'94

Third Conference on CLIPS Proceedings (Electronic Version)

September 12–14, 1994
Lyndon B. Johnson Space Center

Third Conference on CLIPS
November 15, 1994

CONTENTS

INTRODUCTION

This document is the electronic version of the *Proceedings of the Third Conference on CLIPS*. It was created using the electronic version of papers supplied by the authors. Not all authors supplied electronic versions of their papers and in other cases the appropriate applications to read the electronic versions were not available, so not all of the papers presented at the conference are in this document. Some pictures and tables contained in the electronic versions of the papers did not survive the conversion process to Microsoft Word for Macintosh. Such deletions are noted as [Figure Deleted] and [Table Deleted]. In addition, all papers were reformatted for uniformity (any errors generated during this process were purely unintentional).

THE BUFFER DIAGNOSTIC PROTOTYPE: A FAULT ISOLATION APPLICATION USING CLIPS

Ken Porter
Systems Engineer
MS: R1-408
Harris Space Systems Corporation
295 Barnes Blvd.
PO Box 5000
Rockledge, FL 32955

This paper describes problem domain characteristics and development experiences from using CLIPS 6.0 in a proof-of-concept troubleshooting application called the Buffer Diagnostic Prototype.

The problem domain is a large digital communications subsystem called the Real-Time Network (RTN), which was designed to upgrade the Launch Processing System used for Shuttle support at KSC. The RTN enables up to 255 computers to share 50,000 data points with millisecond response times. The RTN's extensive built-in test capability but lack of any automatic fault isolation capability presents a unique opportunity for a diagnostic expert system application.

The Buffer Diagnostic Prototype addresses RTN diagnosis with a multiple strategy approach. A novel technique called "faulty causality" employs inexact qualitative models to process test results. Experiential knowledge provides a capability to recognize symptom-fault associations. The implementation utilizes rule-based and procedural programming techniques, including a goal-directed control structure and simple text-based generic user interface that may be re-usable for other rapid prototyping applications. Although limited in scope, this project demonstrates a diagnostic approach that may be adapted to troubleshoot a broad range of equipment.

ON THE DEVELOPMENT OF AN EXPERT SYSTEM FOR WHEELCHAIR SELECTION

Gregory R. Madey, Sulaiman A. Alaraini, and Mohamed A. Nour

Kent State University
Kent, Ohio, 44240
gmadey@synapse.kent.edu

Charlotte A. Bhansin

Cleveland Clinic
9500 Euclid Avenue
Cleveland, Ohio 44195.

ABSTRACT

The prescription of wheelchairs for the Multiple Sclerosis (MS) patients involves the examination of a number of complicated factors including ambulation status, length of diagnosis, funding sources, to name a few. Consequently, only a few experts exist in this area. To aid medical therapists with the wheelchair selection decision, a prototype medical expert system (ES) was developed. This paper describes and discusses the steps of designing and developing the system, the experiences of the authors, and the lessons learned from working on this project. Wheelchair_Advisor, programmed in CLIPS, serves as a diagnosis, classification, prescription, and training tool in the MS field. Interviews, insurance letters, forms, and prototyping were used to gain knowledge regarding the wheelchair selection problem. Among the lessons learned are that evolutionary prototyping is superior to the conventional system development life-cycle (SDLC), the wheelchair selection is a good candidate for ES applications, and that ES can be applied to other similar medical subdomains.

INTRODUCTION

The medical field was one of the first testing grounds for Expert System (ES) technology; the now classic expert system, MYCIN, has often been cited as one of the great breakthroughs in Expert Systems. MYCIN, however, is only one of a large number of expert system applications introduced over the last two decades in the medical field alone [17]. Other examples include NURSExpert [1], CENTAUR, DIAGNOSER, MEDI and GUIDON [17], MEDICS [3], and DiagFH [10] to mention only a few. However, no expert system, to our knowledge, has been developed for the wheelchair selection problem. In this paper, we report on a new application of ES in the medical field; the paper discusses the experiences of the authors with a prototype system developed, using CLIPS, to delineate a wheelchair selection for multiple sclerosis (MS) patients. Our work, therefore, contributes to the existing applications of medical expert/support systems by expanding the domain of applications to the wheelchair selection problem and demonstrating the utility of CLIPS on this problem domain. We will demonstrate that the complexity of the wheelchair selection decision makes it a prime target for an expert system application.

To prescribe a wheelchair for a patient with MS involves more than knowing the patient's disease condition and available choices of potential wheelchairs. A complex web of factors has to be untangled to reach an appropriate choice of a wheelchair. The decision is complicated by such factors as physical body measurements, age and life style, degree of neuralgic impairment, and environmental factors.

