though he/she has passed that course. When a student accumulates enough credits and satisfies all degree requirements, he/she will graduate with a degree. At any

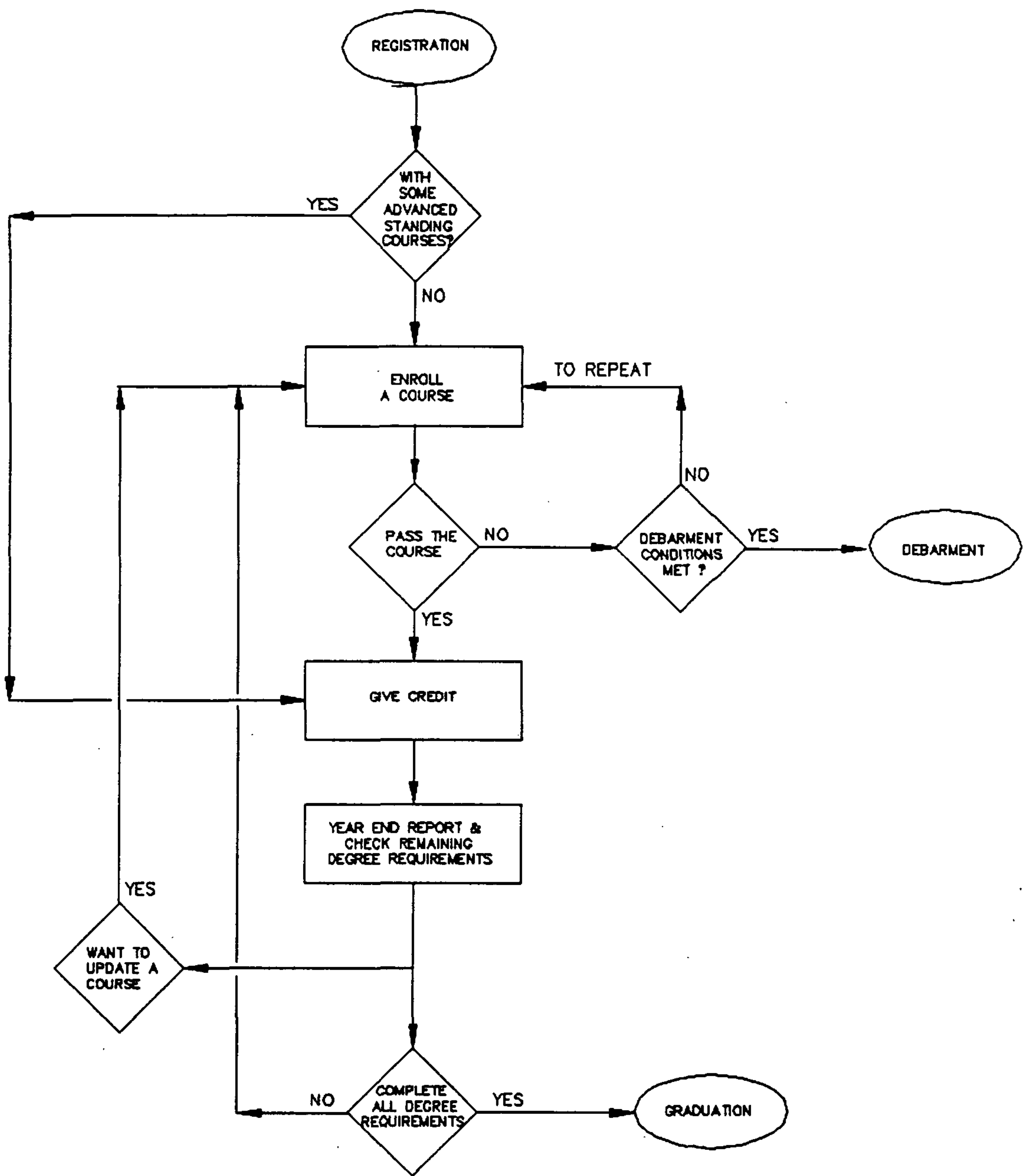


FIG 1: STUDENT LIFE CYCLE DIAGRAM

time, upon his/her discretion, a student can withdraw from the program which he/she is enroling.

The flow within the diagram is directed by the rule or a group of rules in a "box" of the diagram. Examples of those rules in the boxes are, the rules which determine whether a student is allowed to enrol into a particular course; what is the passing grade of a course (passing grade of a major course is not the same as that of a minor course); whether a student meets the debarment conditions, and whether a student satisfies all requirements for a degree; etc, etc. To illustrate this, a general rule, in the box "Enroled in a course", which governs the acceptance of a student to enrol a course is outlined below,

```
(defrule course-enrolment-accepted
   (enroling-student ?enroling-student)
   (enroling-course ?enroling-course)
   (or (permission ?enroling-student ?enroling-course)
      (and (prerequisites-satisfied ?enroling-student
               ?enroling-course)
          (course-restrictions-satisfied ?enroling-student
               ?enroling-course)))
   (student-registration ?enroling-student)
   (enroling-course-offered ?enroling-course)
   (not (enroled-in ?enroling-student ?enroling-course $?))
   (not (suspension ?enroling-student $?))
   (not (debarment ?enroling-student $?))
   (course-type ?enroling-course ? ?course-type)
   (enroled-in-fact-remark ?remark)
   (not (more-than-three-repeated-courses ?enroling-student)
=>
   (assert (enroled-in ?enroling-student ?enroling-course
       ?course-type ?remark))
   (assert (clear-old-enrolment-facts))
   (assert (update-student-records)))
```

This rule is translated in simple English as,

IF a person (wants);
    to enrol into a course;
    he/she has either permission from the chairman; or
    he/she satisfies all pre-requisites of the course; plus
    he/she fulfils all additional restrictions of the course;
    he/she is registered in the college;
    the course is offered;
    he/she is not taking the course currently;
    he/she is not under suspension;
    he/she is not debarred;
    he/she has not repeated more than three courses;
THEN
    assert the fact that he/she is enroled;
    assert the fact to trigger other rules to remove the
        temporary facts;
    assert the fact to trigger other rules to update his/her
        records.

The "Generic Rule" concept is used so that the similar rules can be consolidated into a single rule and this rule will be triggered by different facts. Such arrangement can avoid to hard code many similar rules and to render the maintenance easier. An example of this concept is the pre-requisites list check for enroling a course. Different course has different pre-requisites and the number of pre-requisites varies from zero to many. Hard coding method is to code for each offered course a rule such as,

(defrule pre-requisites-satisfied-course-xyz

```
    (passed pre-requisite1)
    (passed pre-requisite2)
     .....
    (passed pre-requisiteN)
=>
    (assert (pre-requisites-list-satisfied)))
```

To set the rules for all offered courses is time and main memory consuming. The generic rule method is to set up two rules and to let two rules call each other until some conditions happen. Rule 1 extracts from the facts one pre-requisite and increases the counter until no more pre-requisite. If it is no more pre-requisite, the rule will insert a fact that states "the pre-requisites list check is satisfied". Rule 2 checks whether the student has passed the pre-requisite, if so, it triggers Rule 1 again for next pre-requisite.

Another concern is the *modular programming* of the system. As indicated in the Life Cycle Diagram, each "box" of the diagram represents a procedure similar to that of a procedural programming language. CLIPS is a non-procedural programming language which does not render to partition the rules easily. Using the *small problem concept* is one of the alternatives, but it requires intensive I/O tasks. The system uses the "calling fact" concept which injects into the first pattern of each rule of a "procedure" a calling fact. And, this fact will be asserted into the fact base when this procedure is called and it will be removed right after the procedure is finished. The "calling fact" concept may require more time for matching the patterns of the rules, but it is justified as it cuts down the I/O requirements and it also provides a safety feature to avoid triggering other rules accidentally.

Portability is also a concern of the system design. The "generic rule" concept discussed above is a kind of the portability. Another important concept to improve the portability of the system is the "Degree requirements achievement" concept. The idea of this concept is to decouple the degree requirements into a group of small facts of which a small fact represents a simple degree requirement. The simple facts are stored in a fact base file. Whenever a student finishes a credit or fulfils a degree requirement, a "degree requirements achievement" fact will be

asserted into the fact base and the system will match it with the decoupled degree requirements facts mentioned above. When all degree requirements facts are achieved, the student has completed the program, otherwise, the outstanding requirements are obtainable easily. For different disciplines, it is only required to analyze and to decouple the degree requirements of that particular discipline and store them into the fact base file.

The facts using in the system can be classified into three categories,

1. The permanent facts are those relatively stable facts which will only be changed with the curriculum revision. Examples of such facts are the course-type, the degree requirements facts;
2. The dynamic facts are those facts changing with the progress of the student in his/her life cycle such as the degree requirements achievement, the enroled in course facts of a student at particular time;
3. The control facts are those facts asserted into the fact base to trigger some other rules or to perform some tasks temporary. Those facts for procedural concepts are one of those.

During execution, the permanent and dynamic facts are stored in the fact base and it may be down loaded to the file folders if necessarily. While the control facts will be removed from the fact base as soon as when this fact is no longer required.

COMPUTING ENVIRONMENT/COST

The development of the system was done using two Intel 80286 based microcomputers with conventional configurations. About 300 man hours were spent for the development of the existing system.

FUTURE ENHANCEMENTS

As it stands, the number of students that can be present in the database is (severely) limited by the available main memory. In addition, efficient retrieval of student records requires indexing that must be built during run time, in case that the entire database is loaded in main memory during each run. A solution to both of these problems (space limitations and expense of retrieval) is to use a conventional database management system to store the students' records, and interface the DBMS with CLIPS. Then, all data pertaining to a *particular* student can be retrieved from the database and brought into main memory upon any individual request. Using the indexing features available in most DBMSs, the retrieval would be fast; also, assuming that a small number of students' records (less than 10, for example) can be held in main memory, there would be no space limitations. Since the existing system works with a small number of students present in main memory, once such an interface is built, the system would require no modifications, but it can act as the intelligent module of a larger system.

REFERENCES

William Gale, Nick Chan, and Anestis Toptsis (1990), "CASA: An Expert Database System for Student Guidance", *Proc. Fourth UIC/CCC Partnership Program Conference*, Chicago, May 1990.

N92-16609

A CLIPS BASED PERSONAL COMPUTER HARDWARE DIAGNOSTIC SYSTEM

George M. Whitson

Computer Science Department
The University of Texas at Tyler
Tyler, Texas 75701

Abstract. Often the person designated to repair personal computers has little or no knowledge of how to repair a computer. This paper describes a simple expert system to aid these inexperienced repair people. The first component of the system leads the repair person through a number of simple system checks such as making sure all cables are tight and that the dip switches are set correctly. The second component of the system assists the repair person in evaluating the error codes generated by the computer. The final component the system applies a large knowledge base to attempt to identify the component (components) of the personal computer that is (are) malfunctioning. We have implemented and tested our design with a full system to diagnose problems for an IBM compatible system based on the 8088 chip. In our tests, the inexperienced repair people found the system very useful in diagnosing their hardware problems.

STATEMENT OF THE PROBLEM

Many universities and corporations have a large number of personal computers. Many of these personal computers have been purchased by departments that have little expertise in computer repair. When these computers need to be repaired someone in the department becomes the designated repair person. This person rarely has the expertise to do the repairs and wastes much time acquiring the expertise of a repair person. Since other computers of the same type are usually available it is easy to test to see if a component is faulty if one has the expertise. This is clearly an ideal situation for an expert system. The main purpose of the expert system is to apply the many rules that human computer experts remember to apply such as "is the floppy disk motor running" at exactly the correct time. A secondary purpose of the expert system is to apply a huge database of information about numeric error codes that are known for the system board and component boards in a meaningful fashion.

KNOWLEDGE ACQUISITION

As with all expert systems, the knowledge acquisition phase of this expert system was the most difficult part of its development. To begin the process, I discussed with several expert the overall design of the system and, after several sessions, we agreed on the

overall design presented in the next section. The experts provided lists of error codes and our expert system design was then implemented to give optimal organization of the rules for the error code analysis. The experts then designed a set of help screens that were driven by a knowledge base to lead the inexperienced user through a set of standard system checks to be sure that no obvious problems were causing the computer to malfunction. Finally, the experts were interviewed over a number of sessions to build the main knowledge base that was to assist the inexperienced user in identifying the exact component(s) that was (were) causing the computer to malfunction.

To enhance the ability of the Knowledge Engineers in obtaining the knowledge from the experts, I set up a room with test computers and a large set of boards that had been removed from other computers during repairs. In observing how the repair person discovered the bad boards or components, the rules were determined in relatively short order. After a prototype expert system was developed, a few rules were added and modified and, after our full system was tested, we added a few more.

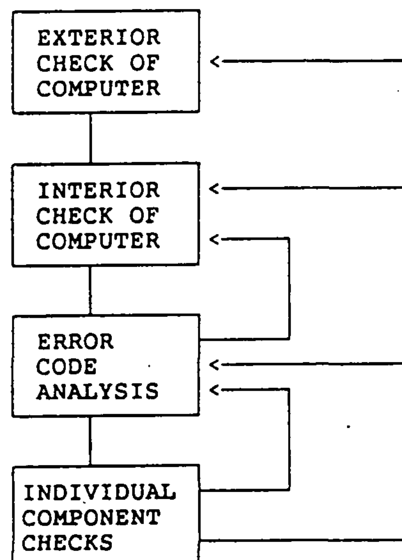
In general, our technique consisted of the following steps:

(1) Design of the phases of the system.
(2) Quick discovery of most of the rules by using a special hardware lab.
(3) Modification of the knowledge base after the development of a prototype.
(4) Further modification of the system after the first full version of the system was used.

SYSTEM DESIGN

The expert system tool that we choose to use for our expert system was CLIPS. The simple structure of the shell and the ability to interface it to graphics and database packages makes it ideal for this type of expert system. After our initial interview of the experts, we decided to use the system architecture shown in Figure 1.

Figure 1.



The system has four major phases that, in general, will be executed in order, with the computer malfunction being detected at the earliest possible time. Each of the phases had its own rule base, with the most complex and interrelated rules being those of the fourth phase. As indicated in the architecture, some of the rules in a later phase could place the user back in an earlier phase.

INITIAL CHECK OF THE SYSTEM

This is an important component of the system. The knowledge base consists of a number of rules that present some help screens in an intelligent fashion and prompts the user to check for obvious system problems that are not related to a component failure. A typical help screen rule is given in Figure 2.

```
(defrule pc_power
  (or ?temp <- (power N)
      ?temp <- (power D)
  )
  =>
  (retract ?item)
  (printout t t t "The power switch is on the right side of PC to the back." t

            " The O setting means OFF and the 1 setting means ON." t
            " If the Power is off (O)" t
            "   Make sure the electrical outlet is working." f
            "   Make sure the power cables are connected." t
            "   Make that all fuses are in good condition." t
            "   Turn Power On " )
  (printout t t t "The fan should make a noise when power is on."
            " Make sure that video monitor is also on. " t
            " Check for any error massages that may appear." t )
  (printout t t "If there is not a DOS boot disk in the disk drive," t
            " or the primary disk drive doesn't work," t
            " the IBM PC will start up in basic mode," t
            " Form basic mode you can execute basic language commands." t )
  (printout t t t "Did the power come on?" t
            "  Enter Y(yes), N(no), D(don't know) : " )
  (assert (power on =(read)) ))
```

Figure 2.

This is a simple rule to tell the user to be sure that the power is on, but one that our inexperienced repair persons suggested as an addition after prototype testing.

This phase has two parts: (1) simple checks to be made with the cover on the machine and (2) simple checks to be made with the cover off the machine. Some checks that the system prompts the user to perform with the cover on are:

(1) check all cables
(2) check power
(3) check monitor power and adjustments
(4) check that software loaded correctly and drive doors closed
(5) reboot system

In addition to the above checks, the users are given as much basic information about the components of a personal computer as possible. For example, one help screen that can be invoked by the rules helps the user identify which card is the graphics adapter card. Some of the checks the user is prompted to perform once the cover is removed are:

(1) check the dip switches
(2) check to see if all boards tight
(3) check to see if motherboard chips are tight
(4) check to see if all internal cables connected are tight
(5) check dip switches on component boards
(6) reboot

Considerable extra information is given the user about the safe procedures to use when working on the computer, such as when to shut off the power to protect a board before it is reseated.

CHECK OF ERROR CODES

The error codes that can be generated by a personal computer during its boot procedure can be massive. While lists are available, they are often missing at repair-time. We included a set of rules in our expert system that would analyze the error code produced by the computer and suggest which components were most likely to be defective. Some of the most popular component boards also have error codes and we added modules for these as well. One of the most useful of the error codes is a memory bank error. One of the rules for our memory bank error check is given in Figure 3.

```
(defrule memory-error
   ?x <- (phase memory-bank-check y)
   =>
   (retract ?x)
   (system "cls")
   (printout t "If an error code is displayed in the pattern of 'XXXX 201 '")
   (printout t crlf "where X is any number or letter and:          " crlf)
   (printout t crlf)
   (printout t "If the first X is '0', there is a memory problem on ")
   (printout t "the system board.                          ")
   (printout t crlf "If the second X is '0', failure in bank 0   ")
   (printout t crlf "If the second X is '4', failure in bank 1   ")
   (printout t crlf "If the second X is '8', failure in bank 2   ")
   (printout t crlf "If the second X is 'C', failure in bank 3   ")
   (printout t crlf)
   (printout t crlf "If the last two XXs are:                   ")
   (printout t crlf " '00'          replace parity module of the bank")
   (printout t crlf " '01'          replace module '0' of the bank")
   (printout t crlf " '02'          replace module '1' of the bank")
   (printout t crlf " '04'          replace module '2' of the bank")
   (printout t crlf " '08'          replace module '3' of the bank")
   (printout t crlf " '10'          replace module '4' of the bank")
   (printout t crlf " '20'          replace module '5' of the bank")
   (printout t crlf " '40'          replace module '6' of the bank")
   (printout t crlf " '80'          replace module '7' of the bank" crlf)
   (printout t crlf " Are you still having problems? (y or n):")
   (assert (answer memory-bank-check =(read))))
```

Figure 3

In general, the error codes for our 8088 expert system were good enough so that this module could tell the user exactly what board to replace. If there were no error codes or if this module could not decide from the error codes what the problem was, the user would go to the next module.

COMPONENT CHECKS

This phase of the expert system begins by asking the user to reboot the system after a 30 second delay and then to pay careful attention to the questions asked in Figure 4.

(1) Is the fan running (y or n). Listen closely for the fan. It may be hard to hear.
(2) The cursor appears on the monitor screen (y or n).
(3) You hear one beep (as opposed to no beep or multiple beeps) (y or n).
(4) The light on the disk drive comes on and you hear the drive motor running inside the disk drive (y or n).
(5) The screen comes up in BASIC if not booting from DOS (y or n).

Figure 4.

This menu is used to select one of the following modules of rules for the user.

(1) power supply
(2) graphics adapter and monitor
(3) keyboard
(4) floppy disk controller and drives

These modules were seen as those most often giving trouble at this point by our domain experts. If the user's answers to the first menu questions result in no rules in (1) to (4) being used, then a number of other options are presented to the user each of which can trigger a module if the user answers the prompting questions with enough information to indicate that a diagnostic module should be invoked. The main option that we have implemented to date is a hard drive and adapter module. Options to check communications cards and printer cards have also been added. In future versions we would like to add modules to cover modems and tape drives. There are many interrelated rules in this phase. A typical section of code from our current system is shown in Figure 5.

```
(defrule No-motor
     ?x <- (motor-spinning n)
     ?y <- (phase disk drive)
     =>
     (retract ?x ?y)
     (printout t crlf "Please check the power supply unit." crlf)
     (printout t "If the power supply is ok, then replace the disk drive.")
     (printout t crlf)
     (assert (last-choice 3)))

(defrule check-BASIC
     (phase BASIC)
     =>
     (system "cls")
     (printout t crlf "Insert a DOS disk in drive A: and reboot." crlf)
     (printout t crlf "Watch for DOS to boot up." crlf)
     (printout t crlf "Did DOS boot? (y or n): ")
     (assert (DOW BOOTS =(read))))
```

Figure 5.

SUMMARY

Our current system has about 150 rules. This is a little misleading since some of the rules result in the user being presented a menu selection that in reality is really another set of rules. The system works fairly well, although we consider it still to be in the development phase. For many problems, the user will be guided to exactly what's wrong. In some cases the system gives up because it does not have enough knowledge. It is clear that such a system could be perfected. What is not as clear is whether or not it is really practical to develop such a system for every type of personal computer. We think it is and plan to continue our development of the basic system.

Some prototypes of our current systems have been developed as class projects in our Expert Systems course at The University of Texas at Tyler over the past few years and several students have developed prototype systems as independent study projects. The final system described in this paper has had input from many students at our University. Unfortunately, there were too many to give credit to all by name, thus, I simply mention their contribution in this summary.

As an extension of the ideas of this paper we are now applying the basic systems architecture to developing a hardware diagnostic system for any system with many individual components. As a part of our extension we are embedding several neural networks that will determine when sets of electrical and mechanical signals should fire a CLIPS rule. At this point it appears that our basic architecture is a solid basis for this extension.

REFERENCES

IBM Technical Reference for the IBM PC and Options and Adapters.

IBM Hardware Reference for the IBM PC.