MOTIVATIONS FOR COMPUTER-AIDED WHEELCHAIR SELECTION

The motivations for using computer-aided wheelchair selection support fall into two categories: those from the therapist's standpoint and those from the patients' standpoint.

From the Therapist's Standpoint:

The use of computer-aided selection of wheelchairs benefits the therapist in several ways. The prescription of wheelchairs for the MS population is complicated by diverse presentations and symptom fluctuations as well as many other factors. The selection of a well-suited wheelchair is a function of the following variables.

1. **Ambulation status**: In general, a patient is classified as able to walk or not able to walk. In multiple sclerosis, some ability to walk may be limited to short distances or specific terrains. This factors into the type of wheelchair they will need.

2. **Environments to be traversed**: This variable includes both indoor and outdoor locations. The existence of ramps in the patient's residence and the size of the bathroom are among the environmental factors relevant to the choice of an appropriate wheelchair. The frequency of need in each environment helps determine the priority.

3. **Distances to be traversed**: Under this factor, both the indoor and outdoor activities are considered. Self-propelling for short distances may be feasible for some individuals who stay primarily at home, so a manual wheelchair may be appropriate, although disability level may be high. More active users may require a power wheelchair to travel long distances in the community.

4. **Transport of the wheelchair:** Consideration must be made for the wheelchair to disassemble into parts so as to fit in a car or to be sized to fit on public lift-equipped busses.

5. **Caregiver characteristics**: If a caregiver exists for the patient in question, the characteristics of the caregiver(s) are considered in the wheelchair selection. The age, number of caregivers, tolerance for equipment, their health, and the degree of support for the patient are some of the factors to be evaluated when selecting a wheelchair for the patient.

6. **User characteristics**: This variable includes both physical and cognitive dimensions. For the physical, body measurements of the patient are essential to the selection processes, along with qualities of posture, balance, and abnormal muscle tone. Wheelchairs come in made-to-order frame sizes and appropriate sizing is essential. Physical abilities are also evaluated: involuntary movements of the extremities and head are noted. Areas requiring support for best posture and function are documented. As for the cognitive dimension, physical and occupational therapists have to consider the extent to which the patient can safely use the devices. Other issues to be examined by the therapists are the ability to learn the electronic system of a powered wheelchair, to respond quickly in dangerous situations and the ability to report discomfort or problems with fit of the wheelchair.

7. **Length of diagnosis—history of disease course**: This composite variable aids in determining if the MS symptoms of the individual are stable. If they seem stable, a less-modular wheelchair can be appropriate. A progressive disease course would require many modular options for future needs; as the MS symptoms change, it would be possible to modify the wheelchair to fit the needs of the patients.

8. **Currently owned wheelchairs**: Therapists need to consider this item early in their analysis. The current wheelchair may or may not meet some of the needs of the patient. One possibility is that the current wheelchair can be modified to meet the patient's needs. Another possibility is that the wheelchair needs to be replaced because it is inappropriate for current needs, beyond repair or desired modifications can not be performed.

9. **Funding sources of past and potential wheelchairs**: This factor is considered at the end of the process but it is a crucial one. Most patients are restricted in terms of the number of wheelchairs that they can purchase over time under their insurance coverage. Typically, a therapist examines and evaluates the factors that determine the needs of the patient to narrow down the choices of the available wheelchairs. Once the options are reduced, the therapist uses the funding source variable to choose among the options. The funding sources can include Medicaid, Medicare, private insurance (e.g., third party), private purchase, or charity (e.g., MS Society Equipment Loan Program). Each one of these sources has its own rules regarding the wheelchair selection problem. For example, some policies restrict the purchase of a new wheelchair to one every five years. Some will not cover a manual wheelchair if an electric wheelchair was previously obtained. Hence, such restrictions need to be factored in when considering the selection of an appropriate wheelchair.

10. **Current wheelchairs on the market**: There are over 500 models, each offering multiple options of sizing, weight, frame styles, footrests, armrests, cushions, and supports. A current database of technical information would greatly aid in wheelchair selection.

Note that the degree of importance placed on each one of the foregoing factors is not fixed. There is a complex interaction between variables for each patient under consideration. It can be seen from the above illustration that selecting an appropriate wheelchair would be difficult to solve algorithmically. Hence, an expert system is a good candidate for this kind of problem. It was observed by the expert involved in this pilot project that the process of selecting the wheelchair involves both forward and backward reasoning. A therapist starts with the factors that are considered to be important to the patient in question and then narrows down the options available to the patient. This process involves forward reasoning and it is estimated to be eighty percent (80%) of the overall analysis performed by the therapist. The rest of the reasoning, twenty percent (20%), is devoted to backward chaining where the therapist starts with a specific set of wheelchairs and sees if they meet the needs of the patient as well as the requirements of the funding source.