Giarrantano and Riley. (1989). Expert Systems, PWS-KENT.

Hayes-Roth, Waterman, Lenat. (1983). *Building Expert Systems,* Addison-Wesley.

Michalski, Carbonel and Mitchell. *(1986. Machine Learning: An Artificial Intelligence Approach-Vol. 2,* Morgan Kaufmann.

Westphal, Chris (ed). (1989). *SIGART Newsletter (special edition on Knowledge Acquisition),* ACM Publications, New York, N.Y. 10249, pp 6,198.

Whitson, Wu and Taylor. (1990). *Using an Artificial Neural System to Determine the Knowledge Base of an Expert System,* Proceedings of the ACM Symposium on Small Computers, Washington.

SESSION 8 A

PRECEDING PAGE BLANK NOT FILMED

N92-16610

PVEX - AN EXPERT SYSTEM FOR PRODUCIBILITY/VALUE ENGINEERING

Chun S. Lam* and Warren Moseley**
Computer Science Department
The University of Alabama in Huntsville, Huntsville, AL 35899

ABSTRACT. This paper describes PVEX, an expert system that solves the problem of selection of the material and process in missile manufacturing. The producibility and value problem has been deeply investigated in the past years, and was written in dBase III and PROLOG before. This project represents a new approach to it in that the solution is achieved by introducing hypothetical reasoning, heruistic criteria integrated with a simple hypertext system and shell programming. PVEX combines KMS with Unix scripts which graphically depicts decision trees. The decision trees convey high level qualitative problem solving knowledge to users, and a stand-alone help facility and technical documentation is available through KMS. The system the authors developed is considerably less development costs than any other comparable expert system.

INTRODUCTION

The selection of the material from which the structure to be made, and the selection of the process by which the shape of the structure is assembled are two major steps in missile manufacturing. The procedures by which the material and the process are determined are to a great part a function of the experience of the design and manufacturing engineer, of the performance requirements for the final product and of the cost of the material and the process. For sophisticated missile systems, these procedures are interrelated and difficult to separate, particularly where the engineer's experience is concerned. Accordingly, some means is needed for capturing the knowledge and experience of the design and manufacturing engineer at the onset of the design of the structure and the manufacturing processes. If this means were properly defined, it could also be used to "reverse" engineer structures already designed, and then in turn determine more cost effective ways of producing these structures. The state-of-the-art in expert systems provides the essential elements for developing such a capability. This paper describes the steps taken to develop such a system, the Producibility/Value Engineering Expert System (PVEX), from a rudimentary prototype written originally in dBASE III.

PVEX, as it has evolved over its development, links a computer-aided design system and a very powerful database management system together to form a highly "intelligent" design tool useful both for initial design and for "reverse" design of metal structures for missile systems. It also has the necessary interface provisions for being incorporated into an automated computer integrated manufacturing (CIM) system, which is the logical extension for the system. The system is hosted by a Sun Microsystems 3/60 workstation.

REQUIREMENTS

Compatible with both the Sun 3/60 workstation and CADDS was the basic requirement for developing PVEX, which meant that PVEX must be also UNIX based. Since so much of the

* Lam's current address is Intergraph Corporation, One Madison Industrial Park, Huntsville, AL 35894

** Moseley's current address is Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213

PRECEDING PAGE BLANK NOT FILMED

design knowledge consisted of materials and process databases, capable of accessing these databases was the second requirement for PVEX. Because of the practical difficulties in creating an all-inclusive database, a subsidiary requirement was identified that PVEX be able to access both its own and external databases. Thus, if some seldom used material were to be considered, PVEX could access an appropriate database through a telecommunications line when necessary. The third requirement for developing PVEX was that the system must provide the means for handling materials and process selection heuristics; that is, the procedures and logic by which the material and forming/shaping processes are selected by the design engineers.

These three major requirements imply several subordinate requirements, including finding appropriate software tools, selection of the development process, development of the materials selection and process selection heuristics, and coding and validation of the system. The most crucial of these requirements was the development of the material and process selection heuristics, as this is one of the least well documented aspects of the design process. Further, this is a process that is heavily dependent upon the experience of the designer, so this in effect forced the requirement that the PVEX system be capable of capturing this experience as it would be used by the designer.

There were several other requirements identified for the system, in part as a consequence of good software practice and in part as a demand that the user be able to use the system without extensive training. Thus, the system is highly interactive, with an abundance of help facilities. To a great extent, the software used is self-documenting, so that the code itself serves as its own documentation. Finally, it seemed reasonable that the system be easily transportable to other UNIX systems with little or no modification.

DEVELOPMENT TOOL SELECTION

The rudimentary version of PVEX was written in dBASE III, which proved to be inadequate for the intended use. A subsequent version was written in PROLOG, but it too was inadequate. It was evident that the selection of the language and development tool required for the PVEX should be investigated in depth. Accordingly, there were five tools that were considered in some detail. These were:

- Automated Reasoning Tool (ART)
- CLIPS
- Knowledge Pro
- Knowledge Management System (KMS)
- Nexpert *Object*

The procedure was to set up prototypes with each of these tools and then select the best one. As it turned out, the cost and availability constraints associated with Nexpert *Object* ruled it out from further consideration.

ART

ART is an expert system development tool marketed by Inference Corporation. It is a full programming environment providing reasoning and database access via an inference engine and an object-oriented knowledge representation mechanism.

ART was considered as the reasoning component of PVEX in the early stage of development. Domain knowledge such as rules and heuristics were implemented using the ART language. The advantages of using ART as the reasoning facility are: (1) ART language is very powerful for organizing declarative knowledge into a knowledge base, and for organizing knowledge into rules. (2) ART includes an interactive development environment (the ART Studio) that offers monitoring and debugging aids for application development. (3) It provides the facility (ARTIST) for building complete a user interface including menus and graphical displays.

However, ART maps the language into Lisp code and integrates it into the Lisp environment, and Lisp is not available at Production Engineering Division (PED). Also, ART rule bases and knowledge bases are difficult to modify/maintain by personnel without any previous experience in logic programming, and this is a burden to the user. For example, rules must be written in a special format to take advantage of backward chaining, and then they are not useful for forward chaining. For these reasons, ART was not considered a viable tool for PVEX.

CLIPS

The C Language Production System (CLIPS) is an expert system shell developed by the NASA/Johnson Space Center. CLIPS was designed specifically to provide easy portability, compatibility and lower cost. The primary representation methodology is a forward chaining rule language based on the RETE algorithm. Advantages to this package are: (1) CLIPS was free of cost to PED. (2) The knowledge-base structure contains all of the heuristical rules in one database, thus making the database very maintainable. (3) An inference engine controls the overall execution. (4) CLIPS code is transportable to any machine with a standard C compiler.

However, CLIPS is an inflexible and fairly limited language. Although it provides an inference engine, only forward chaining is supported. It does not provide an object-oriented knowledge representation system, and efforts to create one became quickly cumbersome. Programs in CLIPS are complicated and difficult to make user friendly. In addition, it was determined that any necessary reasoning could be accomplished via a hypertext-driven system, and that a true inference engine, while desirable, was not mandatory. Since an objective of PVEX is to provide a powerful system while also providing a very friendly environment, CLIPS was also not considered as a viable tool.

KNOWLEDGEPRO

Knowledge Pro is a PC-based expert system shell was designed to provide users with a programming environment free of complex syntax that is easy to learn and use. Its main feature is the ability to incorporate hypertext documents. In such a document, any area of text throughout the document can be linked to additional levels of information that clarify the concept. Using such an expert shell can provide an on-line help facility for viewing different levels of help at anytime. Giving the user the capability to decide what to read next depending on interest and need is referred to as "non-linear" text. Advantages to this system are: (1) inference capability/rules; (2) forward and backward chaining; (3) automatic linking of text / hypertext; (4) help facility; (5) windowing system; and (6) menu generator.

Unfortunately, this application program was found to be incompatible with the Sun environment. Other disadvantages are listed as poor user interface, length of code, block structure limits flexibility, and slow database manipulation. Thus, KnowledgePro was also ruled out as a viable tool.

KNOWLEDGE MANAGEMENT SYSTEM (KMS)

Knowledge Management System (KMS) is a distributed hypermedia-based knowledge management system developed by Scribe Systems, Inc. The major components of KMS include (A) the facility to build up frames and links between frames which contain the knowledge associated with the domain in the actual links (B) the linear document formatting program facilitates moving through the hierarchy (tree) structure of frames; and (C) the Action Language which meets the programming needs of end users. "A frame is a screen-sized 'worksheet.' Like a sheet of paper, it may contain any arrangement of text, graphics, and images."

KMS is very attractive because its frames are treated as a document and the level of effort to maintain the links is analogous to the documentation maintenance process. Documents are organized as a hierarchy and hence lend structure to the organization of the knowledge and its links. This also facilitates the browsing process when working with a KMS document in the

interactive mode. KMS provides a rich hypermedia-based knowledge representation with template sharing and a context-sensitive cross referencing capability. It is also compatible with UNIX and the Sun workstation.

Since the producibility heuristics acquired from the PED domain expert can be readily organized into a hierarchy which is strongly related to a tree structure, KMS was the logical choice as the primary reasoning facility in PVEX. Each KMS frame represents a decision node in the heuristic tree, and the path between a node and its subtree is joined by a link. Another advantage of using KMS is that there is no need to build an independent user interface, because the user interface is built when the decision tree is constructed. KMS provides a very user friendly environment to create, modify, or delete objects in a frame as well as frames in a frame set. By incorporating KMS Action Language, the reasoning facility may easily interface with databases via UNIX scripts.

KMS also was utilized as the explanation and help facility because its hypermedia/hyper-graphics system is well-suited to implementing non-linear on-line documents. The reasoning and explanation facility thus are closely coupled without requiring much effort to interface them. Thus, KMS appeared to be the most suitable of the several application programs for use in developing the P/VEX model.

DATABASE TOOLS

The necessity to access several databases from CADDS and/or KMS required that the choice of database tools be considered. Only two choices were available, AWK and INGRES. Both are compatible with UNIX based systems.

The data with which PVEX is dealing is materials and processes information. This information is primarily static with the exception of costs, thus making a row/column database structure highly useful.

AWK was originally designed and implemented as a part of a UNIX tools experiment. It is a simple programming language that handles simple mechanical data manipulation very efficiently. An AWK program usually is a sequence of patterns and actions indicating what to look for in the input data and what to do when it is found. AWK searches a set of files for records matched by patterns. When a match is found, the corresponding action is performed. AWK expects rows or records containing columns or fields in a simple ASCII format. Maintenance of AWK programs is simple.

On the other hand, INGRES was prohibitive because of expense and license restrictions. AWK was chosen as the database manipulation tool for its ease of use, low cost, and portability.

PVEX FUNCTIONALITY

As stated in the Introduction, the host of the system is a Sun Microsystems workstation, operating under the UNIX operating system. The chosen expert system shell, KMS, operates through script files in an interactive manner. This allows access to the CADDS applications program from within KMS, so that a part and its specifications may be addressed directly from KMS. Context-sensitive help facilities assist the user in both use of the system and in capturing the reasoning of the user during a work session with PVEX. The system, as defined under the operating environment, is self-documenting and is fully transportable to other hosts operating under the UNIX operating system (see Figure 1).

The heuristics that have been incorporated into PVEX are written in individual frames, rather than in the form of a rule base. In so far as the user is concerned, there is no difference in performance. Knowledge is stored into four major databases; (1) the materials database; (2) the processes database; (3) the knowledge database; and (4) the comparator database. These databases, together with the user's responses or requests, are used in making recommendations to the user for materials and process selection. PVEX does not itself select a preferred material or process; that is the user's responsibility.

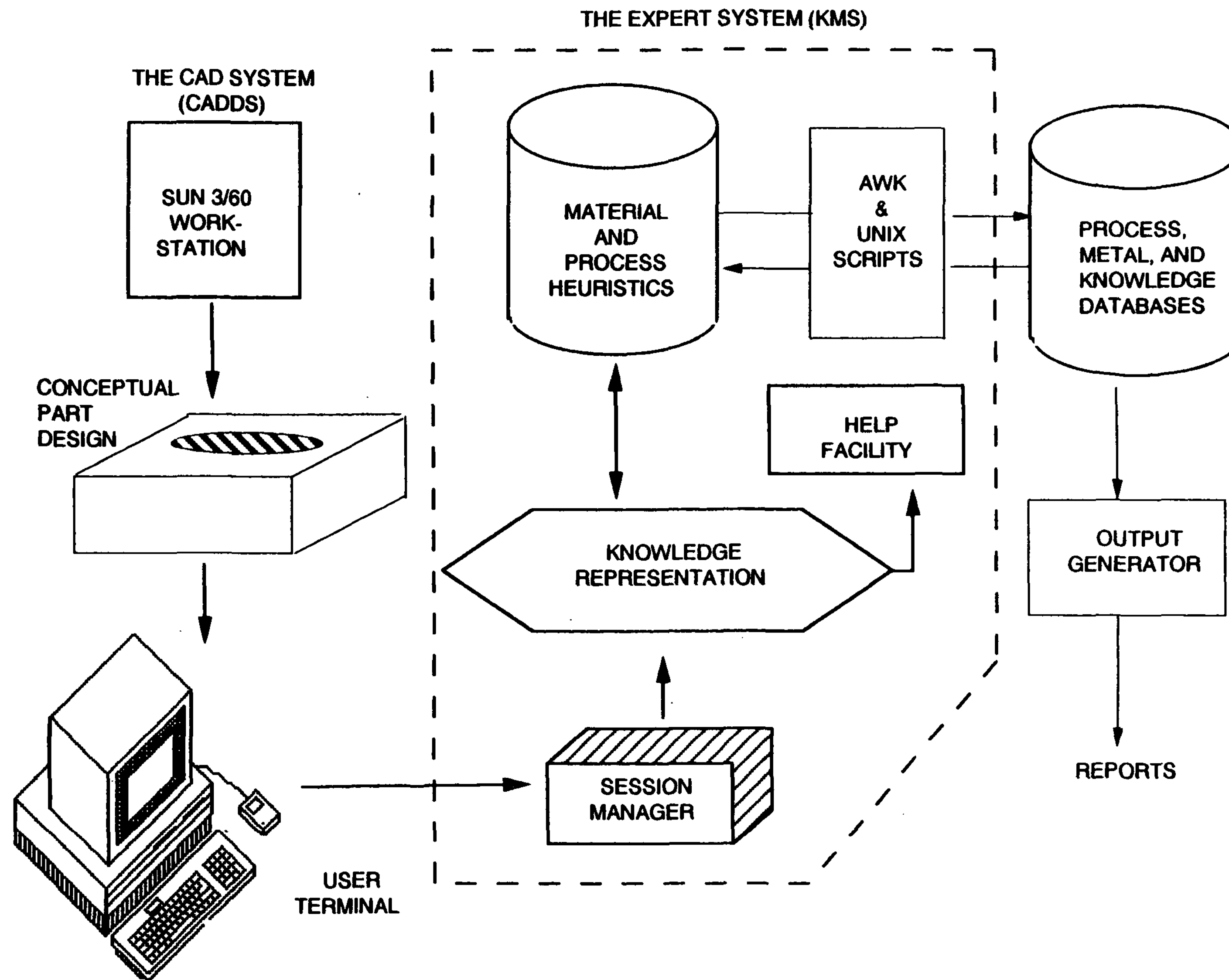


Figure 1. Producibility/Value Engineering Expert System Overview

SYSTEM FUNCTION

The user's objective in using the system is to generate a feasible recommendation for the material and process to use for a specific part in a weapon system. First, the system gathers general information pertaining to the user through a script file. The information includes user's name, date, part name, system name, drawing name, and drawing number.

When this information is provided by the user, PVEX prompts for answers which are used to determine the path the user will follow while processing. The answers are stored in temporary files. The system then asks the user if an initial material is known. If there is a known initial material, a script file is invoked to retrieve the initial material. However, if the user does not know an initial material or would like to verify a predetermined material, the system begins to traverse the material selection tree. Once flow begins in the material selection tree, PVEX asks a series of questions based on heuristical knowledge in friendly menu driven screens. Answers to these heuristical questions are stored in a data file. This data file contains vital facts based on the responses to the heuristical questions concerning materials. This file acts as a facts file to show the user WHY particular materials were chosen. Control is performed through script files called from within KMS which interface with the UNIX working environment. Continuing with the system, a script is invoked to open a window and to search the material database and to list feasible materials. Control of the order of material operational database manipulation programs (AWK) is captured in this script. If no materials were chosen, a message will appear to inform the user. Next, the user is prompted for further information about each material. If the selection is no, the system will return control back into KMS. However, if the selection is yes to requiring more information pertinent to material selection, control is passed to a awk file. This awk file returns a list of material properties for each selected material. This list is displayed on the screen or is sent to a printer. Once the process is finished, control of the system returns to KMS.

Once control is returned to KMS, the user can either A) continue with process selection, B) re-evaluate material selection, or C) exit from the system. Depending upon the selection, the user will either flow down the process hierarchical tree or have further options, which include generating a report.

The process hierarchical tree flow works in much the same manner as the material tree flow. The process tree flow asks the user heuristical questions based on processes and stores the associated data in a file. This file will act as a facts file to show the user WHY particular processes were chosen.

DATABASE IMPLEMENTATION

Database generation was necessary to provide the system with knowledge about materials and processes so that it reasons effectively when generating producibility and value engineering alternatives. A survey of existing databases concluded that there are no readily available databases which provide sufficient detail and proper format to meet PED specifications for an internal database. The four databases were then generated through efforts of PED domain experts and UAH research efforts. The comparator and knowledge databases are strictly knowledge databases whereas the process and metals databases are strictly numerical databases. The comparator database is a list of processes matched with known materials to which these processes apply. However, the knowledge database is a list of processes and particular features that are associated with the process. Not all features can be utilized by a particular process.

HELP FACILITY

An on-line help facility is integrated into PVEX through KMS frame links. It makes finding information easy and quickly. The help facility consists of 280 frames which contain in-depth definitions and graphic illustrations of typical material and process selection terms. Information included in this facility was gathered from experienced engineers and from well-established up-to-date handbooks of metals and processes.

The help facility was designed so that it could be used as a stand-alone utility, as well as an aid to the engineer who is operating the PVEX system. It can be accessed as a stand-alone utility through the PVEX initial frame. From this point, control is maintained through submenus and return commands. It is structured so the user can go back and forth from menu to term and back to menu to make several selections in one run. An example of the structured help is shown in Figure 2. In addition to being a stand-alone utility, the help facility can also be accessed from the PVEX heuristic frames. Control is passed to the help frame and then returns the user to the point in the system where he needed the help. The user can then continue with the expert system. An example of stand-alone operation is given in Figure 3.

GRAPHICAL INTERFACE

To enhance the user interface, the expert system provides on-screen pictures, or graphics, to provide a more detailed description of the current question or statement. Some of these graphics were simply drawn on the Sun workstation using CADDS and saved as a Sun format "rasterfile" using the screendump command. These images are required to be 1-bit deep (black & white), as opposed to 8-bit (color) rasterfiles. Other, more complex images, that appeared in books and other printed publications, were obtained using a scanner connected to computers other than the Sun workstations.

The scanned images were imported into KMS in a couple of days which in turn would have required weeks if drawn directly in KMS or CADDS. Advantages of this capability is that PED can view current images and add more images to the PVEX system, as desired, while running on any UNIX machine that is able to view rasterfiles.