A computer-aided support system can play a significant role in helping the therapist cope with the factors mentioned above. It can guide the therapist in making the best decision about what wheelchair and features need to be prescribed, based on comparison to other successful cases. It can aid the therapist by insuring thorough evaluation. Also, it can help the therapist keep abreast of new products on the market. Such a system insures quality in the wheelchair selection process. An inappropriately prescribed wheelchair usurps coverage and prevents re-prescription of a more appropriate chair. In addition, the standardized reporting format could also be used to conduct more objective studies on wheelchair prescription.

Such a system also has value as a training tool for both novice therapists and therapy students. A tutorial in which real-life or simulated applications are demonstrated can be used for teaching and training. Furthermore, innovations in the wheelchair industry change frequently. The use of computer-based support can overcome this problem. A database of currently available wheelchairs kept and updated on a regular basis, is needed in the field of rehabilitation technology. Finally, the documentation of valuable expertise as reflected by real-life applications will be easier using a computer based system. In this context, an expert therapist is a scarce

resource. Hence, years of experience involving the prescription of numerous wheelchairs can be stored in the system and used later as a reference by therapists who practice in more general areas.

From the Patient's Standpoint:

Of all patients with Multiple Sclerosis (MS), about 40 percent will lose the ability to ambulate. Thus, wheeled mobility stands out as a primary need in this population. Because of the nature of the wheelchair selection problem, it is not unusual for the medical therapist/specialist to prescribe a seemingly appropriate wheelchair for a particular patient only to have the patient reject the wheelchair. The importance of the selection of an appropriate wheelchair for a particular patient cannot be overstated. From the MS patient's standpoint, the selection of a suitable wheelchair is critical for the following reasons:

1. **Insurance**: Because of funding restrictions, the patient might be restricted to a wheelchair for a minimum number of years before being eligible for another wheelchair. The MS patient wants to be sure the right chair is prescribed.

2. **Cost**: The prescription of an appropriate wheelchair should take the cost factor into consideration, especially if the patient is to bear that cost, for patients' resources vary. Also cost consideration is important due to funding restrictions imposed by insurers or Medicaid/Medicare programs. The costs of a wheelchair can range from several hundred to several thousands of dollars.

3. **Mobility and comfort**: The selection of an inappropriate wheelchair will limit already diminished mobility and deny the individual MS patient the potential for increased functional independence from an otherwise suitable wheelchair.

4. **Health**: An inappropriate wheelchair not only may inhibit mobility and cause discomfort, but it may worsen the patient's condition, e.g., postured deformities, pressure sores, etc.

5. **Image and psychological factors**: A suitably selected wheelchair might enhance the patient's personal image, and thus contribute to more community/social involvement. For example, a young MS patient might desire a sporty wheelchair to remain active and socially involved.

Because of the foregoing reasons, it is desirable to have a computer-aided wheelchair selection support system that will hopefully maximize the benefits in the selected wheelchair.

Rationale For Using An Expert System

As was discussed earlier, the selection of the wheelchair for the MS population involves the examination of presentations and symptom fluctuations. Because of the complexity of these factors, only a few therapists are available with a body of expertise to tackle the wheelchair selection decision. A computer-aided system, however, would capitalize on this expertise and make it more widely available. Hence a knowledge-based system seems appropriate, more specifically, a knowledge-based expert system. The next section discusses medical expert systems in general and develops a taxonomy for them. We then show where our prototype system fits relative to this taxonomy.

TAXONOMIC FRAMEWORK FOR MEDICAL EXPERT SYSTEMS

The wide range of intelligent (knowledge-based) medical systems today can be broadly classified using the taxonomy shown in Figure 1. This taxonomy is based on three broad dimensions: technology, domain, and application type.

1. **Technology**: Technology is further divided into: 1) hardware platform (e.g. PC-based, workstation-based, etc.), 2) AI method (solution), and 3) programming tools. A medical knowledge-based system can thus be classified on whether it is PC-based, mainframe-based, etc. It can also be classified on whether it is an expert system solution [4], a neural network (ANN) [9], a natural language system, interactive hypermedia [8], a paper-based [16], etc. Programming tools include AI programming languages and shells. Examples include OPS5, Lisp, Prolog, and CLIPS [15].

2. **Domain**: Knowledge-based systems have been applied in a variety of medical subdomains [1, 2, 3, 4, 10, 14, 17]. Example subdomains include: heart diseases, blood analysis, asthma, artificial limbs, childhood diseases, and this project on multiple sclerosis (MS). It is difficult, however, to neatly classify medical computer-aided systems on the basis of medical subdomains since many of these systems have overlapping domains.