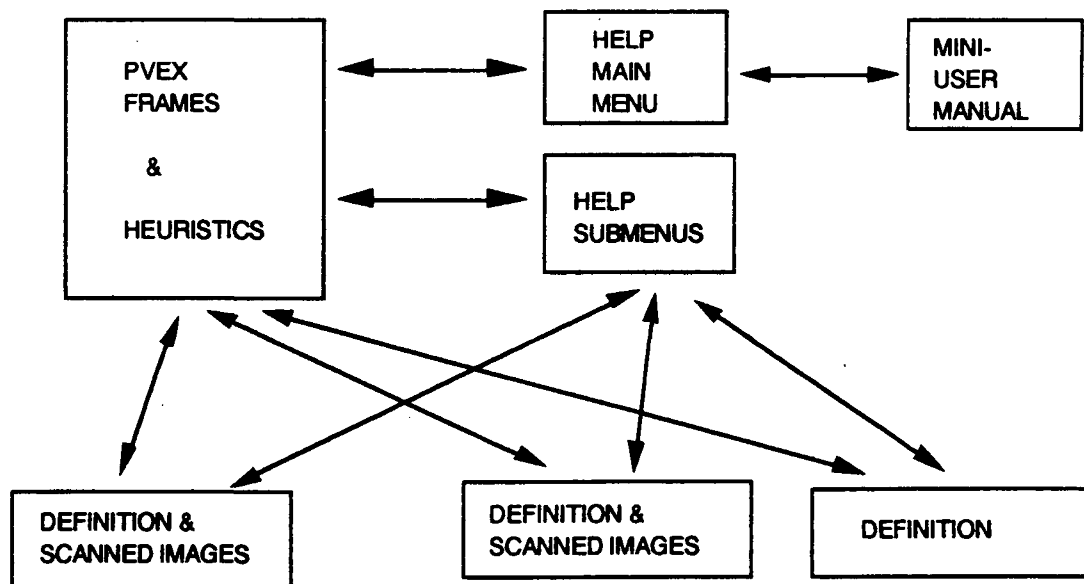


Figure 2. System Building Blocks of the Help Facility

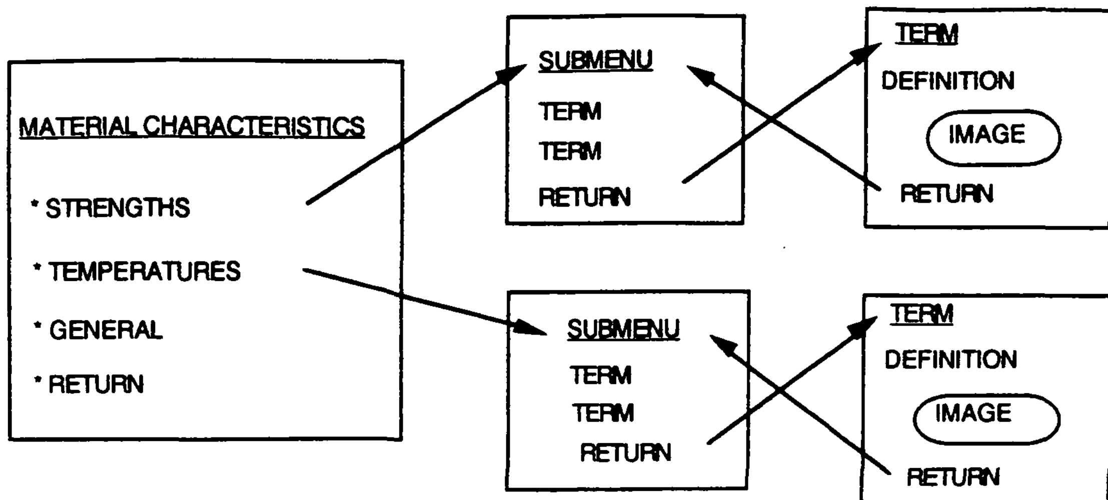


Figure 3. Stand-Alone Operation of The Help Facility

CONCLUSIONS

As the foregoing discussion indicates, PVEX is an extremely powerful, flexible and easy-to-use design and design audit system. It can be up-dated "on the fly" and is readily transportable to other UNIX based systems. The coding is self-documenting, and its ability to capture knowledge from the user, databases and the CADDS system provides an ability heretofore not realizable. While KMS itself does not use inferencing rules in the traditional sense of conventional expert systems applications programs, its frame structure accomplished the same purpose and also provides the necessary linking to other frames or utilities. Since the system also supports the display of scanned images of parts and assemblies not already in the CADDS file format, the evaluation capability of PVEX extends beyond the capabilities of CADDS.

The capability provided by PVEX is necessary for achieving concurrent or simultaneous engineering; e.g., the design of the product and the process at the same time and in an interactive mode. As is evident from the structure of PVEX, the material is selected first, followed by the process selection. These steps can be iterated as many times as necessary in order to obtain an optimal matching between material and process.

The structure of PVEX can be easily extended. As the system is structured, composite and non-metallic materials can be incorporated into PVEX with very little modification. The basic form of PVEX can be extended to other material and process selection cases. While there are a number of refinements that could be added to PVEX, it is in its present form a useful design and production tool ready for use by producibility engineers in designing new products or in the re-design or evaluation of existing products.

ACKNOWLEDGEMENTS

The authors wish to thank other PVEX team members Fred Anderson, Carolyn Ausborn, Frank Baird, James Clark, R.J.Lin, Ben Lowers, and Leonard Yarbrough for their contribution to this work.

REFERENCES

1. Aho, A., et. al., *The AWK Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
2. Aho, A., et. al., "AWK - A Pattern Scanning and Processing Language," *Software Practice and Experience,* July 1978.
3. Akscyn, R., et. al., "KMS: A Distributed Hypermedia System For Managing Knowledge In Organizations," *Communications of the ACM*, July 1988, Vol. 31, Number 7.
4. Conklin, J., "A Survey of Hypertext," *IEEE Computer*, Sept. 1987.
5. Kerrighan, B., and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

N92-16611

RULE GROUPINGS IN EXPERT SYSTEMS USING NEAREST NEIGHBOUR DECISION RULES, AND CONVEX HULLS.

Stergios Anastasiadis

McGill University, School Of Computer Science,
3480 University Avenue,
Montreal, Quebec, Canada, H3A 2A7.
stergios@bnr.ca
calserg@homer.cs.mcgill.ca

Abstract. Expert System shells lack in many areas of software engineering. Large rule-based systems are not semantically comprehensible, difficult to debug, and impossible to modify or validate. Partitioning a set of rules found CLIPS, into groups of rules which reflect the underlying semantic subdomains of the problem, will address adequately the concerns stated above. In this paper we introduce techniques to structure a CLIPS rule-base into groups of rules that inherently have common semantic information. The concepts involved are imported from the fields of A.I., Pattern Recognition, and Statistical Inference. Techniques focus on the areas of feature selection, classification, and a criteria of how "good" the classification technique is, based on Bayesian Decision Theory. We discuss a variety of distance metrics for measuring the "closeness" of CLIPS rules and describe various Nearest Neighbor classification algorithms based on the above metrics.

INTRODUCTION

Knowledge is a collection of related facts that can be used in inference systems such as CLIPS. A production rule is a method for representing knowledge, among others that exist in the world of knowledge representation, such as inclusion hierarchies, mathematical logic, frames, scripts, semantic networks, constraints, and relational databases[10]. In CLIPS this method of representation is particularly appropriate since knowledge is "action-oriented". Although production rules lend themselves to be used in inference systems, validation and verification of syntactic and semantic correctness is difficult to achieve.

Rule-based systems have been developed with a very narrow scope. They don't allow interaction with other types of knowledge and mechanisms used to derive the required results are driven by pure syntactical procedures. Therefore, to move away from isolation, rules must not embody all of the knowledge in a system and furthermore, rules must encapsulate semantic information that would make the derivation process more sophisticated and efficient.

Production systems are not developed one rule at a time, but the complexity and success of the system relies on the inter-dependencies and interactions between rules[7]. Small systems might contain a couple of hundred rules, resulting in a total of many hundreds and/or thousands of patterns that must be matched in order to logically proceed to another derivation. This results in exponential growth of pattern matching without taking advantage of the dependencies and interactions that exist between rules ; which dependencies and interactions were the reasons why the system was established to begin with.

CLIPS does provide saliences for rules as a semantic feature of the rule-base. This alone does not help in removing redundancies or improving the validity of the system. Grouping the rule-base into groups of related rules will extract the underlying domain knowledge from the rule-base. Attaching then priorities to each group, and to each rule in a group will further help the system in its derivation process, in its ability to distinguish between various types of knowledge [2,3], and in its ability to exchange and communicate with other types of knowledge in order to validate, verify, and explain its conclusions [1].

In this paper we will present a mechanism to measure distances between CLIPS rules, and using those results we will present classification schemes that will insert rules into the appropriate group.Then we will also show how accurate the classification schemes are by providing a statistical estimation of misclassification. This estimation will help when one has already classified the rules into groups, and requires to insert a new rule into the "best" group. Finally, we will make some concluding remarks and some possible extensions to the scheme.

OVERVIEW OF METHOD

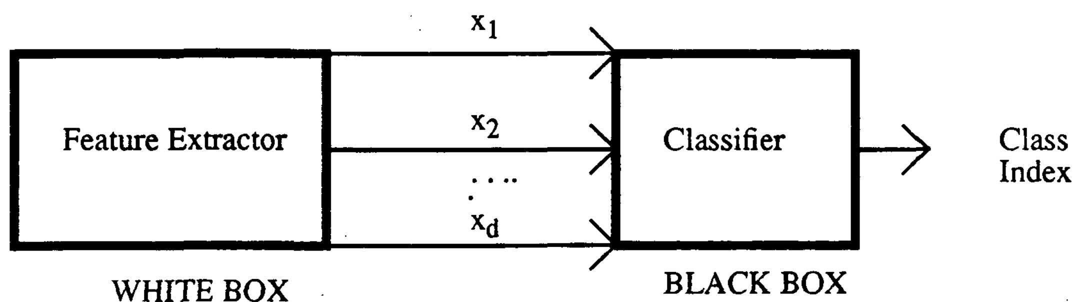
The general approach is to select the crucial information from a rule, and classify it in the appropriate group. Therefore, feature selection, extraction, and classification is the logical route followed by this scheme. One can either allow the system to generate the groups on its own, or the user can choose a rule that represents a group, and then the system classifies the other rules accordingly.

The important characteristics that are going to be used from a rule are inserted in a feature vector X. The required parsing of the rule is done in order to fill in the vector. Information that is selected from a rule are patterns from the antecedent, consequent, or from both. Hence, each rule has its own feature vector,

$$X_i = (x_1, x_2, \ldots ,x_d) ,$$

where d is the feature vector size, and i is the rule number. The feature vector size has a value that is set before run-time, and depends on how complex the rules are. The more patterns that exist, the higher value d will obtain.

The general flow is as follows :



The "WHITE BOX" of the feature extractor is application dependant and produces the feature vector for a given rule. The "BLACK BOX" of the classifier is a universal method comprised of a discriminant function and a maximum selector that returns some class index indicating which group the rule has been put in. The "BLACK BOX" classifier is shown in Figure 1.

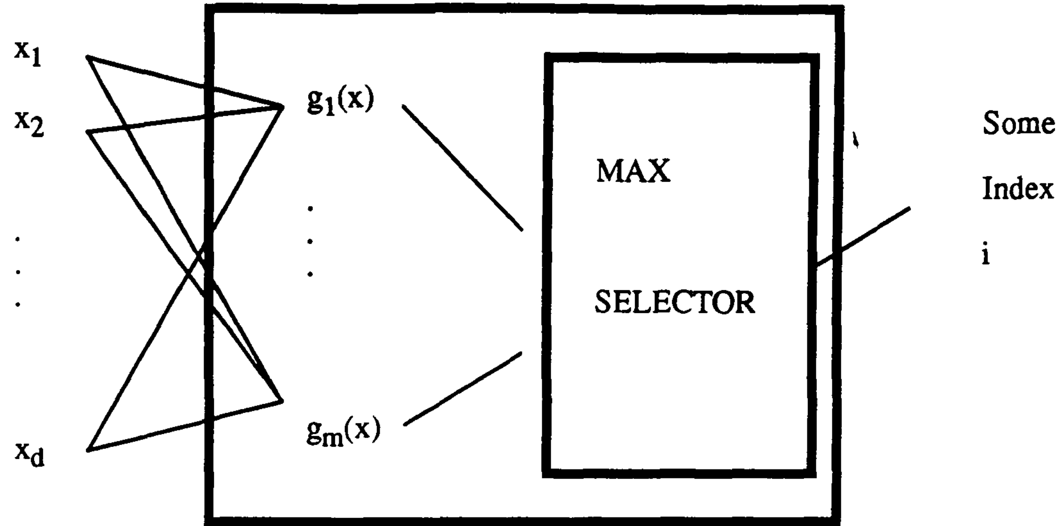


Figure 1 : "BLACK BOX" Classifier.

The discriminant function indicates the confidence that a feature vector comes from a particular class or group of rules. In Figure 1 the function $g_i(x)$ is the discriminant function, and m is the number of classes or group of rules that exist in the system. The selector is used as a "voting" mechanism to finally determine where the rule belongs.

Graphically, decisions are represented as regions, and the boundaries between regions are determined by the difference among the discriminant functions that represent each region. For example given two feature vectors X_1, X_2, and regions R_1, R_2, then the boundaries are found through the equation :

$$g_1(X) - g_2(X) = 0$$

The regions might look like what is shown in Figure 2.

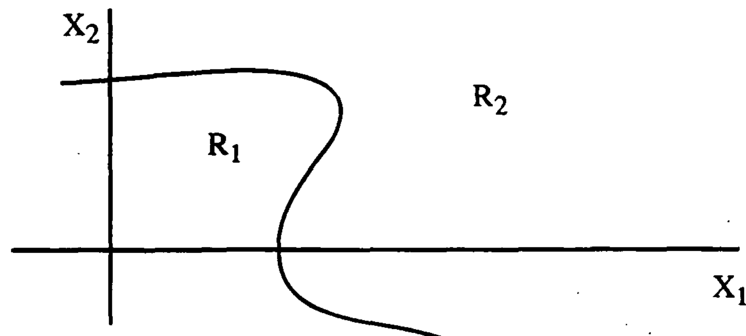


Figure 2 : Boundaries between regions.

DISTANCE METRICS

Different metrics capture different information from the rule-base of an expert system, and each rule-base lends itself to a particular distance metric. In CLIPS a rule-base is made up of rules with antecedents and consequents. Each antecedent or consequent is composed of patterns which match facts in working memory during run time. Each pattern is further divided into tokens, which in CLIPS can be a word, string or number.

As part of our formulation reserved words are ignored. All strings and variables are suppressed. Strings convey little domain knowledge, and variables have values only during run time, are local to a rule, and hence cannot carry along information when we are statically relating rules.

Here are the four distance metrics that will be used in our classification scheme :

$$\text{Dall}(\text{Rule}(1,2)) = \frac{\text{NumOfTokensInRule1} + \text{NumOfTokensInRule2}}{\text{NumOfCommonTokensFromRule1andRule2}}$$

$$\text{Dcon}(\text{Rule}(1,2)) = \frac{\text{NumOfTokensInConOfRule1} + \text{NumOfTokensInConOfRule2}}{\text{NumOfCommonTokensInConFromRule1andRule2}}$$

$$\text{Dant}(\text{Rule}(1,2)) = \frac{\text{NumOfTokensInAntOfRule1} + \text{NumOfTokensInAntOfRule2}}{\text{NumOfCommonTokensInAntFromRule1andRule2}}$$

$$\text{Dca}(\text{Rule}(1,2)) = \frac{\text{NumOfTokensInConOfRule1} + \text{NumOfTokensInAntOfRule2}}{\text{NumOfCommonTokensInConFromRule1andAntFromRule2}}$$

If any of the denominators of the above metrics is zero, then the value will be infinity. This is the case because we can see that the commonality between two rules is inversely proportional to the distance between them. Hence, a small distance indicates that the two rules have many patterns in common.

The Dca metric is used when the fundamental relationships between the rules of the system drive the derivation process from the consequent of one rule to the antecedents of other rules (refer to example 1). If in a rule-based system a large amount of knowledge is present in the antecedent of the rules, then the Dant metric would be appropriate. Similarly for the consequent part, then Dcon would be the metric of interest. Finally, it might be the case, as in diagnostic systems, that rule-interdependency does not allow one to use one of the above three metrics. Both the antecedent and the consequent of each rule play a different role each time a new derivation is to be made. So, since we want to take both sides into consideration, the Dall metric will be used (refer to example 2).

The following examples illustrate how the above metrics are used, and how one makes the decision on using a particular metric :

EXAMPLE 1 :

```
(defrule rule1 (a) => (assert (b))
(defrule rule2 (b) (c) => (assert (d))
(defrule rule3 (d) (e) => (assert (f))
```

Dca(1,2) = 3/1= 3; Dant(1,2) = Dcon(1,2) = INFINITY; Dall(1,2) = 5/1 = 5.
Dca(1,3) = Dant(1,3) = Dcon(1,3) = Dall(1,3) = INFINITY.
Dca(2,3) = 3/1 = 3; Dant(2,3) = Dcon(2,3) = INFINITY; Dall(2,3) = 6/1 = 6.
Therefore, Dca would be the appropriate metric to use.

EXAMPLE 2 :

```
(defrule rule1 (a) (b) => (assert (c)) (assert (d)))
(defrule rule2 (a) (e) => (assert (f)) (assert (g)))
(defrule rule3 (c) (d) (h) => (assert (i)))
```

Dca(1,2) = Dcon(1,2) = INFINITY; Dant(1,2) = 4/1 = 4; Dall(1,2) = 8/1 = 8.
Dca(1,3) = 5/2; Dcon(1,3) = Dant(1,3) = INFINITY; Dall(1,3) = 8/2 = 4;
Dca(2,3) = Dcon(2,3) = Dant(2,3) = Dall(2,3) = INFINITY;
The metric Dcon is ruled out, but Dca and Dant have unstable behaviors. Therefore, Dall would be selected.

CLASSIFICATION

The approaches that will be presented here are algorithmic and graphical. Three classification schemes are introduced, and an attempt will be made to analyze these schemes through some graphical representation. Then the Nearest Neighbor rule will be introduced, along with how the three schemes fit in to the decision making.

The basic assumption here is that initially each rule belongs in its own group, although the user has the ability to create initially his own groups with a representative rule from the rule-base being the initial member of each group. Lets begin with some definitions.

Let $\{X,C\} = \{X_1, C_1; ...; X_N, C_N\}$ be the set of N pattern samples available, where N is the number of rules in the CLIPS rule-base, and X_i and C_i denote, respectively, the feature vector and the label for the classification information of the ith pattern sample. Values for C_i are in the range [1,m], where m is the number of groups present in the rule-base as defined above. It's obvious with the above assumption that $1 <= m <= N$.

In the following schemes training and testing a set of data samples is frequently used. Training a data set means to use a decision rule (and in our case it will be the Nearest Neighbor rule) that will insert each of the feature vectors into the appropriate group. Given this initial classification one takes another set of data, that is possibly mutually exclusive from the training set, and examines if the feature vectors in the set for testing could be classified correctly in the classification scheme derived during the training session.

Here are the three methods for classification :

METHOD 1 : (Resubstitution - R method).[5]

The steps for classification are :

1) The classifier is trained on {X,C}.

2) The classifier is tested on {X,C}.

METHOD 2 : (Holdout - H method).[4]

The steps are :

1) Partition {X,C} into K randomly chosen pairs of sets of equal size

$$\{X,C\}^1_a : \{X,C\}^1_b , \ldots , \{X,C\}^K_a : \{X,C\}^K_b$$

such that for i = 1,...,K , $\{X,C\}^i_a$ and $\{X,C\}^i_b$ are mutually exclusive.

2) For i = 1,...,K , train the classifier on $\{X,C\}^i_a$ and test it on $\{X,C\}^i_b$.

METHOD 3 : (Leave-me-out - U method).[8]

The steps are :

1) Take one pattern sample $\{X_i, C_i\}$ out of {X, C} to create $\{X, C\}_i$.

2) Train the classifier on $\{X, C\}_i$.

3) Test the classifier on $\{X_i, C_i\}$. If X_i is classified into the group associated with C_i, then set $e_i = 0$; otherwise $e_i = 1$, where e_i acts as an error indicator.

4) Do steps 1)-3) for i = 1, ... , N to obtain values for all e_i's.

It turns out that the R-method is very biased as one will see when calculating the probability of misclassification. For the H-method traditionally 50 percent of the available samples have been used for training, and the other 50 percent for testing, and in the U-method when N is particularly large the method works well, but when N is relatively small it is not very reliable. Furthermore, it is an overly optimistic estimate of performance. Although lots of computations are required, the method tends to be unbiased and uses all the available data with high variance in discrete cases.

Apply the Nearest Neighbor rule with one of these methods (see next section on Nearest Neighbor rule), and fill in the values in {X, C}.

Now how can one graphically take this information and check for correctness? This can be done through what is called a convex hull. Let R denote the set of real numbers. Then R^2 denotes the set of all points in the plane. Define a set of points D from R^2 as being a **convex domain** if, for any two points x and y in D, the line segment xy is entirely contained in D. So if S is a set of points from R^2, then the **convex hull** of S is the boundary of the smallest convex domain in R^2 containing S [9]. See Figures 3,4 for an example of a convex domain and a convex hull, respectively.

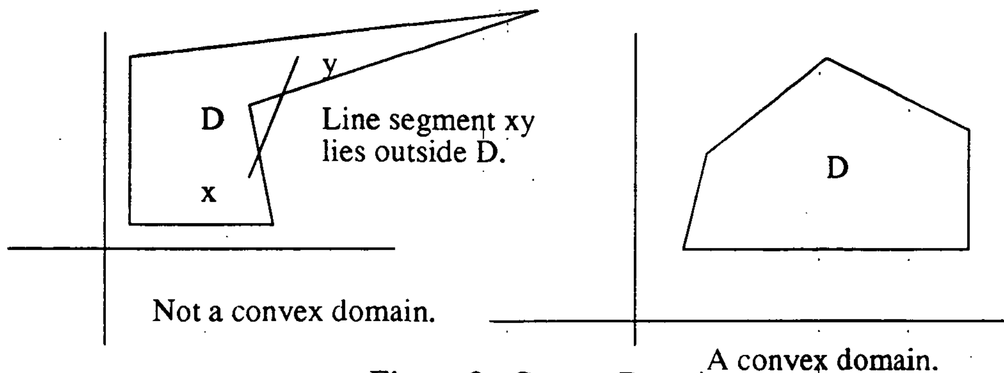


Figure 3 : Convex Domain.