3. **Application Type**: The application type dimension describes the function of the knowledge-based system for which it is developed. These applications types include diagnosis [10], classification [17], prescription/selection [15], tutoring/training [14], data analysis and interpretation, prognosis, and knowledge/technology transfer. Many knowledge-based systems are built to support more than one of these functions.

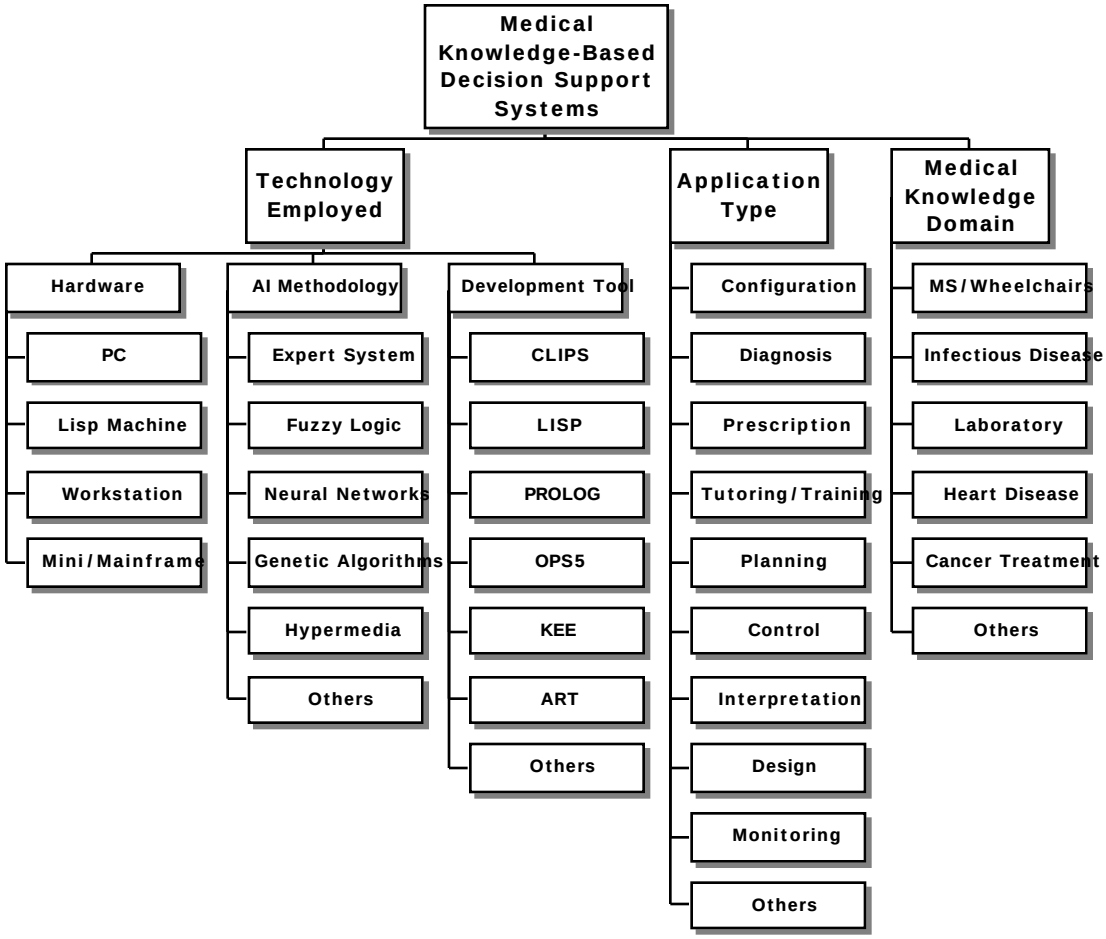


Figure 1. A Taxonomic Framework for Knowledge-Based Decision Support Systems

Review of Medical Expert Systems

Our emphasis in this paper is on medical *expert systems*, which is a subset of the computer-aided support systems in the technology dimension mentioned above. Some of the well known medical expert systems include the following [17]:

1. **CENTAUR**: The domain of this expert system is lung diseases, developed in the INTERLISP programming tool by the Stanford University. Operational functions include diagnostic interpretation of pulmonary function tests.

2. **DIAGNOSER**: Deals with heart diseases, develop in LISP by the University of Minnesota.

3. **GUIDON**: The medical domain include bacterial infections. It is developed in INTERLISP by the Stanford University.

4. **MDX**: Deals with liver problems, developed in LISP by the Ohio State University.

5. **MED1**: Deals with chest pain, developed in INTERLISP at the University of Kaiserlautern.

6. **MYCIN**: Best known of all medical expert systems, MYCIN's medical subdomains include bacteremia, meningitis, and Cystis infections. It was developed at Stanford University and the main operational functions include diagnosis of the causes of infections, treatment, and education.

7. **NEUREX**: Concerned with the nervous system, NEUREX was developed in LISP at the University of Maryland. Its functions include diagnosis and classification of the diseases of the nervous system.

8. **CARAD**: This expert system handles radiology; it was developed at the Free University of Brussels. Its main functions is the interpretation and classification of X-ray photographs [3].

Our Wheelchair_Advisor stands apart from these expert systems listed above by its unique domain of wheelchair prescription for MS patients and our choice of the programming tool. This project involved the use of a PC and the expert system shell CLIPS [6, 7, 13, 18]. The functions/objectives of the Wheelchair_Advisor included diagnosis, classification, prescription, and training. Figure 2 maps these characteristics into a classification scheme to show where our prototype expert system fits relative to current computer-aided medical systems. As Figure 2 indicates, and to our best knowledge, no other expert system application has been developed in the domain of wheelchairs for MS patients.

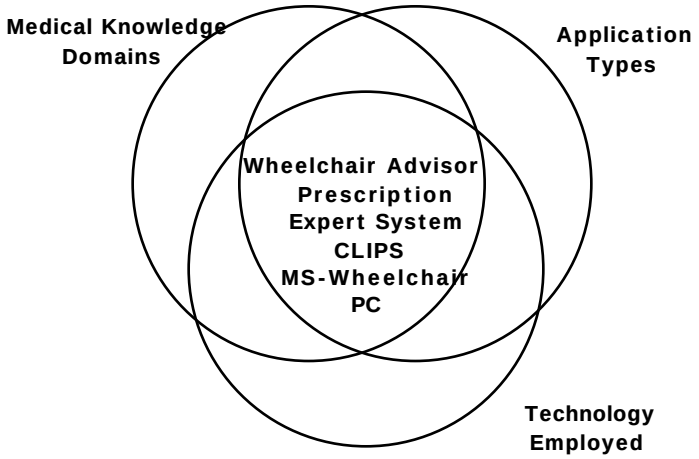


Figure 2. Classification Framework for Medical Decision Support Systems

THE WHEELCHAIR EXPERT SYSTEM PROJECT

The Environment

The Cleveland Clinic Foundation's Mellen Center for Multiple Sclerosis Treatment and Research was initiated in 1985 with a grant from the Mellen Foundation. The Mellen Center, the largest and most comprehensive full-time MS center in the country, is an interdisciplinary outpatient rehabilitation facility providing direct patient care, education, and basic and clinical research into the causes and management of MS. In 1993, the Mellen Center had 14,000 patient visits for its services of neurology, nursing, occupational therapy, physical therapy, psychology, and social work. Approximately 350 new patients are seen each year.

The Knowledge Engineering Process

The knowledge engineering process has often been described as the "knowledge engineering bottleneck" due to the difficulty and complexity of this process. To deal with the complexity of the knowledge engineering process, three basic methodologies were used to elicit knowledge from the expert: interviews, insurance documents, forms, and prototyping.

1. **Interviews**: Multiple interviews were conducted with the expert by three knowledge engineers (KE) all of whom, including the expert, are the authors of this paper. A typical session lasted from 3 to 5 hours.

2. **Insurance Letters/Other Forms**:The insurance and other prescription forms supplied the knowledge engineers with the missing links in the pieces of knowledge gained from the interviews. These forms embodied actual cases describing patient symptoms, condition, cognitive/psychological state, and the recommended wheelchair. Because of the difficulties of obtaining sufficient knowledge using interviews only, as pointed out above, the knowledge obtained from these documents was invaluable inasmuch as it complemented the expertise derived directly from the expert.

3. **Prototyping**: The interviews went side by side with an actual prototype developed to foster better communication between the expert and the KE's. This helped offset some of the limitations of the interviewing process. Each subsequent version of the prototype provided a chance for the expert to "endorse" the KE's interpretation of the knowledge supplied in the previous interview. At times the expert would clarify a previous answer and supply a new one; thus it became clear that the prototype helped correct errors in communication and misinterpretations.

The System-Building Process

The project was conducted in an interactive fashion and rapid prototyping was used to develop the system. Figure 3 shows the block diagram of the prototype system. First, the patient's needs and constraints are considered. This data can be provided on line or by using an input text file in which the data about a particular patient is stored. To accomplish this task a number of rules of the type IF/THEN are implemented. The result of this examination, which is a template of facts about the patient in question, is then used by the search module which in turns uses this information while searching the wheelchair database to find the appropriate wheelchair(s). Note that the optimizer module consists also of IF/THEN rules. As for the wheelchair database, it contains a list of wheelchairs with different features. An explanation facility where the reasoning of the system is explained to the user can be added to the system. Finally, there is a solution set module where the recommendations of the ES are included. In the next subsection, a description of CLIPS, an expert system language, is presented. Then, sample screens and dialogue are shown.