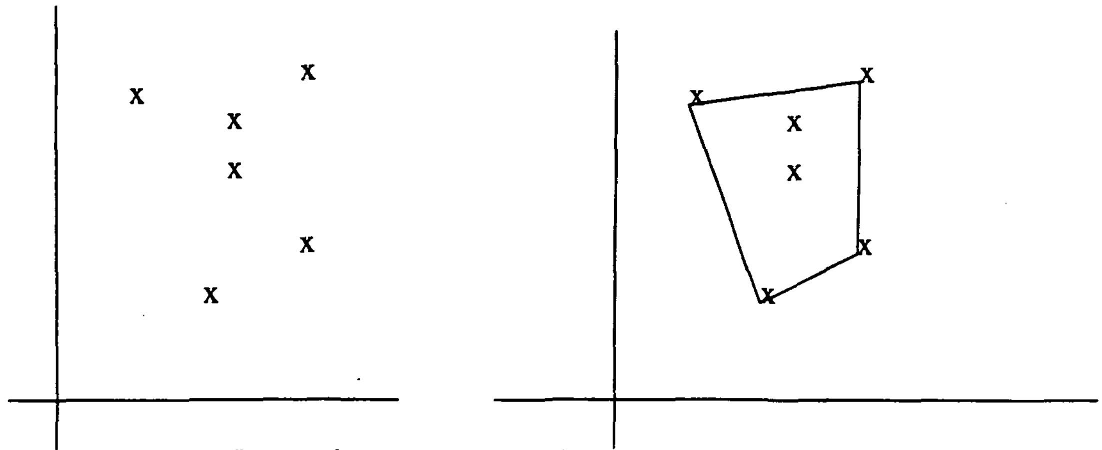


Figure 4 : The convex hull of the points on the left is on the right.

One can transform the set of points in {X, C} using a function F, the details of which are beyond the scope of this paper. So applying F to each element of {X, C} one has : $F(X_i, C_i) = (i, k)$ meaning that rule i belongs to group k, for every i belonging to [1,N].Hence, we have pairs of numbers that can be graphed in R^2 just like in Figure 4, but the difference here is that one does not construct the convex hull of the entire set, but one builds the convex hulls of each group individually. So, if after classification one finds that the N original rules fall in r (with $1 <= r <= m <= N$) distinct groups, then it must be the case that the r convex hulls that are produced are all **linearly separable** between them (refer to Figure 5 for an example).

Two sets of points are **linearly separable** if and only if there exists a hyperplane H that separates them [9]. Therefore, two sets are separable if there convex hulls are disjoint, and our classification is a success only if all the convex hulls that correspond to the groups that have been derived are disjoint between them.

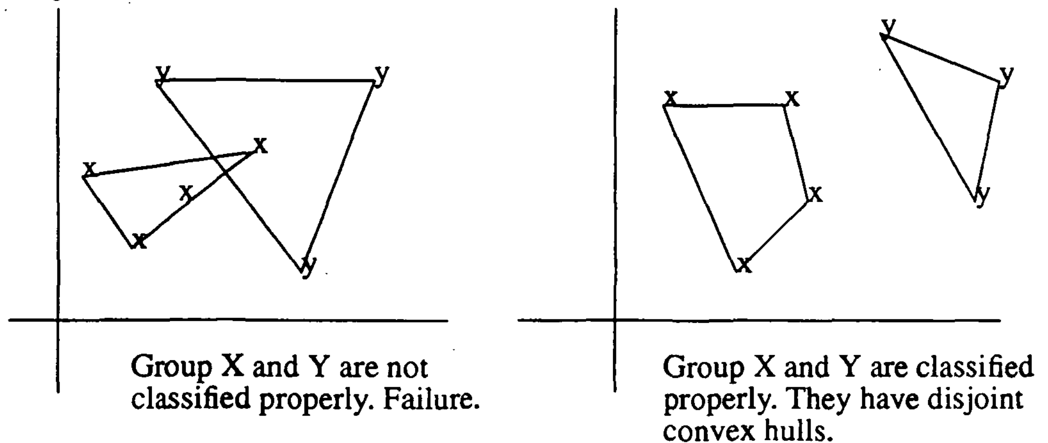


Figure 5 : Two separable sets on the right and two non-separable sets on the left.

NEAREST NEIGHBOR RULE

The basic idea behind this concept is given a rule "A" find its distance with all the other rules in the training set. From those distances one chooses the smallest, and classify rule A as being in the same group as the rule for which they have the smallest distance.

Let G = the number of samples in the training set. Therefore, there are N -G samples in the testing data set.The general algorithm general algorithm is shown below :

Assign K = G
CLASSIFY the training set
FOR every rule i in the testing set
  CALCULATE the distance of rule i with all other rules in the training set
  FIND the K smallest distances between rule i and every rule in the training set
  FOR every K-pair of rules (i,j) if rule j belongs to group A, then group A gets a vote
  FIND the group that has the most votes, say A". Then rule i is classified in A".

Many of these steps can be done together, but it's written out explicitly in order for one to understand the algorithm . As one can see by varying K, one can get any of the above methods. Particularly one may also group the rules into a testing set and a training set discretely, choosing possibly mutually exclusive sets. Finally, as mentioned earlier, by classifying a training set one simply results to the default, where each rule belongs in its own group, or the expert selects rules that might be pivotal to a derivation as being the head of a group.

In the experiments that will be discussed the U-method was used. If K = 1, then it's the pure method, otherwise by varying K one is able to classify even with a more unbiased attitude. The ideal value for K, as mentioned in many papers and proven empirically, is :

$$\sqrt{N}$$

ESTIMATION OF MISCLASSIFICATION

For each of the above methods a probability of misclassification will be derived. There exist many probabilities of error that are used in ones analysis. Here we will consider two, and the second one will be chosen for our derivations.

OPTIMAL/BAYES PROBABILTY OF ERROR : It is denoted by P^B_e and is given by :

$$1 - \oint MAXi \{P(\frac{X}{Ci}) P(Ci)\} dX$$

$P(X/C_i)$ and $P(C_i)$ are the class conditional probability density function and a priori probability of the ith group, respectively. The maximum value of the product is taken over all i, ie. over all possible groups of rules in the system. This error probability results when one has complete knowledge of the probability density functions with which to construct the optimal decision rule and uses the Bayes decision rule [6].

PROBABILITY OF ERROR : It is denoted by P_e and is the probability of error on future performance when the classifier is trained on the given data set. In practice one usually obtains a data set which is not only finite, but in fact quite small. Frequently no knowledge is available concerning the underlying distributions. In such cases one would like to know what the resulting probability of error is going to be on future pattern samples when the classifier is trained on the given data set. All the above methods train and then test the classification. Using this probability one will see how "good" the original classification was.

Let $P_e[R]$ denote the probability of error for the R-method using the Nearest Neighbor rule. If K denotes the number of incorrect classification attempts in step 2) of the process, then the resulting probability is K / N, where N is the number of rules in the system.

Let $P_e[H]$ denote the probability of error for the H-method using the Nearest Neighbor rule. At step 2) of the experiment let $P_e[H]_i$ be the number of errors found when attempting to test $\{X, C\}^i_b$ on the trained classifier $\{X, C\}^i_a$. With i varying from 1, ..., K , then the resulting probability of error is :

$$(1 / K) (P_e[H]_1 + ... + P_e[H]_K).$$

Let $P_e[U]$ denote the probability of error for the U-method using the Nearest Neighbor rule. During the entire process one has been tabulating the errors of misclassification when testing a sample pattern on the trained set (e_i). Therefore, the resulting probability of error is :

$$(1 / N) (P_e[U]_1 + ... + P_e[U]_N).$$

In spite of its advantages with regards to bias, the U-method suffers from at least two disadvantages. Although it is desirable to have the average of errors "close" to the actual error of probability, it is more important to use a method with a small variance. Having confidence about a particular classification of a data set might be preferred to being just unbiased. It has been shown that the U-method has much greater variance than the R-method, making it even more likely that it will be used less, compared to the simplicity of implementation of the R-method. Finally, the U-method requires excessive computation in the form of N training sessions, unless N is small.

EXPERIMENTAL RESULTS

Table 1: U-Method

K	ERRORS
6	2
12	5
24	13
36	24

The analysis of the automobile rule-base (auto.clp)
There are 36 rules. The Dcon metric for obvious reasons is used. From the rules, 3 are selected as group leaders. One rule summarizes diagnostics, the second rule captures data collection and the other possible solutions. One uses the Nearest Neighbor rule with the U-method. Various values for K are considered. Observe that the square root of N (ie. 36) is the best solution.

Errors indicate how many misclassifications were attempted. Simialr results are derived with other methods.

CONCLUSIONS

From the information presented it is evident that there is more information encapsulated in the system. Underlying subdomains are created with more semantics attached to them in terms of classification, validation of the semantics of the rule-base is achievable through convex hulls, the rule-base is now more semantically comprehensible, and problems are simpler to debug.

Initially one must select the appropriate distance metric, choose a suitable method of classification depending on the characteristics of the system, and apply the Nearest Neighbor rule to the data. At the same time errors at run time are collected in order to come up with a probability of misclassification.

As an extension to the scheme one might add saliences to each group derived, and to further enhance the system one might add saliences to each rule in a group. Another addition might include for the system to select on its own the best metric after examining the probabilities of error along with the data. It is obvious that this analysis can be executed when entering the CLIPS shell, but as one develops the rule base it is possible to classify one rule at a time, or gather newly defined rules and process them in batch mode. Finally, gathering information on how many times a rule is used, how many times a rule is used to initiate a derivation, and how many times a rule is used to conclude a derivation can prove to be very useful information in a more complex scheme of classifying a rule-based system.

ACKNOWLEDGEMENT

I would like to thank Bell-Northern Research Ltd. for the resources that were used in order for this research to be completed and published.

REFERENCES

[1] S. Anastasiadis (1990). CLIPS Enhanced with Objects, Backward Chaining, and Explanation Facilities. *First CLIPS Conference Proceedings,* Houston, pp. 621-641.

[2] B. Chandrasekharan (1986). Generic tasks in knowledge based reasoning : High-level building blocks for expert system design. *IEEE Expert, Fall 1986.*

[3] W.J. Clancey (1983). The advantages of abstract control knowledge in expert system design. *National Conference on Artificial Intelligence,* pp. 74-78, 1983.

[4] R. Duda, P Hart (1973). *Pattern classification and scene analysis,* New York : Wiley, 1973.

[5] W. Highleyman (1962). The design and analysis of pattern recognition experiments. *Bell System Tech. Journal,* vol. 41, pp. 723-744, Mar. 1962.

[6] R.V. Hogs, E.A. Tanis (1977). Bayesian Decision Theory. *Probability and Statistical Inference*, Macmillan.

[7] M.R. Genesereth, N.J. Nilsson (1987). Knowledge and Belief. *Logical Foundations of Artificial Intelligence,* Morgan Kaufmann.

[8] P. Lachenbruch (1967). An almost unbiased method for obtaining confidence intervals for the probability of misclassification in discriminant analysis. *Biometrics,* vol. 23 , pp. 639-645, Dec. 1967.

[9] F.P. Preparata, M.I. Shamos (1985). Convex Hulls : Basic Algorithms. Extensions and Applications. *Computational Geometry,* Springer-Verlag.

[10] S.L. Tanimoto (1987). Productions and Matching, Knowledge Representation. *The Elements of Artificial Intelligence,* Computer Science Press.

N92-16612

DEBUGGING EXPERT SYSTEMS USING A DYNAMICALLY CREATED HYPERTEXT NETWORK

Craig D. B. Boyle and John F. Schuette

Dept. of Computer Science
Texas A&M University
College Station, Texas 77843-3112
internet: craig@cs.tamu.edu

Abstract. The labor-intensive nature of expert system writing and debugging has motivated this study. Our hypothesis is that a hypertext based debugging tool is easier and faster than one traditional tool, the graphical execution trace.

HESDE (Hypertext Expert System Debugging Environment) uses Hypertext nodes and links to represent the objects and their relationships created during the execution of a rule based expert system. HESDE operates transparently on top of the CLIPS rule based system environment and is used during the knowledge-base debugging process. During the execution process HESDE builds an execution trace. Use of facts, rules, and their values are automatically stored in a Hypertext network for each execution cycle. After the execution process the knowledge engineer may access the Hypertext network and browse the network created. The network may be viewed in terms of rules, facts and values.

An experiment was conducted to compare HESDE with a graphical debugging environment. Subjects were given representative tasks. For speed and accuracy, in eight of the eleven tasks given to subjects HESDE was significantly better.

INTRODUCTION

Knowledge based systems are being increasingly used in diverse environments, writing such a system is no longer a research issue. However, the labor intensive and error-prone nature of their development has spurred researchers to examine development tools.

The hypothesis of this study is that a hypertext (Conklin, 1987; Nielsen, 1990) based debugging tool is more productive than one traditional tool, the graphical execution trace. In this context, the definition of a "productive" system is one that:

- is easy to learn and use,
- is fast to use,
- helps the knowledge engineer "understand" the expert system, and
- is relatively insensitive to changes in the particulars of the expert system's execution.

This definition of productivity is motivated by several factors. As with traditional programs, expert systems evolve over a potentially long lifetime. Over this lifetime, personnel not involved with the original development of the project will eventually be used to maintain the system. Their tools should make it relatively easy for the new knowledge engineer to grasp the processing of the system. Also,

apparently minor changes in an expert system may bring about drastic changes in the course of a consultation. Depending on the tool, such a change may make the debugging task harder for the knowledge engineer; it would be helpful if the debugging tool presents the knowledge engineer with a familiar landscape.

It is not hard to imagine a hypertext document that aids the knowledge engineer in his tasks. The objects of the expert system (facts, rules, etc.) could be represented by nodes. Links between the nodes could represent the causal relationships between rules and facts and the antecedent relationships between facts and rules. If such a network were built by an inference engine during a consultation, the knowledge engineer could browse the network afterwards to answer his questions about the session.

The research presented here investigates the potential role for hypertext as a tool to debug knowledge based expert systems. This discussion has several parts; first the expert system debugging task will be considered and differentiated from knowledge acquisition tools. This introduction to debugging and maintaining expert systems will provide some important insights into the characteristics of appropriate tools. For instance, what kinds of information does the knowledge engineer need when debugging his expert system? The traditional tool for monitoring an expert system's execution is the graphical execution trace. This trace is usually presented as a graph indicating the various relationships between specific facts and rules used in the consultation.

Previous research points to a need for effective expert system maintenance tools and hypertext is a candidate medium for those tools. The third section of this paper specifies the design of a novel hypertext network, named HESDE (Hypertext Expert System Debugging Environment), for debugging expert systems. This hypertext network is built automatically by the expert system's inference engine which is a modified version of the CLIPS engine (CLIPS, 1989). The network represents the important objects in the consultation via nodes and their interrelationships via links between the nodes. After the consultation halts, the knowledge engineer simply browses through the network to answer his debugging questions. The network provides links among facts, rules, agenda information and cycle number.

A critical question is whether this hypertext network is more useful to knowledge engineers than tools already in use. The fourth section describes an experiment conducted to evaluate that system's effectiveness as a debugging tool.

BACKGROUND

Significant expert systems continually evolve (Bachant et al, 1984; Smith 1984). Therefore, there should be a methodology and a set of tools for the maintenance and debugging of expert systems. Because of their non-procedural nature, however, expert systems debugging is not always amenable to the "snapshot" strategies utilized by debugging methods and tools commonly used for conventional programming languages. Different tools must be designed for the maintenance of expert systems.

The important questions to designers of these tools should be: What kinds of things do these tools have to do? What kinds of information do they have to convey? Traditional debugging tools tell us about the objects in a running program: instructions and data. Analogous tools for expert systems should do the same; they should tell us something about the objects in an expert system consultation: the agendas, rules, and facts. Neches et al have designed a system

that uses execution traces, domain knowledge, and knowledge about expert systems themselves to provide more useful explanations (Neches et al, 1985). They conclude that this "approach also offers other benefits related to development and maintenance." These benefits mostly are a result of the separation of the different types of knowledge (domain, expert system, etc.), but some benefit is also provided by the improved explanations. Because of this relationship between explanation and maintenance, it is useful to summarize the results of the research into different types of explanations.

There are various methods available to allow an expert system to explain its reasoning. These include simply having "canned text" in the system to handle expected questions, using a execution trace to deduce the pattern of reasoning, and building models of the problem domain to allow higher level explanations (Swartout, 1981; Swartout, 1983). Regardless of the method used, the designer must have an idea of what kinds of questions the user (or knowledge engineer) might ask.

Question Types

Swartout (Swartout, 1981) describes three types of questions that could be asked of an explanation system. The first can be called descriptive questions; they are asked to clarify the methods used by the system. The second type of question is the justification question: "Why did you conclude fact x?" In a debugging context, this question may take the form of "What rule set the value of fact x to y?" Except in trivial cases, this type of question is difficult to answer. Finally, Swartout's third type of question could be called a clarification question: "What do you mean by term x?". To answer these questions requires a significant amount of domain knowledge.

Answer Types

Gilbert (Gilbert, 1987) divides his answer types into three groups according to the source of the answer. These groups are the theory group, the domain group, and the case level group. The theory and domain groups correspond roughly to Swartout's clarification question type. Answers from the case level group deal with issues related to particular consultations and are therefore the most interesting to a debugging knowledge engineer. The case level answer types and their relevance to the debugging process include the following answer types:

An *instantiation answer* tells whether a certain fact exists in the knowledge base. Such information is commonly needed by a knowledge engineer to solve debugging problems (e.g., "I expected fact x to get asserted during the consultation. Did it?")

A *classification answer* is a decision made about some situation. Such an answer is usually the end product of the consultation and is obviously of interest to the knowledge engineer.

A *prescription answer* may be the by-product of a diagnosis expert system. They are typically generated by additional rules after the diagnosis (classification) is made. Again, these answers are of interest to the debugging knowledge engineer because they help indicate the visible correctness of the system.

A *justification answer* tells how some conclusion was reached. A closely related answer type is the *antecedent answer*; it describes events that lead to the conclusion. The answer to both question types can be found by examining a trace of the rules fired and facts asserted in the consultation. This is of great usefulness to the knowledge engineer because he can use these answers to verify that facts are asserted for the correct reasons, or can perhaps determine why some expected event did not occur.

HYPERTEXT

Hypertext dates back to Vannevar Bush's memex (Bush, 1945). Recent advances in personal workstation technology (particularly high-performance displays and windowing systems) have enabled hypertext to become a practical solution to many information management problems.

The Hypertext Expert System Debugging Environment – an Overview

The hypertext network introduced here is a new type of debugging tool; it is a Hypertext Expert System Debugging Environment (HESDE). In general, a HESDE is constructed automatically during the execution of some expert system's inference engine; the particular HESDE described in this paper is built by an forward chaining inference engine based on the engine in CLIPS. HESDE transparently records the execution of CLIPS, noting rule firings, fact instantiations and their interrelationships. After the execution of CLIPS, HESDE presents a browsable version of the execution environment to the user. Figure 1 shows the HESDE environment. Once CLIPS execution is complete the user may browse the network. The next sections will explain the functionality, structure and use of the network.

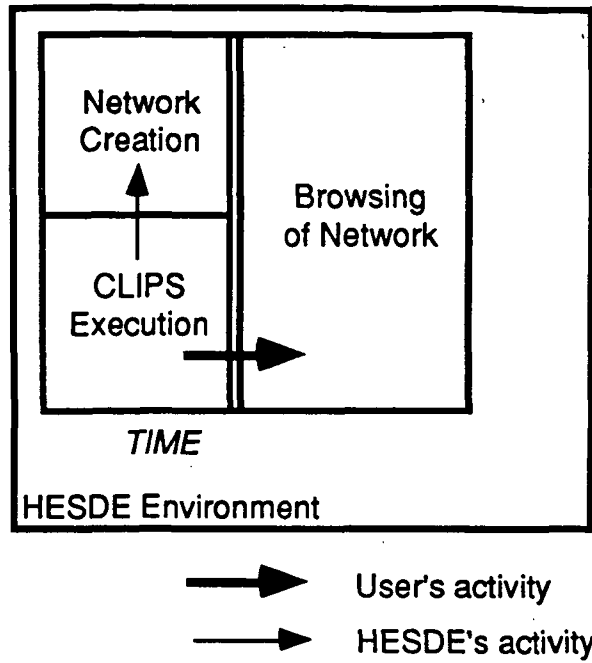


Figure 1. HESDE Environment

The knowledge engineer browses the HESDE network after execution of the expert system to find information about the objects and relationships created during that execution. HESDE is not just a snapshot of the system state at the end of execution, but a complete, cycle-by-cycle and inference-by-inference record of activity. Ideally, a HESDE would operate concurrently with the activity of the expert system, the present system only allows a post-execution analysis.