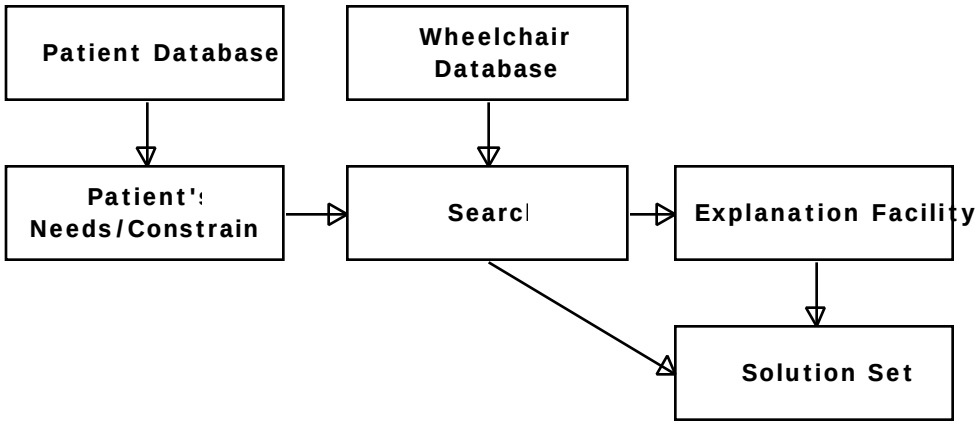


Figure 3. Block Diagram of the System Design

CLIPS

CLIPS (short for C Language Integrated Production System), developed at NASA/Johnson Space Center, has recently shown increasing usage [6, 7, 11, 13]. CLIPS is a forward-chaining rule-based language that resembles OPS5 and ART, other widely known rule-based development environments. Figure 4 shows the basic components of CLIPS, which are essential for an ES. Following this figure is a brief description of each component.

1. User Interface: The mechanism by which the user and the expert system communicate.

2. Fact-list: A global memory for data. For example, the primary symptom of an MS patient can be represented in CLIPS syntax as in Figure 5. For clarity, the reserved key words of CLIPS are printed in bold letters.

3. Knowledge-base: Contains all the rules used by the expert system. For instance, consider the following partial rule that is used by the system to list all the primary symptoms of an MS patient:

IF user has a primary symptom of cerebellar ataxia
THEN the primary symptom is cerebellar ataxia

In the CLIPS syntax, this rule and the associated dialogue can be written as shown in Figure 6.

4. Inference engine: Makes inferences by deciding which rules are satisfied by facts, prioritizes the satisfied rules, and executes the rule with the highest priority.

5. Agenda: A prioritized list created by the inference engine of instances of rules whose patterns are satisfied by facts in the fact list. The following shows the contents of the agenda at some stage:

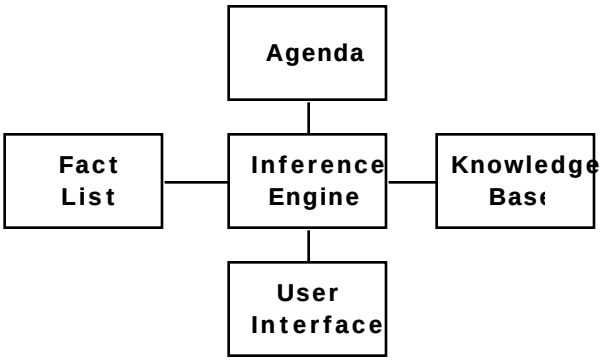


Figure 4. CLIPS Basic Components. Adapted from [6].

English:
The primary symptom of the patient is cerebellar ataxia.

CLIPS:

```
(deffacts user-data
    (ms symptoms primary cerebellar ataxia)
 )
```

Figure 5. CLIPS Syntax for storing facts

In Figure 7, three instantiated rules are placed on the agenda. Each entry in the agenda is divided into three parts: Priority of the rule instance, name of the rule, and the fact-identifiers. For the first entry in the agenda, for example:

- 2 refers to the priority.
- ms-symptom-primary is the name of the rule.
- f-5 is the fact-identifier of the fact that matches the pattern of the rule. Such facts are stored as in Figure 5.

```
(defrule ms-symptoms-primary
     ?phase <- (phase ms symptom)
=>
     (retract ?phase)
     (printout t crlf "What is the primary symptom of the MS
                       patient? ")
     (bind ?answer (readline))
     (if (not (stringp ?answer))
     then (printout t crlf "Please check again!" crlf)
     (assert (phase ms symptom))
     (if (stringp ?answer)
     then (bind $?sym (str-explode ?answer))
     (assert (ms symptoms primary $?sym secondary))))
```