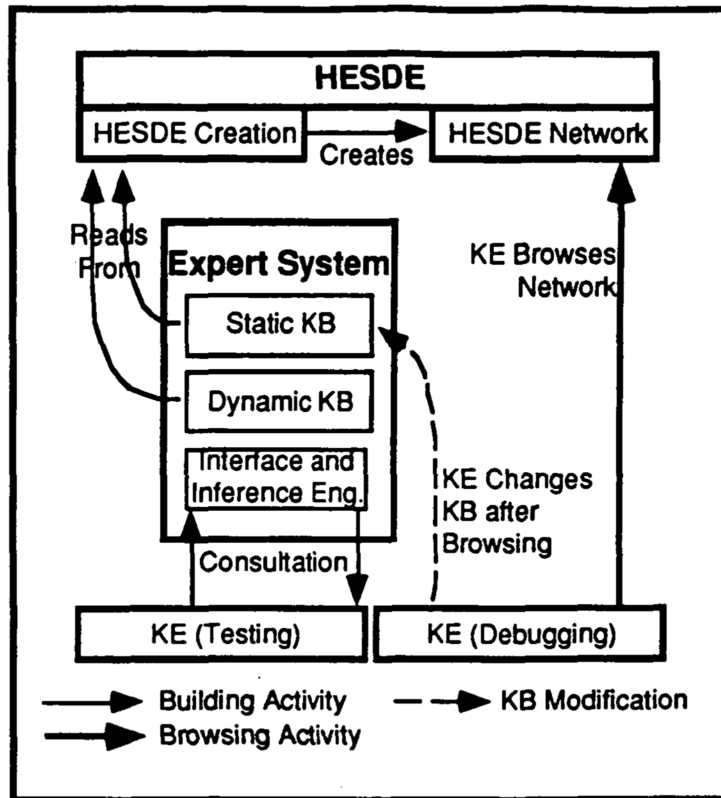


Figure 2. Overview of HESDE's activity

Hypertext model used by HESDE

A node in the HESDE system uses the card model used in hypertext systems such as NoteCards[1] (Xerox, 1985) and HyperCard[2] (Apple, 1990). In the HESDE systems, the node appears as a fixed-size window arbitrarily located on the screen. The card is smaller than the display, so multiple probably overlapping cards are visible simultaneously. The ability to display multiple cards simultaneously is important; in the HESDE system, it allows the knowledge engineer to leave one card on the screen while going to find another. If only one card were visible at a time, the knowledge engineer would be forced to remember the contents of the earlier card reducing the utility of the tool. Figure 3 shows a typical card.

---

[1]NoteCards is a trademark of Xerox Corporation.

[2]HyperCard is a trademark of Apple Computer, Inc.

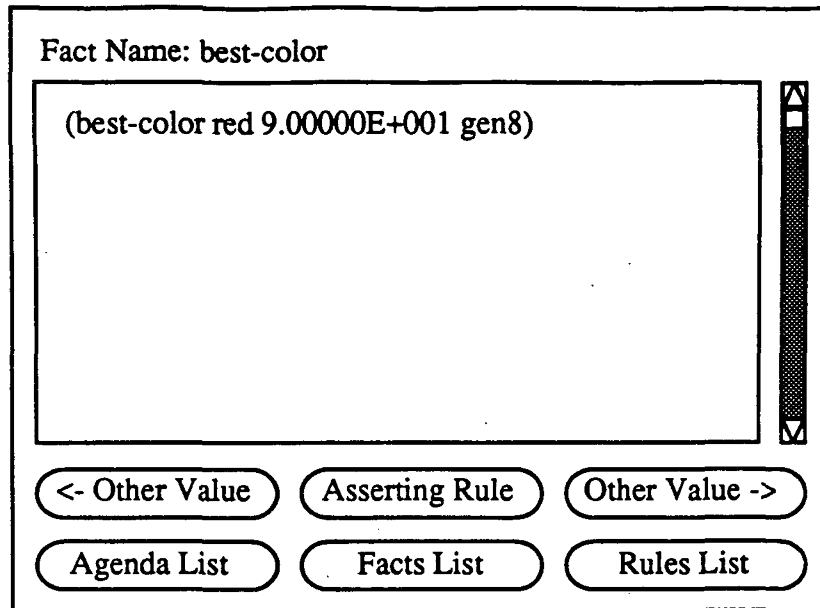


Figure 3. A typical card in HESDE[3]

Link Issues

All links are represented explicitly as buttons on the card. These buttons may appear "directly on" the card or as a part of the scrollable field. The HESDE system consist of a Hypertext browser and a Hypertext network creator. The network created by the HESDE is static and immutable. It has no capability for user-authoring–i.e. it is not (currently) possible for the user to change rules permanently or see the effect of fact changes on network status.

HESDE Network

We will now explain the structure and use of the HESDE network. Firstly an overview of the network structure and contents is presented. Secondly the three subnets and their interconnections are described in detail. Finally, the performance of the HESDE is briefly compared to that of a graphical execution trace.

Network Overview

The network created by HESDE is a directed graph containing several node types. The different node types are summarized in Table 1.

[3]The figures presented in this paper are schematic diagrams for clarity's sake.

Node type	Displays	Link targets
RIN	rule instance	master rule previous rule instance subsequent rule instance rule agenda antecedent facts
RMN	rule prototype	rule instance
RLN	list of rules	master rules
FIN	fact instance	previous fact instance subsequent fact instance asserting rule instance
FLN	list of facts	fact instances
AIN	rules on agenda	rule instances
ALN	list of agendas	agenda instances

Table 1. Node types and functions

The node types defined for the HESDE correspond to the basic objects in an expert system: the rules, facts, and agendas. Because this HESDE is based on CLIPS and because these are the only types of objects in CLIPS, the above list of node types is sufficient. New node types and relationships may have to be added to represent features supported by other expert system tools.

Subnets

Although the HESDE network is a single connected directed graph, it is convenient to describe it in terms of three subnets: the rules subnet, the facts subnet, and the agenda subnet. This division is made because at the highest level, the system presents the consultation as lists of the major conceptual objects in the consultation: rules, facts, and agendas.

These subnets each have a single node that serves as a "root" node for that subnet. In each subnet, this root node is essentially one of the lists mentioned above (rules, facts, or agendas). All nodes in the HESDE net contain links to each of these root or list nodes. This cross-linkage provides an efficient way for a browsing knowledge engineer to access a standard reference point in any particular subnet; getting lost is a common problem in large hypertext networks (Conklin, 1987).

The rules subnet

The first subnet in the HESDE net is the rules subnet. It consists of three types of nodes: Rule Master Nodes (RMN), Rule Instance Nodes (RIN), and a Rule List Node (RLN).

As the knowledge base is loaded, the system creates a RMN for each rule. At the outset, this node contains only the text of the rule in question. See Figure 4 for an example of an RMN.

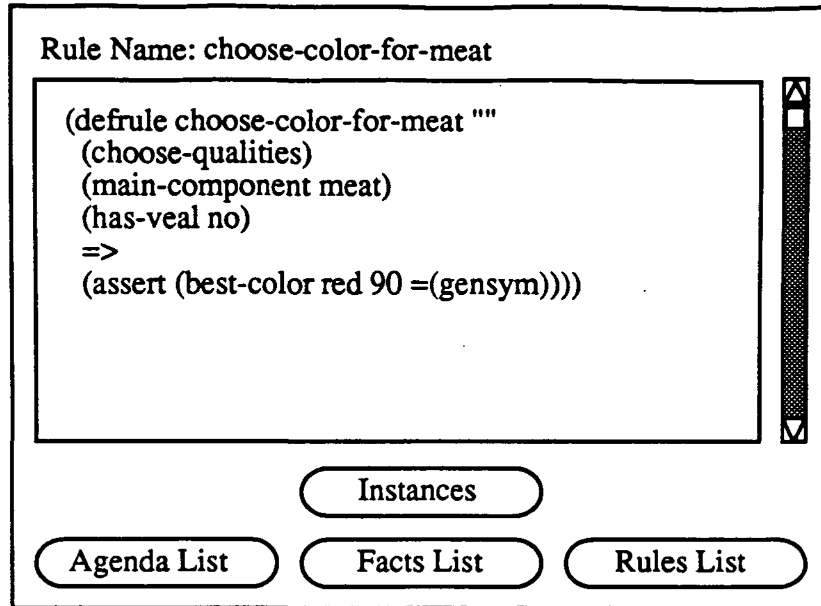


Figure 4. A typical RMN

Once the consultation is started, an RIN is created for every rule instance enrolled in an agenda. An RIN is similar in content to an RMN, but has some additional content. In particular, links to fact nodes are imbedded in appropriate places of the rule text. (The creation of HESDE nodes corresponding to facts in the knowledge base will be explained later.) These links are provided to allow the knowledge engineer to quickly determine the value of a fact bound to a particular rule. RIN's and RMN's are interconnected in the following manner; the first instantiation of a particular rule creates a new link on that rule's RMN that points to the first RIN for that rule. As new instances of that rule is created, the new RIN's are prepended to the chain of RIN's already connected to that RMN. This provides a path from the master rule to the instance rules. The reverse path is provided by a link in every RIN that points back to the RMN for the appropriate rule. This interconnection strategy is illustrated in Figure 5.

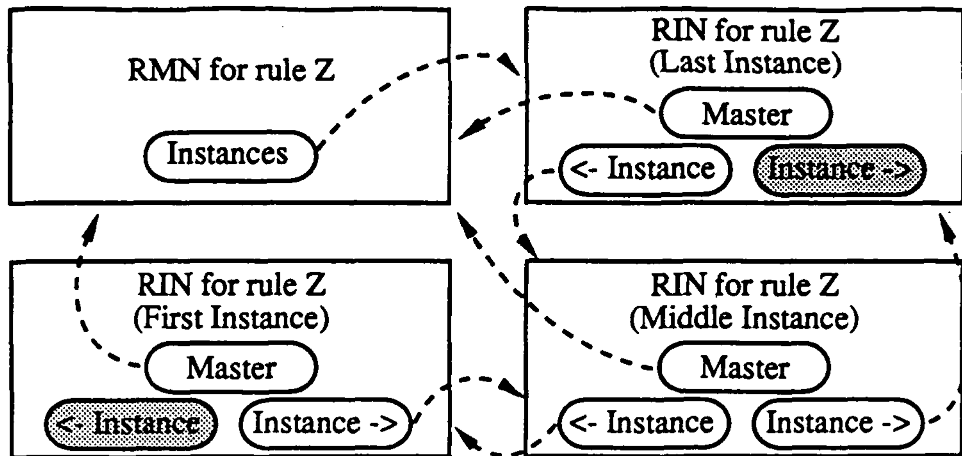


Figure 5. Some rule node interconnections

There are two reasons for providing both master and instance rule nodes. First, we need the RMN's because we can't rely solely on RIN's; it is common for a knowledge engineer to need information about a rule that was never instantiated. For instance, he may ask "why didn't rule x fire?" Without an RMN, there would be no evidence whatsoever of an uninstantiated rule. Second, the RIN's are necessary because the system needs to be able to keep track of the history of the consultation; any given rule may have multiple instantiations during the consultation (including simultaneous multiple instances), so a single node for each rule will not suffice.

A secondary use for RMN's in future systems may be to provide a logical point at which the knowledge engineer may edit the definition of a rule. The current HESDE system provides no editing capability, but such an extension would be a logical one.

The RLN is the last node type in the rules subnet. It serves as the root of the rules subnet and contains a link for every RMN created by the knowledge base loading process. These links are alphabetized according to rule name for ease in searching. As mentioned earlier, every node in the HESDE network has a link to the RLN to allow quick access to this reference point. Figure 6 shows a RLN and its links to other rule nodes in the network.

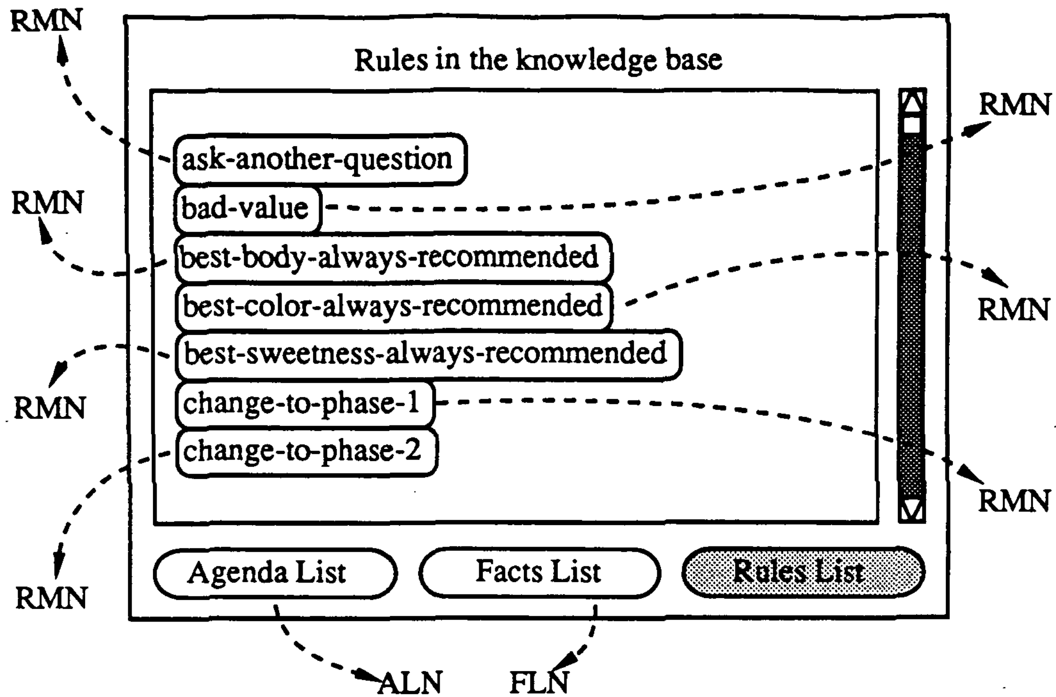


Figure 6. A RLN and its place in the network

The facts subnet

The second subnet is the facts subnet. There are just two types of nodes in the facts subnet: Fact Instance Nodes (FIN) and a Fact List Node (FLN).

A new FIN is created for every fact asserted during a consultation. Each FIN contains a printed representation of the fact instance in question. Before explaining the facts subnet further, it is important to define the HESDE's concept of a fact.

The interpretation attached to facts and their representations is clearly system dependent. For instance, facts in the KEE[4] system typically represent the characteristics of some object in a class hierarchy (Intellicorp, 1987). On the other hand, in CLIPS, facts are essentially just arbitrary lists of symbols (CLIPS, 1989). As will be detailed in the implementation section, the HESDE described here is based on the CLIPS expert system building tool. Therefore, the appropriate model of a fact is a list of symbols.

There is no actual semantic network connecting facts in a CLIPS knowledge base. Instead, the relations between facts are implicit and are enforced by conventional CLIPS programming style. In a CLIPS knowledge base, the first element of a fact's list representation usually names the fact. Any subsequent elements in the list are viewed as slots or values for the entity named in the first element. For example, the CLIPS facts (temperature 350) and (temperature 400)

---

[4]KEE is a registered trademark of Intellicorp, Inc.

may suggest two readings of a thermometer or two recommendations for a roasting temperature.

Therefore, using this fact model, the printed representation of a fact in a FIN is simply a display of that fact's list of elements. Figure 7 illustrates a typical FIN.

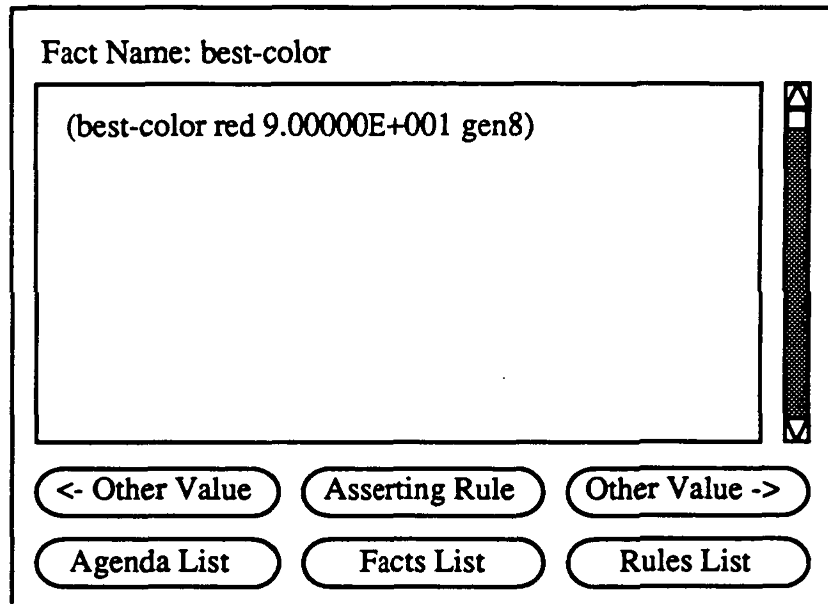


Figure 7. A FIN

Note that, in general, there may be more than one instance of the same fact in the knowledge base (i.e., more than one fact with the same name or first element). Whenever a new FIN is created, a check is made to see if there are any previous versions of the same fact in the knowledge base. If there are, two links are added to the facts subnet. First, a link is made from the new FIN to the pre-existing FIN. This link may be followed by the knowledge engineer to discover the "history" of a given fact. Second, a link is made from the older FIN to the new one; this link simply provides a return path to more recent versions of a fact.

As in the rules subnet, there is a root or list node used to provide direct access to any FIN in the facts subnet. This is the role of the FLN. After the first version of a fact is asserted and its FIN is created, a new link is added to the FLN. The target of this link is simply the newly created FIN. The buttons representing these links from the FLN are displayed in alphabetical order to allow the knowledge engineer to quickly find the FIN for a specific fact. Figure 8 shows an example of a FLN.

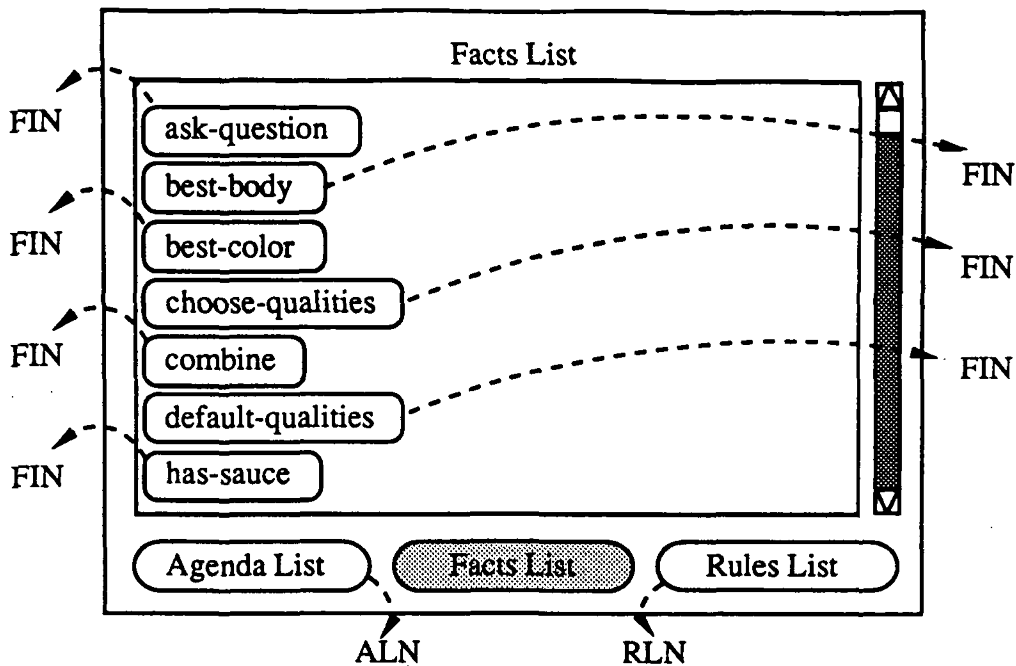


Figure 8. An example of a FLN

As discussed above, there may be more than one instance of a given fact at one time. The FLN lists only one link per fact name regardless of how many versions of that fact there may be. This restriction prevents the number of links from the FLN from growing too quickly. A fact link from the FLN always points to the most recently asserted version of that fact. If the knowledge engineer needs to examine a previous version, it is always available through the "previous version" links of the more recent versions of that fact. The method of interconnecting nodes in the fact subnet is illustrated in Figure 9.

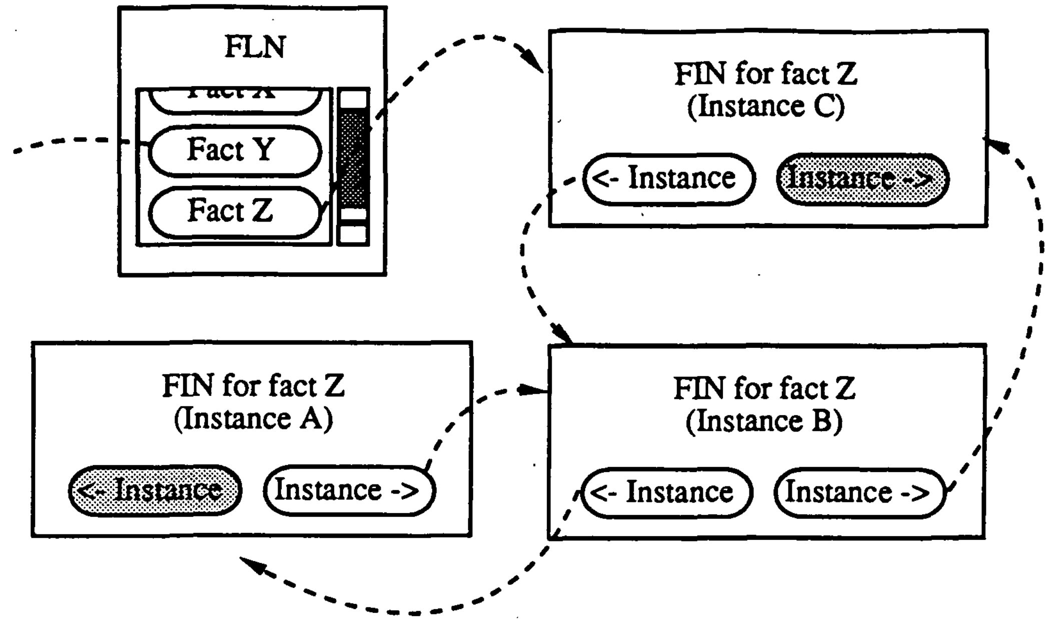


Figure 9. Fact interconnections

The agenda subnet

In an expert system, the agenda is the list of rule instances that are eligible to fire during a given cycle. Agendas are sometimes called conflict sets (Brownston et al, 1985). The agenda subnet is responsible for maintaining copies of all the agendas generated by the consultation. This information is valuable to the knowledge engineer because it helps provide the answer to questions such as "what rules were under consideration at point x and which one was fired?"

In the HESDE network, the agenda subnet consists of two types of nodes: Agenda Instance Nodes (AIN), and the Agenda List Node (ALN). In each recognize-act cycle of the inference engine, the system generates a new AIN for the current agenda. That AIN contains a link to every RIN that corresponds to a rule instance in the current agenda. Figure 10 illustrates such a node.

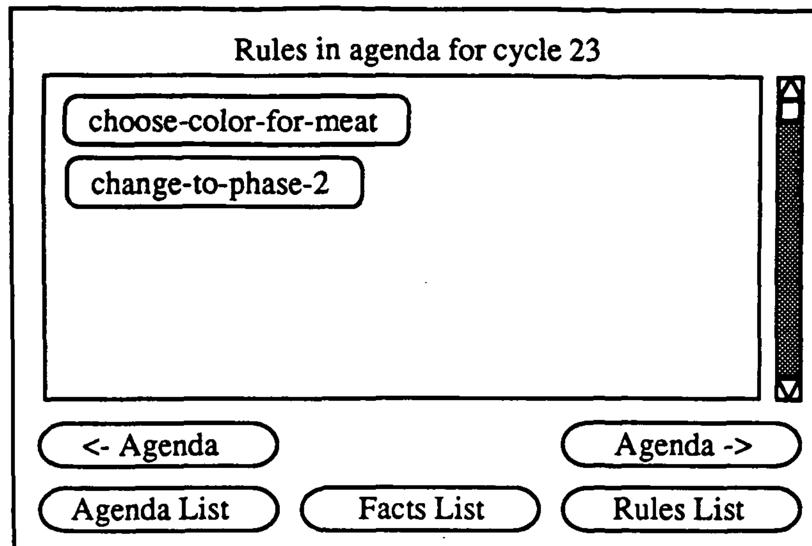


Figure 10. An AIN

So that all AIN's are accessible to the knowledge engineer, an ALN is created at the beginning of the consultation. As each new agenda is created by the inference engine, a link to its AIN is added to the links on the ALN. This ALN is presented as one of the roots of the hypertext network.

Subnet interconnections

Most links have their source and target in the same subnet. The most useful links in the network, however, are those that have their source and target in different subnets. This is because these links show the relationships between different types of objects: rules and facts, for instance. Now that all three subnets have been explained, these critical links can be described in detail.

The first class of subnet interconnection links are the agenda-rule links. As described above, they link AIN's to the RIN's corresponding to the rules in that agenda.

The next class of interconnection links are the rule-agenda links. In the description of the agenda subnet, the links from an AIN to a RIN were described. To provide the return link, in every RIN, there is a link from that RIN to the AIN corresponding to the agenda of which that particular rule instance is a part. This link allows the knowledge engineer to quickly determine of which agenda a particular rule instance is a member. Through the AIN, he can also determine what other rules were on the agenda at the same time as a particular rule instance. Figure 11 shows examples of agenda-rule and rule-agenda links.

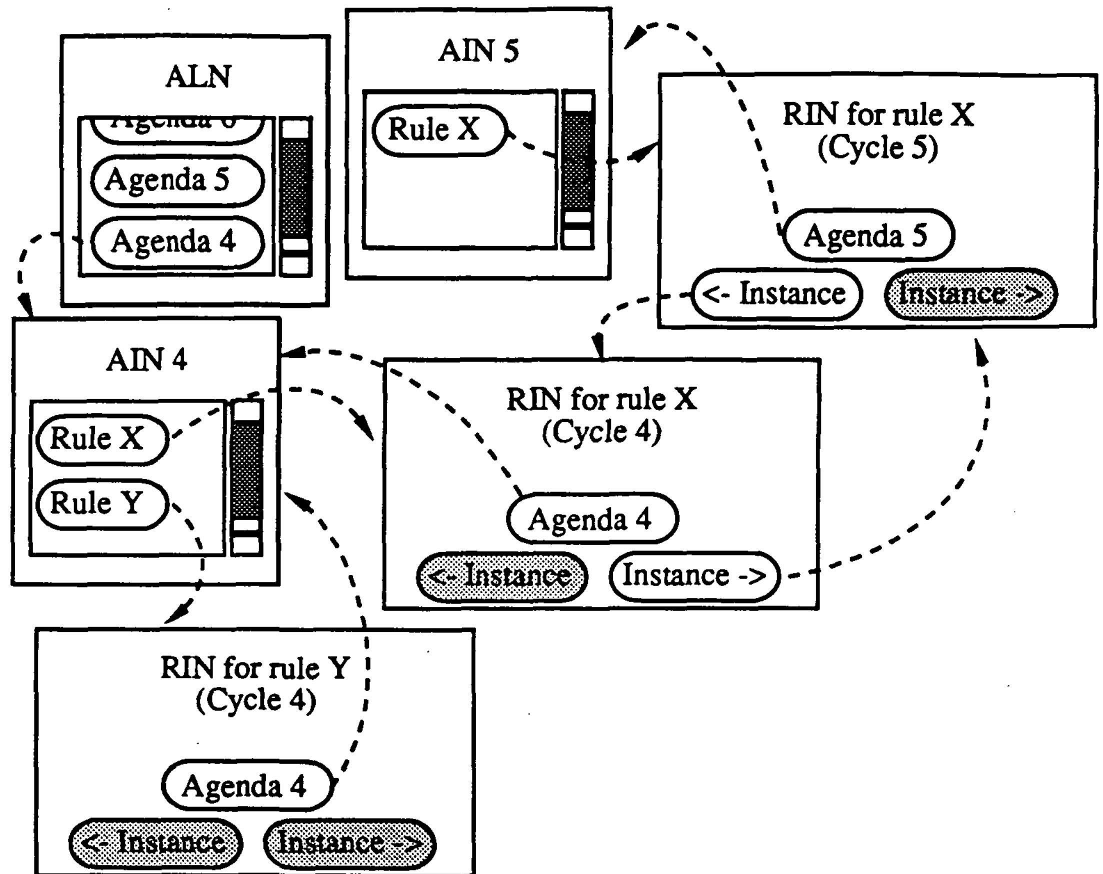


Figure 11. Agenda and rule interconnections

Another class of interconnection links is the fact-rule links. As each FIN is created, a link is established between that fact's node and the RIN of the rule that asserted that fact. By following such a link between a fact node and a rule node, a knowledge engineer can determine the rule responsible for a particular fact.

Finally, rule-fact links are the last class of interconnection links. These links allow the knowledge engineer to examine why a particular rule instance was eligible to fire. A rule instance is simply a binding of a rule with some set of facts that match the conditions named on the left hand side of that rule. As discussed above, each RIN corresponds to a specific rule instance and contains the source text of a particular rule. When a RIN is created, every mention of a fact in the left hand side of the rule is turned into a link to the FIN for that fact. By following these rule-fact links, the knowledge engineer can easily determine the value of that fact, what rule asserted it (through that FIN's fact-rule link), etc.

EVALUATION

To show the usefulness of HESDE, it was compared with a popular conventional tool. For comparison, we chose KEE (Intellicorp, 1987) which uses a graphical overview as the principal means of viewing a network. For

experimentation, six users were assigned to KEE and HESDE respectively. They were asked to simulate the debugging of a pre-defined expert system by performing a set of typical tasks.

The task list was compiled based on the kinds of information a debugging knowledge engineer might need. For instance, tasks 1, 4, 8, and 11 correspond to Gilbert's (Gilbert, 1987) justification answers. Tasks 6 and 10 are related to his antecedence answer type. The other tasks deal with the basic programming concepts: tasks 2 and 9 relate to control of rule firing; tasks 3, 5, and 7 just correspond to the knowledge engineer's examining the text of a rule.

Results

For eight of the eleven tasks HESDE proved to be statistically significantly better in terms of speed and accuracy. There was no significant difference in incidental learning between the subject groups.

Task completion time was measured as the time between clicks of the "go" and "stop" buttons with deductions made for the subject's looking back at the script. The deductions for re-reading the script were determined with a stopwatch during the experimental trial. In the case of the KEE subjects, additional deductions were made for machine response time. These deductions were determined using the video record of each subject's session. Figure 12 presents the mean task completion times by task.

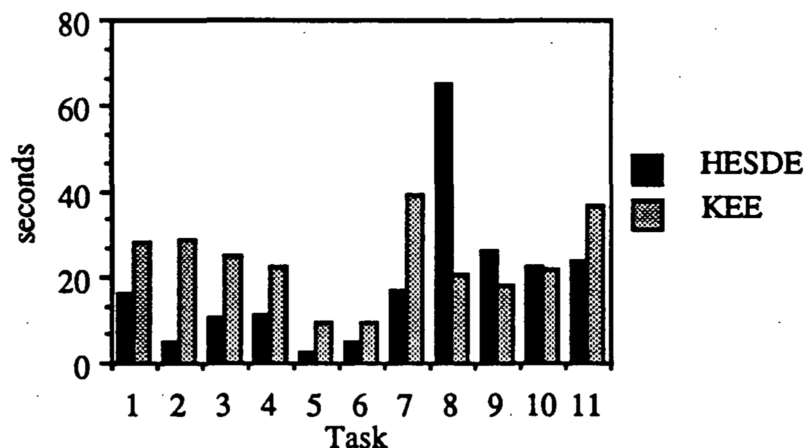


Figure 12. Graph of mean task completion times

These results indicate that the HESDE system is significantly faster ($p < 0.05$) for tasks 1-7 and marginally faster ($p < 0.10$) for task 11. KEE is significantly faster for task 8 and marginally faster for task 9. Task 10 seems to take the same time in both systems.

CONCLUSION

HESDE provides a robust and flexible hypertext-based environment for viewing and debugging an expert system's rule base. Testing has shown that HESDE outperforms at least one commercial expert system's debugging environment. We believe that HESDE will best be used as a complement to other debugging tools rather than a replacement.

What generalizations can be made from these results? For instance, to what extent are the observed differences due to weaknesses in the particular graphical trace studied here? This is unlikely; the problems with the KEE graphical trace are inherent to all graphical traces. These are the need to search through the trace space instead of directly accessing elements, the inability for the user to keep a previously visited location on the screen, and the sensitivity of the trace to changes in the consultation due to its spatial representation.

REFERENCES

Apple Computer, Inc. (1990) *HyperCard User's Guide*. Apple Computer, Inc., Cupertino, CA, 1990.

Bachant, J., and McDermott, J. (1984) R1 revisited: Four years in the trenches. *AI Magazine 5*, 3 (Fall 1984), 21-32.

Bush, V. (1945) As we may think. *Atlantic Monthly*, July 1945, 101-108.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985) *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, Mass., 1985.

CLIPS (1989) Artificial Intelligence Section, Johnson Space Center. *CLIPS Reference Manual*. NASA.

Conklin, J. (1987) Hypertext: An introduction and survey. *IEEE Computer 20*, 9 (September 1987), 17-41.

Gilbert, G. (1987) Question and answer types. Research and Development in Expert Systems IV; Moralee, D., ed., 1987.

IntelliCorp, Inc. (1987) *IntelliCorp KEE Software Development System Rulesystem3 Reference Manual.*. Intellicorp, Inc., 1987.

Neches, R., Swartout, W., and Moore, J. (1985) Explainable (and maintainable) expert systems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Volume 1, 1985, 382-398.

Nielsen, J. (1990) *Hypertext and Hypermedia*, Academic Press, 1990.

Smith, R. (1984) On the development of commercial expert systems. *AI Magazine 5*, 3 (Fall 1984), 61-73.

Swartout, W. (1981) Explaining and justifying expert consulting programs. In *Proceedings of the Seventh Annual Joint Conference on Artificial Intelligence*, August 1981, 815-822.

Swartout, W. (1983) XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence 21*, 3 (September 1983), 285-325.

Xerox Special Information Systems. (1985) *NoteCards Release 1.2k Reference Manual.*. Xerox Corporation, Pasadena, CA, 1985.

SESSION 8 B

PRECEDING PAGE BLANK NOT FILMED

N92-16613

IMPLEMENTING A FRAME REPRESENTATION IN CLIPS/COOL

Leonard Myers and James Snyder

CAD Research Unit, Design Institute
California Polytechnic State University, San Luis Obispo

Abstract. The purpose of this paper is to describe and evaluate an implementation of frames in COOL. The test case is a frame-based semantic network previously implemented in CLIPS Version 4.3 as part of the Intelligent Computer-Aided Design System (ICADS) and reported in the first CLIPS conference.

THE ICADS SYSTEM

The CAD Research Unit of the Design Institute at Cal Poly is engaged in a long term development of an Intelligent Computer Aided Design System (ICADS), (Pohl et al. 1989). Central to this work is the philosophy that the system should be a valuable assistant to the designer throughout the entire design activity. The current working ICADS prototype is a distributive system that can run with a variety of hardware processors in a Unix environment. It includes an Expert Design Advisor that has six knowledge based systems working as domain experts, a Blackboard Coordination Expert, two knowledge bases and several sources of reference data. The Expert Design Advisor in the prototype interprets a drawing as it is being made by a designer working in a CAD environment. It reacts in real-time to monitor the evolving floorplan from the viewpoints of experts in the domains of Access, Climate, Cost, Lighting, Sound, and Structure. The system also provides for additional interaction with the designer outside the CAD system to help realize a resultant design that is free from conflicts in the six areas listed.

The Frame Representation

All architectural information that is used for inference in the ICADS model, including the project specification information, is put into a specific frame representation (Assal and Myers 1990). The inferences are made by expert systems written in CLIPS (NASA 1990), an expert system shell, so the frame representation is designed particularly for the form of CLIPS facts. The frame implementation consists of representation, generation, and manipulation features beyond that which will be described here. However, the most important ideas can be easily seen.

Each frame will either hold a class or an instance of a class. If the frame holds a class, it will describe the basic characteristics of the class including: default values; demons that dynamically obtain values or perform tasks for instances of the class; the names of the slots in the class; and relations between the class and other classes.

Table 1 shows a simple declaration of an instance of a 'wall' class. The frame is merely a collection of CLIPS facts that satisfy certain rules of form. The number '21' in the

PRECEDING PAGE BLANK NOT FILMED

(FRAME	wall	21)			
(VALUE	wall	length	21	10	)
(VALUE	wall	orientation	21	180	)
(RELATION	wall	window	21	33	)
(VALUE	window	width	21	36	)
(VALUE	window	leads-to	21	16	)
(VALUE	space	name	16	EXTERNAL)	

Table 1. A Partial Instance of a 'wall' Class

example is an identifier for the particular instance of this wall class. The first slot shown holds the length of the particular wall. Another slot in the wall frame is denoted as 'orientation' and identifies the placement of the wall. Further, there is a relation to the 'window' class, which is interpreted to mean that a wall can have a window. As implied by these examples, the 'RELATION' fact is normally used to identify a 'has-a' relation. For example, '33' might be the instance identifier of a particular window that is in the wall. The wall also has the slots, 'width' and 'leads-to'. The final 'VALUE' fact shows that the window 'leads-to' the outside world, which is identified as 'EXTERNAL', through the name of the space instance referenced by '16', connecting the last two facts.

In addition to the above 'FRAME', 'VALUE', and 'EXTERNAL' keywords, there are also 'DEFAULT' and 'DEAMON' facts. The 'DEFAULT' fact can be used to establish values that are not set when an instance of a class is generated. For example, in the current ICADS system the ceiling-height slot of any instance of a space is 8 feet, by default. Finally, there is a 'DEAMON' that indicates actions that should be taken if certain conditions apply for the particular slot that is referenced. One of the options for the 'DEAMON' is 'if-needed'. This particular option has been used to generate the perimeter of a space whenever a space is created. Note that the difference between the 'DEFAULT' and DEAMON' functions is one of complexity. The 'DEFAULT' can only identify a constant to be used for a value. The 'DEAMON' can cause arbitrarily complex calculation to determine a value.

The instances of classes are used to represent architectural objects. For example, the first drawing action of the designer might be to sketch a rectangle which is then labeled 'office'. As the lines that make the four walls for this space are drawn, the system generates facts that describe what is represented by the drawing, in terms of the frames that have been defined for architectural objects. When the connection of the four walls, '21', '42', '35', and '27' is completed, the system will determine that a 'space' has been created and the instance for a 'space' frame would have been generated as seen in Table 2.

(FRAME	space	243)			
(VALUE	space	name	243	office	)
(VALUE	space	floor-height	243	1	)
(VALUE	space	ceiling-height	243	8	)
(VALUE	space	area	243	160	)
(RELATION	space	wall	243	21	)
(RELATION	space	wall	243	42	)
(RELATION	space	wall	243	35	)
(RELATION	space	wall	243	27	)
(RELATION	space	symbol	243	22	)
(VALUE	space	perimeter	243	52	)

Table 2. Modification of a 'space' Instance

The Semantic Network

The frames used to represent architectural objects are referred to as design object frames and the collection of all such frames used by the Expert Design Advisor is called the Semantic Network. Thus, it is the values of the design object frames that represent the current state of the design solution within the context of the project.

As the designer draws within the CAD system, additional frames are automatically and transparently generated by the system to describe the architectural objects being represented and the relations between them. The frame information is generated in a blackboard environment to which the domain experts and the coordination expert are connected. As a result, the rules in any domain expert can fire as soon as their requisite facts are posted to the blackboard. This in turn might result in the posting of advice, in the form of new frame information, that could instantiate rules within the coordination expert.

Geometric Design Objects

The architectural objects currently represented by the system include the following seven geometric design objects:

FLOOR, SPACE, WALL, DOOR, WINDOW, SEGMENT, and SYMBOL.

The FLOOR object refers to the level of a building, such as the first or second "floor". The SPACE object refers to an area, such as a room or duct. WALL, DOOR, and WINDOW have the obvious reference values.

The SEGMENT object refers to any part of a WALL object that is demarcated either by the intersection of another wall or has been drawn by the designer as a distinct wall component. The SYMBOL object represents directly by name any closed shape or icon within a SPACE object (eg., column, chair, table).

Since the focus of the ICADS system is on these design objects, it is natural to evaluate the use of COOL, the CLIPS Object Oriented Language (NASA 1991), to provide an "object" representation as a replacement for the frame representation that is in current use.

The Blackboard Coordination Expert

The principal purpose of the Blackboard Coordination Expert is to assert frame slots, representing the current state of the evaluation process performed by the domain experts, onto the Semantic Network resident in the Blackboard. To accomplish this, the Coordination Expert receives from the Message Router all of the frames which contain results generated by the domain experts. The values that identify the current state of the design fall into one of three basic categories: values which result from solutions proposed by a single domain expert; values which result from solutions proposed by several domain experts for a common current value; and, values which must be inferred from solutions proposed by several domain experts.

In the case of the first category, which represents solution values unique to a single domain expert, the Coordination Expert does not change the values proposed by the domain expert. The proposed solution values are simply asserted as current values into the appropriate frame slots. In the second category two or more domain experts propose differing values for the same solution parameter. In such direct conflict situations it is the responsibility of the Coordination Expert to either determine which of the values is most correct or to derive a compromise value.

The Coordination Expert incorporates resolution rule sets which determine the best current values from those proposed. There is a resolution rule set for each possible direct conflict. In the development of each rule an attempt has been made to achieve a desirable balance between the various design issues. At this level the Coordination Expert can be considered an expert with knowledge of how this balanced integration can be achieved.

For example, if the structural, thermal, and sound domain experts agree in their suggestions for the material to be used for a wall, there is a rule in the Coordination Expert that will assign the common value as the wall's material in the solution value, which is referred to as the 'current value', or 'Blackboard value'.

With the frame representation that rule is written as shown in Table 3.

```
(defrule m2-floor-external-wall-material-weight-rule-1
    (FRAME        floor    ?floor )
    (RELATION     floor    appearance/image    ?floor   ?floor-rel-struct          )
    (FRAME        appearance/image    source-idt   ?floor-rel-struct               )
    (VALUE        appearance/image    source-idt   ?floor-rel-struct     struct    )
    (VALUE        appearance/image    external-wall-material-weight
                  ?floor-rel-struct   ?material)

    (RELATION     floor    appearance/image    ?floor   ?floor-rel-therm           )
    (FRAME        appearance/image    source-idt   ?floor-rel-therm                )
    (VALUE        appearance/image    source-idt   ?floor-rel-struct     thermal   )
    (VALUE        appearance/image    external-wall-material-weight
                  ?floor-rel-thermal  ?material)

    (RELATION     floor    appearance/image    ?floor   ?floor-rel-sound           )
    (FRAME        appearance/image    source-idt   ?floor-rel-sound                )
    (VALUE        appearance/image    source-idt   ?floor-rel-sound      sound     )
    (VALUE        appearance/image    external-wall-material-weight
                  ?floor-rel-sound    ?material)

    (RELATION     floor    appearance/image    ?floor   ?floor-rel-BB              )
    (FRAME        appearance/image    source-idt   ?floor-rel-BB                   )
    (VALUE        appearance/image    source-idt   ?floor-rel-BB    )
    (VALUE        appearance/image    external-wall-material-weight
                  ?floor-rel-BB    ~?material)

=>

    ( bb_assert
        (MODIFY VALUE   appearance/image    external-wall-material-weight
            ?floor-rel-BB    ?material ) )
    (bb_end_message)
)
```

Table 3. A Coordination Rule With Frames

In particular, note that only the single "VALUE" fact for the material-weight slot, or fact, is changed. The rest of the facts that are used to make the frame representation for the appearance/image frame remain the same. In addition, there is no notion of "access" to the frame in order to make this change.