Figure 6. CLIPS Syntax for rules

```
Agenda
    2 ms-symptoms-primary:           f-5
    1 ms-symptoms-secondary:         f-6
    0 ms-symptoms-secondary-more     f-7,f-8
```

Figure 7. CLIPS Agenda

Sample Screens And Dialogue

The above rule, the ms-symptoms-primary rule, can be used to show a scenario of a dialogue between the end user (e.g., a physical therapist) and the expert system as follows:

```
WHAT IS THE PRIMARY SYMPTOM OF THE MS PATIENT?
cerebellar ataxia
WHAT IS THE SECONDARY SYMPTOM OF THE MS PATIENT?
weakness
```

Figure 8. A Sample screen of a dialogue in a session

Based on the new information provided by the end user, the data about the patient will be updated. Accordingly, the fact-list will include a new fact which shows the name of the primary symptom of this patient. The resulting fact is presented in Figure 5. Another impact of this new information will be to update the agenda to include the next rule to be fired, the ms-secondary-symptom rule in this case. This is possible because a new fact, f-5, which was entered by the user as an answer to an on-screen question, now satisfies this rule.

LESSONS LEARNED

There are many lessons to be learned from this project. First: the evolutionary prototyping in designing expert systems is proven to be superior to conventional system development life-cycle. Figure 9 shows the steps involved in designing a system under the traditional method.

Figure 9. System Development Life Cycle (SDLC)

On the other hand, prototyping presents a more efficient way to design a system. Under this method, the end user will be aware of the costs/benefits and, most importantly, will be a part of the development team. In essence, the system will be modified a number of times until the desired system is obtained. Figure 10 shows the steps involved in this method.

Figure 10. Evolutionary Prototyping

Second: the expert system developed in this project has shown the wheelchair selection problem to be a good candidate for ES applications. This project has also shown that there are major

benefits for both the medical practitioners and the MS patients to be derived from such an application. Third, it is evident from this project that other similar medical subdomains might be good candidates for the application of the ES technology. Our project serves to expand the medical applications domain. Fourth, CLIPS was found to flexible, powerful, and intuitive development environment for this application.

CONCLUSIONS

The authors of this paper were involved in a project concerned with the actual development of a wheelchair selection expert system. A prototype expert system (Wheelchair_Advisor) was developed, using CLIPS, to prescribe wheelchairs for Multiple Sclerosis (MS) patients. This paper reports the process, the experiences of the authors, the advantages of evolutionary prototyping for expert system development, and the possibilities for new medical subdomains as candidates for expert system applications.

Our findings show that there are major advantages for using an expert system tool to aid in the analysis and selection of a wheelchair for an MS patient. Such an expert system can also be used as a training and educational tool in the medical industry.

REFERENCES

1. Bobis, Kenneth G. and Bachand, Phyllis M., "NURSExpert: An Integrated Expert System Environment for the Bedside Nurse," in proceedings of *IEEE International Conference on Systems, Man and Cybernetics*, Chicago, 1992, pp. 1063-1068.

2. Bobis, Kenneth G. and Bachand, Phyllis M., "Care Plan Builder: An Expert System for the Bedside Nurse," in proceedings of *IEEE International Conference on Systems, Man and Cybernetics***,** Chicago, 1992, pp. 1069-1074.

3. Bois, Ph. DU, Brans, J.P., Cantraine, F., and Mareschal, B., "MEDICS: An expert system for computer-aided diagnosis using the PROMETHEE multicriteria methods," *European Journal of Operations Research*, Vol. 39, 1989, pp. 284-292.

4. Cagnoni S. and Livi R., "A Knowledge-based System for Time-Qualified Diagnosis and Treatment of Hypertension," in Computer-Based Medical Systems: Proceedings of the *Second Annual IEEE Symposium*, June 26-27, 1989, Minneapolis, Minnesota. pp. 121-123.

5. Fieschi, M., *Artificial Intelligence in Medicine*, Chapman and Hall, London, 1990.

6. Giarratano, Joseph and Riley, Gary, *Expert Systems: Principles and Programming*, PWS-Kent Publishing Company, Boston, 1994.

7. Gonzalez, Avelino J. and Dankel, Douglas D., *The Engineering of Knowledge-Based Systems: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

8. Hammel, Joy M., "Final Report of the AOTA/Apple Grantees," Veterans Administration, Palo Alto, CA, March 1992.

9. K. Kuhn, D. Roesner, T. Zammler, W. Swobodnik, P. Janowitz, J. G. Wechsler, C. Heinlein, M. Reichert, W. Doster, and H. Ditshuneit., "A Neural Network Expert System to Support Decisions in Diagnostic Imaging," in *Proceedings of IEEE 4th Symposium on Computer-Based Medical Systems***,** May 12-14, 1991, Los Angeles, pp. 244-250.

10. Lin, W. and Tang, J.-X., "DiagFH: An Expert System for Diagnosis of Fulminant Hepatitis," in *Proceedings of IEEE 4th Symposium on Computer-Based Medical Systems*, May 12-14, 1991, Los Angeles, pp. 330-337.

11. Martin, Linda and Taylor, Wendy. *A Booklet of CLIPS Applications*, NASA, Johnson Space Center, Houston, TX, 1992.

12. Mouradian, William H., "Knowledge Acquisition in a Medical Domain," *AI Expert*, July 1990, 34-38.

13. NASA, Lyndon B. Johnson Space Center, *CLIPS Basic Programming Guide*, 1991., Houston, TX.

14. Prasad, B. , Wood H., Greer, J. and McCalla G., "A Knowledge-based System for Tutoring Bronchial Asthma Diagnosis," in *Computer-Based Medical Systems: Proceedings of the Second Annual IEEE Symposium*, June 26-27, 1989, Minneapolis, Minnesota. pp. 40-45.

15. Stylianou, Anthony C. and Madey, Gregory R., "An Expert System For Employee Benefits Selection: A Case Study," *Journal of Management Systems*, Vol. 4, No. 2, 1992, pp. 41-59.

16. Ward, Diane and Reed, Stephanie, "PowerSelect: A Prototype for Power Mobility Selection—Based Upon Human Function," in *Proceedings of the Ninth International Seating Symposium***,** February 25-27, 1993, Memphis, TN, pp. 307-310.

17. Waterman, Donald A. *A Guide to Expert Systems*, Addison-Wesley Publishing Company, 1986.

18. Wygant, Robert M., "Clips—A Powerful Development and Delivery Expert System Tool" *Computers in Engineering*, Vol. 17, Nos. 1-4, 1989, pp. 546-549.

EXPERT WITNESS - A SYSTEM FOR DEVELOPING EXPERT MEDICAL TESTIMONY

Raymond Lewandowski, MD.
Center for Genetic Services
7121 S. Padre Island Dr.
Corpus Christi, Texas 78412

David Perkins and David Leasure
Department of Computing and Mathematical Sciences
Texas A&M University - Corpus Christi
6300 Ocean Dr.
Corpus Christi, Texas 78412

ABSTRACT

Expert Witness is an expert system designed to assist attorneys and medical experts in determining the merit of medical malpractice claims in the area of obstetrics. It this by substitutes the time of the medical expert with the time of a paralegal assistant guided by the expert system during the initial investigation of the medical records and patient interviews. The product of the system is a narrative transcript containing important data, immediate conclusions from the data, and overall conclusions of the case that the attorney and medical expert use to make decisions about whether and how to proceed with the case. The transcript may also contain directives for gathering additional information needed for the case.

The system is a modified heuristic classifier and is implemented using over 600 CLIPS rules together with a C-based user interface. The data abstraction and solution refinement are implemented directly using forward chaining production and matching. The use of CLIPS and C is essential to delivering a system that runs on a generic PC platform. The direct implementation in CLIPS together with locality of inference ensures that the system will scale gracefully. Two years of use has revealed no errors in the reasoning.

INTRODUCTION

When preparing a medical malpractice lawsuit, an attorney must identify the relevant facts and use them to decide first if the case has merit. Usually, the attorney consults a medical expert to evaluate the client's medical records and to advise the attorney. The problems for attorneys and clients is that medical experts are both expensive and relatively scarce, the problem of determining fault is tedious and time consuming, and the case load is growing.

Our approach to this problem is to make a preliminary determination of merit without investing large amounts of time from a medical expert. Using an expert system called Expert Witness, the paralegal staff can be guided in their examination of medical records and conducting of client interviews. After data collection, Expert Witness produces a transcript of reasoning that aids the attorney and medical expert in determining the validity of a case. The transcript is very similar to what the medical expert would also have produced, except that it was created with far less expense. By taking this approach, an attorney can determine the preliminary merits of a lawsuit while saving substantial amounts of money. The attorney and medical expert can take on more work. Deserving cases are more likely to be pursued because more cases can be handled overall. Fewer non-meritorious, wasteful cases need be pursued, resulting in saved expense and anguish. Overall, in two years of operation, Expert Witness has been tested in 10 legal offices on numerous cases with no complaints, and based on the success of the system, significant development is planned to greatly expand its coverage.

This paper describes the functional