The alert reader might wonder about the following line, that appears as the second pattern from the bottom of the left hand side:

```
(VALUE        appearance/image    source-idt    ?floor-rel-BB    )
```

It might seem that the 'source-idt' value is missing. Well, it is! As a matter of history, a 'Blackboard value', or 'current value' does not have a source identified. It is simply understood to be the Coordination Expert.

A COOL Approach

The purpose of the frame representation scheme is to allow the ICADS system to reflect an understanding of architectural objects that are known to the user. It seems natural that an object oriented language such as COOL should be able to provide the same type of representation facility in an equal or better manner. However, the notion of "object" is notorious for harboring a wealth of misunderstanding and misuse.

Before attempting a large scale replacement of the thousands of lines of code that use the frame representation in an ICADS system, a careful evaluation must be made. In addition to the use of frames in domain expert systems and the coordination expert the frame scheme is used for utility purposes, such as the recognition and distribution of information in the Message Router module of the Blackboard (Taylor and Myers 1990). The Message Router uses pattern matching to identify the frame information that is distributed to the units that connect to the Blackboard. When a unit, such as a domain expert, connects to the Blackboard it provides a list of the frame information it wishes to receive. Since most units that connect to the Blackboard need only a small fraction of the facts received by the Blackboard, this selective distribution significantly reduces the amount of information communicated over the network.

Since COOL does not yet support pattern matching over objects, the impact of replacing frames in their CLIPS fact form with COOL objects must also be examined with respect to the communication uses of frame information, as well. First, however, an examination of the most common uses of the frame information is made.

Sample COOL Objects

For example, in Table 3 there are two type of frames, the floor frame and the appearance/ image frame. Each instance of such frames can be represented in COOL by the instance of an object that is used to represent the same architectural objects as the frames were intended to encode. A class can be defined for each of these architectural object types and a COOL object can be made correspondingly for each instance of an example frame representation.

For example, the object for the appearance/image frame might be defined, partially as:

```
(defclass    (APPEARANCE/IMAGE     (is-a USER)
    (slot    (FLOOR-ID))
    (slot    (SOURCE-IDT))
    (slot    (EXTERNAL-WALL-MATERIAL-WEIGHT)
)
```

Then if ?floor held the name of a particular FLOOR object, the following could create an APPEARANCE/IMAGE object to record the suggestion of 'heavy' for the external-wall-material-weight that is being made by the structural domain expert:

```
(assert  (appearance/image
              =(make-instance (gensym) of APPEARANCE/IMAGE
                         (FLOOR-ID      ?floor)
                         (SOURCE-IDT  struct)
                         (EXTERNAL-WALL-MATERIAL-WEIGHT       heavy)
         )
)
```

This assertion would generate a fact of the form, (appearance/image genx), where 'x'

is some integer and names the object made by the 'make-instance'. This fact can be matched by patterns in particular of the form, (appearance/image, ?v), to provide a convenient way of finding the instance names of the appearance/image objects in the left hand side of CLIPS rules.

Furthermore, the variable in the above pattern can be constrained by the attributes of the objects so that only certain objects are remembered by the variable. For instance, the form might be: (appearance/image, ?v&:(eq (send ?v get-SOURCE-IDT) struct))
In this case, only the objects that are recommended by the structural expert, that is, which have 'struct' as the value in the SOURCE-IDT slot would be matched.

Unlike the frame representation, the objects that hold the 'current values', 'Blackboard values', will have their SOURCE-IDT slots set to 'Coord'. Thus, with the appropriate definitions of the classes needed, the rule shown in its frame representation form in Table 3 could be used with COOL objects for the same purpose as below:

```
(defrule m2-floor-external-wall-material-weight-rule-1
    (appearance/image   ?BB&:(eq  (send ?BB get-SOURCE-IDT)   Coord) )
    (appearance/image   ?floor1&:(and (eq ( send ?BB  get-FLOOR-ID)
                                              (send ?floor1  get-FLOOR-ID))
                                      (eq (send ?floor1 get-SOURCE-IDT) struct ))

    (appearance/image   ?floor2&:(and (eq ( send ?BB  get-FLOOR-ID)
                                              (send ?floor2  get-FLOOR-ID))
                                      (eq (send ?floor2  get-SOURCE-IDT) thermal ))

    (appearance/image   ?floor3&:(and (eq ( send ?BB  get-FLOOR-ID)
                                              (send ?floor3  get-FLOOR-ID))
                                      (eq (send ?floor3  get-SOURCE-IDT) sound))

    (test  (eq   (send ?floor1  get-EXTERNAL-WALL-MATERIAL-WEIGHT)
                 (send ?floor2  get-EXTERNAL-WALL-MATERIAL-WEIGHT)
                 (send ?floor3  get-EXTERNAL-WALL-MATERIAL-WEIGHT) ) )

    (test  (neq (send ?floor1  get-EXTERNAL-WALL-MATERIAL-WEIGHT)
                (send ?BB  get-EXTERNAL-WALL-MATERIAL-WEIGHT) ) )
=>
    (bind ?wt (send ?floor1  get-EXTERNAL-WALL-MATERIAL-WEIGHT) )
    (bb-assert (MODIFY ?BB  EXTERNAL-WALL-MATERIAL-WEIGHT  ?wt ) )
    (bb-end-message)
)
```

Table 4. The Rule from Table 3, With Objects

This form of the rule sends messages to the appearance/image objects that make suggestions for the same object instance of the floor class, by checking the FLOOR-ID slot of the objects to make sure they are the same. It further sends messages to determine that the three objects whose instance names are held in ?floor1, ?floor2, and ?floor3 are for objects that hold the suggestions from the structural, thermal, and sound domain experts, respectively. Then it checks to make certain that the suggestions for EXTERNAL-WALL-MATERIAL-WEIGHT) from these three objects are the same value. Finally, it determines if the current Blackboard value for this architectural attribute, as held in the ?BB object, is different from the common value in the three domain objects.

For the action part of the rule, consideration must be given to the manner in which the distributed Blackboard works. The Blackboard objects are replicated in the appropriate units

that attach to it. Therefore the action of asserting a result from the coordination unit is effected by a message that is sent from the Blackboard to all units that use the information being posted. Generally the message action is to ADD, MODIFY, or DELETE a fact.

Distributed Blackboard Action

In Table 3 the MODIFY message will be interpreted by each unit that receives it as a command to change the fact that holds the external-wall-material-weight slot value in the instance of the appearance/image frame identified by the value of the ?floor-rel-BB value to become the value of the ?material variable. The unit receiving the message simply retracts the current fact of this form and asserts the new one. Note that it does not change any of the rest of the facts that identify the frame.

In the COOL version of the rule shown in Table 4, an equivalent distributed effect can be achieved by having the objects that hold the Blackboard information, the information asserted by the Coordination Expert, duplicated in the other units by objects that have the same instance name. Thus in the rule in Table 4, the same instance name is being sent to all of the units. The MODIFY function in each receiving unit will then perform the following:

```
(send b-string  put-EXTERNAL-WALL-MATERIAL-WEIGHT m-string )
```

where b-string is the value of ?BB and m-string is the value of ?wt in the message that is transmitted by the bb-assert function

Refreshing Rules

One other consideration must be given to the object scheme. When a slot in an object changes, the rules that might use that slot are unaware of the change. There is no mechanism in CLIPS/COOL that automatically refreshes rules that reference slots in objects, when the slots change their values. Therefore, the action of changing a slot value, as shown in the above paragraph, is accompanied by a retraction and assertion of the associated CLIPS fact for the instance of the object, to make certain that all rules that reference the object will be readied to fire again.

CONCLUSION

The current working model of the ICADS system has the ability to advise a designer in a real-time design activity. It also exhibits a considerable amount of knowledge about the non-geometric attributes of the objects drawn and the real environment for the design project.

The most difficult part of extending the system is the classic problem of knowledge acquisition, for the domain expert systems and even moreso, for the coordination expert. One of the current problems is the difficulty of maintaining the exact form for referencing the same architectural object in references developed by different people at different times in different modules of a highly distributed system. A number of investigations to helping reduce this problem are in process in the ICADS laboratory. However, it seems that a great deal of help could be provided by the use of a representation for the architectural objects that better encapsulates the architectural object information into the expert system environment than the current frame representation protocol. COOL objects provide this improvement.

The use of COOL objects can also provide some speedup in the execution of certain computation. For example, in the current ICADS system the perimeter of a space is determined from the wall length information that is held in the frames for the walls that make up the space. When a wall is added in the drawing, pattern matching is used to identify the space to which the wall belongs. Then the old area fact is retracted and the updated area value is asserted in a new fact. By keeping the instance name of the space in each wall object, a method can be used to update the area slot in the space object whenever a change in

the wall object occurs. This particular action reduces both the pattern matching and retraction operations, which are fairly expensive activities.

In addition, it is easier and more efficient to implement in COOL objects the 'DEFAULT' and 'DEAMON' features from the current frame-based representation.

On the other hand, it can be seen by the example of Table 4 that the rules that reference the objects do not have an efficient way of getting the information from objects so that it can be used in the pattern matching process of their left hand sides. Performance evaluations on equivalent uses of larger scale will have to be done to identify whether a movement of the entire ICADS system to COOL is justified by the esthetic and maintenance advantages of working with COOL objects over working with the current frame-based representation.

REFERENCES

Assal, H. and Myers, L. (1990). An implementation of a frame-based representation in CLIPS, *First CLIPS Conference Proceedings*, Houston, pp.570-580.

Myers, L. and Pohl, J. (1989). ICADS DEMO1: a prototype working model, *Fourth Eurographics Workshop on Intelligent CAD Systems*, Paris, France, pp.33-71.

NASA (1989). *CLIPS Architecture Manual (version 4.3)*, Artificial Intelligence Center, Lyndon B. Johnson Space Center, Houston.

NASA (1991). *CLIPS Architecture Manual (version 5.0)*, Artificial Intelligence Center, Lyndon B. Johnson Space Center, Houston.

Pohl,J., L. Myers, A. Chapman and J. Cotton (1989). *ICADS: Working Model Version 1*, Technical Report: CADRU-03-89, CAD Research Unit, Design Institute, School of Architecture and Environmental Design, California Polytechnic State University, San Luis Obispo, California.

Taylor, J. and Myers, L. (1990). Executing CLIPS expert systems in a distributed environment, *First CLIPS Conference Proceedings*, Houston, pp.686-697.

N92-16614

APPLICATION OF A RULE-BASED KNOWLEDGE SYSTEM USING CLIPS FOR THE TAXONOMY OF SELECTED *OPUNTIA* SPECIES

Bart C. Heymans, Joel P. Onema, Joseph O. Kuti

Horticulture Research Laboratory, College of Agriculture
Texas A&I University, Kingsville, Texas 78363

Abstract. A rule-based knowledge system was developed in CLIPS (C-Language Integrated Production System) for identifying *Opuntia* species in the family Cactaceae, which contains approximately 1,500 different species. This botanist expert tool system is capable of identifying selected *Opuntia* plants from the family level down to the species level when given some basic characteristics of the plants. Many *Opuntia* species are cultivated as ornamental plants and some are significant as food crops. *Opuntia* plants are becoming of increasing importance because of their nutrition and human health potential especially in the treatment of diabetes mellitus. The expert tool system described in this paper can be extremely useful in an unequivocal identification of many useful *Opuntia* species.

INTRODUCTION

Rule-based expert tool systems have been implemented in a variety of applications (NASA 1989). One of the better suited applications of the tool systems is in botany or plant sciences, where they can be used to effect plant identification and taxonomic search. Plant taxonomy is particularly suited for the tool systems because it usually starts out from empirical characteristics, which are sometimes confusing and/or fuzzy in nature, until it narrows down the characteristics to a few but concrete ones for the eventual identification of particular plant species and their botanical synonyms, which are common to all scientific nomenclatures.

This particular tool systems may start out from plant kingdom and end in the eventual identification of the plant species. The overall taxonomic structure obtained constitutes a tree-like hierarchy [fig. 1] in the leaves represents individual species. This hierarchy will be covered in some details later. The CLIPS language, developed by NASA has been chosen for this system because of its portability, flexibility and the capability of its integration with other languages e.g. the C language (NASA 1990).

SYSTEM DESCRIPTION

A general overview of the system is shown in a directed graph [fig. 2]. The system is capable of identifying selected *Opuntia* and *Nopalea* species (Britton and Rose 1963) when adequate information are given on the basic characteristics that distinguish these species from each other (Buxbaum 1950). The system also enables the user to start species identification from any taxonomic level. For example, if the plant family is known, identification may start at this level, or if the plant series and forms are known, these will

become the starting points of the taxonomic search. If on the other hand, no plant speci can be identified, the user has the option to continue the search, to quit or to start all ov again.

To build the tool system we opted for a modular approach. [fig. 4] The syste consists of different knowledge segments modularly linked together as one logical knowled base. Each knowledge segment could be considered as having a salience associated with so that it can only be accessed in a predetermined order. As identification proceeds, item (e.g. in the plant Order, Class, Series, etc...) must be identified and loaded in a pare segment or a subsidiary segment provided the item in question does not constitute the e of the search. Smaller subunits can be loaded from the subsidiary segment when they ha been duly selected. Thus, by means of this modular approach, a cascading effect will occ and plant characteristics selected by the user may fire the necessary rules in the starti segment. These rules will then propagate through intermediary segments and will eventual terminate in the identification of the plant species in one of the leaf segments.
Currently, the functioning modules in the system include a root or driver segment th contains the rules for the plant taxa, several intermediary modules [fig. 5] and the le segments which form core of the program.

The *Opuntia* and *Nopalea* species in the Cactaceae family [fig. 1], as described Britton and Rose (1963) and Pizzeti (1985), were chosen as model for the expert to system. Selected plants belong to the tribe Opuntiae, the the family Cactaceae and the ord Cactales [fig. 1]. In the tribe we identify plants in the genera *Nopalea* and *Opuntia*. Th include species in the series *Dillenianae*, *Streptacanthae*, *Polyacanthae*, *Ficus-indicae*, ar *Robustae*, in the subgenus Platyopuntia and in the genus *Opuntia*. The system has addition capabilities which include the possibility of keeping track of multitude and often confusi botanical synonyms that are common with plant taxonomy. Our system is adaptable to regular updating of the knowledge segment so that the system can display all the botanic synonyms, if so desired. Our system can also give the common name in any or all of fi selected foreign languages (if a common name is available in that language).

The system is menu driven in which the user is presented interactively with a choi between predefined plant characteristics, hierarchically structured [fig. 3] and demonstratic option. Since we opted for a modular approach the system agenda is kept to the minimu so that fewer rules can be fired at a time. In consequence, the system closely mimics t taxonomic categories or taxa used in plant and animal systematics. This allows for quick implementations of new modules from the existing (i.e. not yet implemented) or undefin (i.e. previously not defined in the taxonomy) items, easier merger of the two modules if t two items can be unified taxonomically and deletion of the obsolete segments especial when an item loses its independent status. When a new botanical name is used to identi a plant, the name must only be changed inside its particular segment since its scope limited to that segment. For example when the name Angiospermopsida is substituted f Angiospermae the name has to be changed only inside its segement (i.e. the regnu segment).

The addition of new knowledge segments can potentially stretch the availab memory to its limits, since at a run time only the selected module (i.e. out of a potential limitless number) will be active thus requiring only very limited run time capabilities. O modular approach addresses several potential problems one may encounter in pla taxonomy e.g. constant changes in plant classification at species level and moving pla

genera from one family to the other and the use of different taxonomic methods (Cronsquist 1968). The rule based system described in this paper is capable of dealing with all these problems. When there is a change in taxonomic criteria agreed upon in the scientific community, such as in case of the *Opuntia* species, new rules can be added to reflect these changes. Old taxonomic criteria can be moved to a separate knowledge segment where other botanical synonyms or plant names are stored. When one is identifying a plant species, one can obtain the botanical synonyms or plant names that are currently set as the standard. This will ensure that people working with an older or different taxonomic criteria are still able to use the tool system and have opportunity to learn the current taxonomic criteria and terminology that are internationally acceptable for identifying a particular plant species.

CONCLUSION AND FUTURE PROSPECTS

This project has demonstrated the feasability of using CLIPS to build an Opuntia expert tool system. Although, the knowledge base was implemented for a selected number of *Opuntia* species, their available common names and symnonyms. The system can be expanded, at any time, to include more *Opuntia* species in the family Cactacaea in different foreign languages by adding or expanding the parent or subsidiary segments.

The nature of taxonomic definitions and search, in regards to modelling the real world defintions, contains inherently fuzzy concepts and definitions. Logically, fuzzy qualifiers are most appropriate to represent the taxonomic descriptions. In our case we only concern ourselves marginally with this problem while giving the user more options. In an improved version however, we will implement a different mechanism for dealing with fuzzy definitions. In this perspective, a probabilistic value will be attached to the different manifestations of a characteristic, therefore allowing the system to choose on the basis of the value of the characteristic adopted.

REFERENCES

Britton, N.L. and Rose, J.N. (1963) The Cactaceae. Vol. 1 & 2. Dover Press. New York.

Buxbaum, F. (1950). Morphology of cacti. Abbey Garden Press, Pasadena, California.

Cronquist, A. (1968). The evolution and classification of flowering plants. Houghton Mifflin Company, Boston.

Pizzeti, M. (1985). Guide to Cacti and Succulents Simon and Schuster. New York.

National Aeronautics and Space Administration. (1989). Clips Users Guide Version 4.3. Artificial Intelligence Section. Lyndon B. Johnson Space Center.

National Aeronautics and Space Administration. (1990). In CLIPS, first Clips Conference Proceedings, Houston.

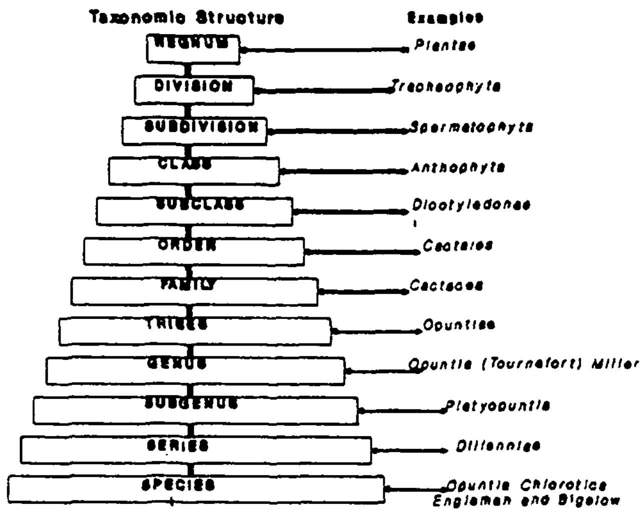


Figure 1.

System Overview

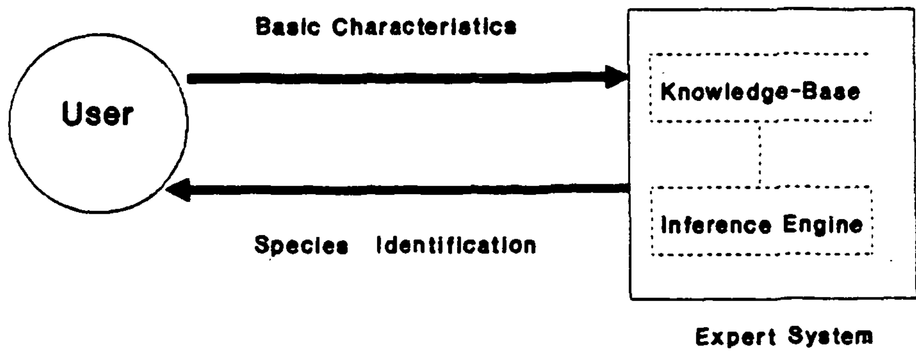


Figure 2.

ORIGINAL PAGE
OF POOR QUALIT

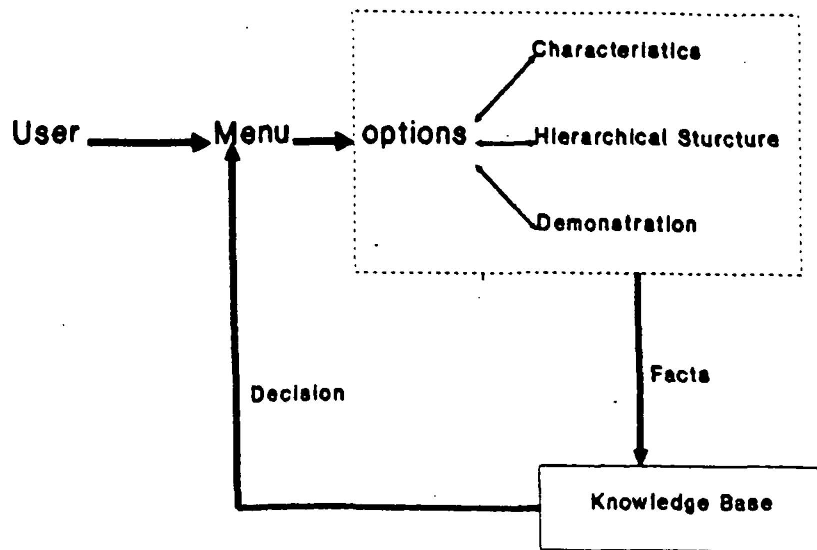


Figure 3.

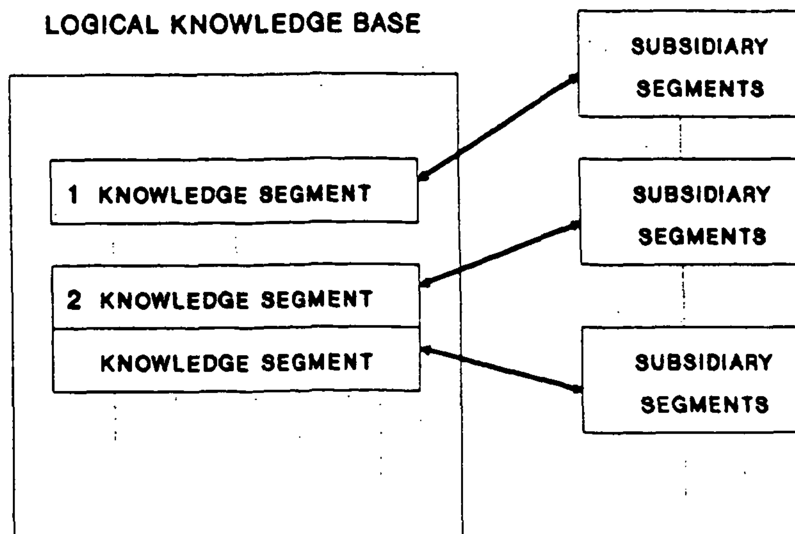


Figure 4. MODULAR CONFIGURATION

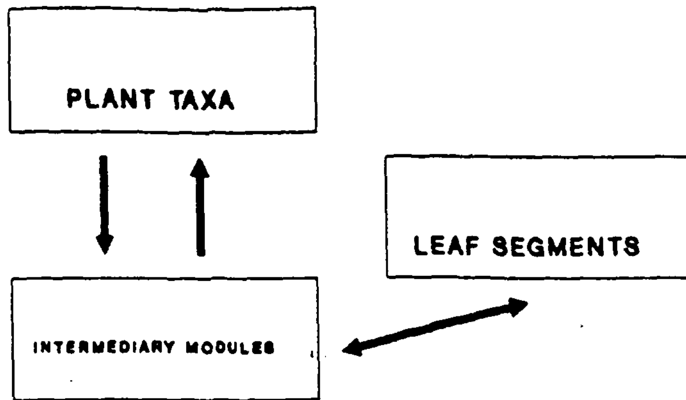


Figure 5. Functional Modules

N92-16615

THE NUTRITION ADVISOR EXPERT SYSTEM

Scott M. Huse and Scott S. Shyne

Rome Laboratory, Computer Systems
Rome Laboratory, Communication Technology

Abstract. The Nutrition Advisor Expert System (NAES) is an expert system written in the C Language Integrated Production System (CLIPS). NAES provides expert knowledge and guidance into the complex world of nutrition management by capturing the knowledge of an expert and placing it at the user's fingertips. Specifically, NAES enables the user to: (1) obtain precise nutrition information for food items, (2) perform nutritional analysis of meal(s), flagging deficiencies based upon the United States Recommended Daily Allowances, (3) predict possible ailments based upon observed nutritional deficiency trends, (4) obtain a top-ten listing of food items for a given nutrient, and (5) conveniently upgrade the database. An explanation facility for the ailment prediction feature is also provided to document the reasoning process.

INTRODUCTION

The Nutrition Advisor Expert System (NAES) is an expert system written in the C Language Integrated Production System (CLIPS). The purpose of NAES is to emulate human expertise in the complex problem domain of nutrition analysis. NAES is a user-friendly, practical expert system with real application potential, e.g., nursing homes, hospitals, doctor's offices, or home use.

NAES allows even a novice user to:

(a) quickly and easily obtain nutrition information on food item(s);
(b) perform nutritional analysis on meal(s);
(c) predict potential ailments based upon nutritional deficiency trends;
(d) generate a top-ten listing of food items for a given nutrient;
(e) conveniently upgrade the database of nutrition knowledge.

NAES consists of a working memory of facts and rules in a knowledge base. The user can easily enter new food item facts into the database or delete existing facts. Consequently, the system can improve its performance with time. The system also features an explanation facility for the ailment prediction module. This important facility provides documentation detailing the reasoning process.

In the development of this system, several artificial intelligence/expert system (AI/ES) techniques were utilized to increase system efficiency and reliability, e.g., fuzzy sets, certainty factors, uncertain evidence, modular design, and efficiency techniques in coding design. The methods by which some of these techniques were implemented in NAES is discussed in detail within the 'System Features' section of this paper.

An expert system should be characterized by good performance, adequate response time, good reliability, understandability, and flexibility (Giarratano and Riley 1989). NAES possesses

each of these important qualities. It is capable of performing quick expert nutritional analysis wit a high degree of reliability and understandability. System flexibility is provided by a convenier and efficient mechanism for modifying the database of nutrition knowledge.

INSTALLATION GUIDE

NAES was written using CLIPS software, version 4.2 for the IBM PC. The software is availabl from the authors on a 5 1/4 inch diskette. In order to run the program, CLIPS must first b installed on your computer. If it is not already installed on your hard drive, copy the CLIP diskette(s) into a sub-directory named 'CLIPS'. Run CLIPS by typing, 'CLIPS'.

Once CLIPS has been installed on your computer, you are ready to run the Nutritio Advisor Expert System. Insert the NAES program diskette into drive A and from the CLIPS directory on your hard drive type, 'copy a:nut .' <RETURN>. This will copy the program into th CLIPS directory of your hard drive. Next, type 'CLIPS' <RETURN>. This will bring up th CLIPS environment. NAES can then be loaded into the system by typing 'load "nut"' <RETURN>. The program will load in about forty seconds, depending of course on you computer system. When the CLIPS' prompt returns, the system is loaded and ready to run. Ru it by typing '(reset)' <RETURN>, and then '(run)' <RETURN>. The (reset) command initialize the system and the (run) command starts program execution.

A main menu should appear with the following six options:

(1) Food Item Information

(2) Dietary Analysis of Meal(s)

(3) Ailment Prediction Based on Nutritional Deficiencies

(4) Top-Ten

(5) Database Update

(6) Exit

To select an option simply enter the menu number followed by the <RETURN> key.

SYSTEM FEATURES

This section provides a detailed discussion of each of the features offered by the system. There are six menu options. A step-by-step explanation of how to use each menu option follows along with a discussion of implementation details.

The first menu option is 'Food Item Information'. Select this option if you are interested in obtaining nutrition information for a given food item. When you select this option you will be prompted to enter the name of the food item. Enter the food item and press <RETURN>.

This module operates in a straightforward manner. The food item name entered by the user is simply matched against the foot item name field in the list of facts. The template used for food items is:

(item <food name> <calories> <calcium> <iodine> <iron> <magnesium> <phosphorus> <sodium> <vitamin A> <vitamin B1> <vitamin B2> <vitamin B6> <vitamin B12> <folic acid> <niacin> <vitamin C> <vitamin D> <vitamin E>)

If a match is found, the corresponding nutritional multi-field values for a standard serving are extracted by the system and displayed on the screen. If, on the other hand, the food item is not present in the database, the user is informed accordingly and instructed to use menu option five,

'Database Update', to update the database. Table 1 shows a sample run of this menu option.

Enter Food Item: beef	
"beef"	
Calories	1481
Calcium	50 mg
Iodine	0 mg
Iron	15 mg
Magnesium	131 mg
Phosphorus	634 mg
Sodium	271 mg
Vitamin A	181 IU
Vitamin B1	0.23 mg
Vitamin B2	0.8999 mg
Vitamin B6	2 mg
Vitamin B12	0 mcg
Folic Acid	0.07 mg
Niacin	18.1 mg
Vitamin C	0 mg
Vitamin D	0 IU
Vitamin E	0.6 mg

Table 1. A run of 'Food Item Information'

The second menu option is 'Dietary Analysis of Meal(s)'. Select this option if you are interested in obtaining nutrition information for a meal or meals. When you select this option you will first be prompted to supply some personal data, i.e., name, age, and gender. If the gender is female and the age is greater than ten, the system will also inquire if you are pregnant. There are fifteen different categories that are utilized based upon sex and age. This information is necessary to determine the appropriate caloric and United States Recommended Daily Allowances (USRDA) guidelines.

Once this information is provided, you will be prompted to enter the name and relative quantities of food items that you either have consumed or plan to consume. This information is categorized on two levels. The first categorization that takes place references the time of day that consumption occurs (breakfast, lunch, or dinner). The second categorization that occurs relates to the nutritional food group that the food item belongs to (meat, dairy, fruit and vegetable, or bread and cereal). If you wish to skip an entry, simply press <RETURN>. Once you have entered all of the necessary information, the system will quickly analyze the nutritional content of your meal(s) and display any nutritional deficiencies based upon your specific USRDA requirements.

The implementation of this module is more complex than that of the first menu option. Each entered food item is asserted into the database in the fact form:

(dayfood <food name> <quantity> <breakfast|lunch|dinner>).

The <quantity> field in this fact represents the fuzzy sets:

quantity = (very small, small, medium, large, very large)

Each member of the fuzzy set represents a factor designed to alter the food's relative

nutritional content based upon serving size. The translation formulas for each member of the fuzzy set are shown in Table 2.

very small	=	<nutrient content>	*	0.50
small	=	<nutrient content>	*	0.75
medium	=	<nutrient content>	*	1.00
large	=	<nutrient content>	*	1.25
very large	=	<nutrient content>	*	1.50

Table 2. Fuzzy set formulas

Once all of the 'dayfood' items have been entered, the 'dayfood' facts are matched against the database of known foods. All of the calories, vitamins, and minerals of the entered foods are totaled and used to generate a fact containing the individual's total nutritional intake for that day. This 'total' nutritional fact is then asserted into the database in the form:

(total <calories> <calcium> <iodine> <iron> <magnesium> <phosphorus> <sodium> <vitamin A> <vitamin B1> <vitamin B2> <vitamin B6> <vitamin B12> <folic acid> <niacin> <vitamin C> <vitamin D> <vitamin E>).

If a 'dayfood' fact does not match an already existing food item in the database, NAES will ask the user if the food item should be asserted into the permanent database of food items. This particular feature of the expert system is very important because it achieves a flexible methodology whereby the ability to 'learn' and improve performance is realized.

Once the new food is asserted, it becomes part of the system's working knowledge base. The nutritional content of the new food item is also added to the 'total' food intake fact so that the complete dietary intake for that day can be used to correlate the nutritional deficiencies with the individual's daily eating habits.

Next, the system determines the person's specific USRDA nutritional group by matching personal data against the database. The system will place the user in one of fifteen different categories based upon age and sex. The nutritional content of the entered food items is compared with the USRDA guidelines for that specific person. The system then generates another fact called 'deficiencies'. It is of the form:

(deficiencies <calories> <calcium> <iodine> <iron> <magnesium> <phosphorus> <sodium> <vitamin A> <vitamin B1> <vitamin B2> <vitamin B6> <vitamin B12> <folic acid> <niacin> <vitamin C> <vitamin D> <vitamin E>).

The system then informs the user of any deficiencies in his or her diet citing the calculated deficiency percentages. The deficient nutrient(s) and their respective deficiencies are then asserted into the database in the form:

(def <nutrient> <amount deficient>).

This information is important because it can answer important health questions such as:

(a) Am I consuming too many calories?
(b) Do I consume appropriate amounts of all the essential nutrients?
(c) What nutrients are lacking in my normal diet?

Table 3 shows a sample run of menu option three, 'Dietary Analysis of Meal(s), by an individual with dietary habits that are less than exemplary.

Please enter your name: Scott
Please enter your age: 24
Please enter your gender (m/f): m

Quantities are: [very small|small|medium|large|very large]

Enter breakfast meat group
Enter breakfast dairy product
Enter breakfast fruit and vegetable group
Enter breakfast bread and cereal group

Enter lunch meat group
Enter lunch dairy product
Enter lunch fruit and vegetable group
Enter lunch bread and cereal group

Enter dinner meat group beef
Enter Quantity of "beef" small
Enter dinner dairy product
Enter dinner fruit and vegetable group peas
Enter Quantity of "peas" very small
Enter dinner bread and cereal group

The following is a list of your USRDA Deficiencies (%):

55.97	Calories
94.13	Calcium
100.00	Iodine
68.5	Magnesium
35.19	Phosphorus
93.79	Sodium
91.28	Vitamin A
74.38	Vitamin B1
52.00	Vitamin B2
25.00	Vitamin B6
100.00	Vitamin B12
86.87	Folic Acid
9.84	Niacin
85.55	Vitamin C
100.00	Vitamin D
97.00	Vitamin E

Table 3. A run of 'Dietary Analysis of Meal(s)'

These deficiencies are utilized by the third major component of this system, the 'Ailment Prediction Based on Nutritional Deficiencies' menu option. Select this option if you are interested in speculating about possible ailments that you may incur based upon the continuation of your observed dietary habits. The knowledge base for this predictive analysis option was derived from the 'Nutrition Almanac' (Kirschmann 1975). This reference notes the fact that nutritional

authorities have linked deficiencies in one or more nutrients to the appearance of a number of diseases. Fortunately, most diseases caused by such deficiencies can be corrected when all essential nutrients are supplied.

This option can only be executed if you have previously run menu option two, 'Dietary Analysis of Meal(s)'. If you attempt to run menu option three, 'Ailment Prediction Based on Nutritional Deficiencies', without first running 'Dietary Analysis of Meal(s)', you will be directed to first run 'Dietary Analysis of Meal(s)'. Assuming that you have previously run 'Dietary Analysis of Meal(s)', selecting menu option three, 'Ailment Prediction Based on Nutritional Deficiencies', will automatically generate a list of zero or more possible ailments that you may incur if you continue to maintain such dietary habits. Using the dietary information entered in menu option two, and selecting menu option three we discover that this particular individual runs the risk of contracting several diseases if he continues to maintain his observed dietary habits.

Ailment: common cold		
Rating: quite possible		
% Deficient:	Vitamin A	91.28 %
	Vitamin B6	25.00 %
	Vitamin C	85.55 %
	Vitamin D	100.00 %
Ailment: rickets		
Rating: quite possible		
% Deficient:	Vitamin D	100.00 %
	Calcium	94.13 %
	Phosphorus	35.19 %
Ailment: scurvy		
Rating: quite possible		
% Deficient:	Vitamin C	85.55 %
Ailment: pellagra		
Rating: possible		
% Deficient:	Vitamin B1	74.38 %
	Vitamin B2	52.00 %
	Niacin	9.84 %

Table 4. A run of 'Ailment Prediction Based on Nutritional Deficiencies'

Table 4 is a sample run of menu option three, 'Ailment Prediction Based on Nutritional Deficiencies'. Listed along with the possible ailments are the fuzzy ratings and percentage deficiencies justifying the possible ailment rating.

This feature works quite efficiently due to the fact that each ailment is a rule. The left hand side of each ailment rule consists of nutrients that, if known to be consistently deficient in a person's diet, may have a causal relationship with the given ailment. These deficiencies are asserted into the knowledge base by running menu option two, 'Dietary Analysis of Meal(s)'. If all of the nutritional deficiencies are present, the ailment rule fires.

The fuzzy rating explanation facility is based upon the average nutritional percentage deficiencies that were calculated in the 'Dietary Analysis of Meal(s)' feature. Ratings are assigned as shown in Table 5.

Average % Deficient	Fuzzy Rating
0 - 25	Slightly Possible
26 - 50	Possible
51 - 75	Fairly Possible
76 - 100	Quite Possible

Table 5. Fuzzy ratings

Menu option four, 'top-ten', generates a top-ten listing of food items for a given nutrient. When you select this feature you will be presented with a seventeen item list (calories and sixteen other key nutrients) from which you are to select one item.

Once you have selected the item of interest, a "Working..." status message will appear. Shortly thereafter, an ordered list of the top-ten foods for that particular nutrient will be displayed on the screen. Table six is a sample run of this menu option.

Vitamin C (mg)		
01	orange	90.00
02	baked potato	20.00
03	banana	15.00
04	peas	13.00
05	corn flakes	8.80
06	pot pie	6.81
07	apple	5.20
08	eggnog	3.00
09	whole milk	2.44
10	apple pie	1.35

Table 6. A run of 'Top Ten'

Implementation of this feature was complicated by the need for a sorting algorithm. Once the user selects the nutrient of interest, the relevant index of the food item facts is known because, by design, there is a direct mapping between the list selection numbers and the food item template nutrition fields. A list is built consisting of all of the pertinent food item nutrient field values.

Next, the numbers are recursively compared in pairs. Beginning at the head of the list, the smaller of the two numbers is removed to another temporary list while the larger of the two remains in the original list. This process is repeated until finally only one number remains in the original list, and that number is the largest number of the original list. That largest number is then asserted into a new 'max' list.

This process is repeated ten times, each iteration using a new list that did not include the previously identified and removed maximum value. Finally, an ordered max list of the top-ten highest numbers is created and then used to match against the database of food item facts. These top-ten food items are then displayed in order on the screen along with their respective nutrient percentages.

Menu option five, 'Database Update', provides the user with a convenient mechanism for updating the database of food item facts. When you select this feature you may either:

(a) add a food item to the database;
(b) retract a food item from the database.

In order to add a food item to the database, you must be able to supply the necessar nutritional details for that food item. To retract a food item, you need only know the name of th food item.

This feature is important because it makes the system flexible. The scope of the pristin database can easily be expanded and thereby improve overall performance. Also, database error are also easily corrected through the retract and add options.

Implementation of this useful feature was simple and direct. To add a food item fact to th database, the user-supplied information is simply asserted using the food item template. To retrac a given food item, a match and retract is performed using the food item name. Table 7 shows sample run for this feature.

```
How would you like to modify the data base?
   1. Add a food item
   2. Retract a food item

Please select 1 or 2:  2

Enter food name:  plum

"plum" has been retracted from the database.
```

Table 7. A run of 'Database Update'

The final menu option is 'Exit'. Select this option only if you want to exit from the Nutrition Advisor Expert System and be returned to the CLIPS' prompt. If you select this option inadvertently, simply type '(reset)' <RETURN>, and '(run)' <RETURN>. This will restart NAES.

Implementation of the exit feature is accomplished quite simply through the system clear screen and halt commands.

FINAL REMARKS

The CLIPS expert system shell provided an excellent developmental environment for the exploration of automated knowledge-based reasoning in the application area of dietary analysis and nutritional guidance. The forward-chaining rule-based language provided inferencing and representation capabilities that allowed the programmers to create the code of the system with a very application-oriented architecture. Facts and rules in the knowledge base could easily be understood because their names directly related the functionality of the rule to the user. The built-in inference engine eliminated the need for the programmer to create any kind of a reasoning mechanism. Additional information can easily be added to the expert system by creating new rules and adding more facts. Through the use of familiar terms in the facts and rules and the elimination of building an inference engine, the CLIPS expert system shell allows a developer to create a full-blown expert system for practically any application in a relatively short period of time.

REFERENCES

Giarratano, J. and Riley, G. (1989). Expert Systems: Principles and Programming, PWS-KENT Publishing Company, pp. 8-9.

Kirschmann, J. (1975). Nutrition Almanac, McGraw-Hill Publishing Company, pp. 108 - 166, 185 - 220.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1991

3. REPORT TYPE AND DATES COVERED
Conference Publication

4. TITLE AND SUBTITLE
2nd CLIPS Conference Proceedings
Volume 2

5. FUNDING NUMBERS

6. AUTHOR(S)
Joseph Giarratano (UHCL) and Christopher Culbert (JSC)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Johnson Space Center
Software Technology Branch/PT4
Houston, Texas 77058

8. PERFORMING ORGANIZATION REPORT NUMBER
S-662

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, D.C. 20546-001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
CP 10085

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unlimited/Unclassified

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Papers presented at the 2nd CLIPS Conference held at the Lyndon B. Johnson Space Center (JSC) September 23, 24, and 25, 1991 are documented herin. CLIPS is an expert system tool developed by the Software Technology Branch at NASA JSC and is used at over 4000 sites by government, industry, and business. During the three days of the conference, over 40 papers were presented by experts from NASA, Department of Defense, other government agencies, universities, and industry.

14. SUBJECT TERMS
CLIPS, expert sytems, knowledge-based systems, Space Shuttle, intelligent tutors, verification and validation, simulation

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited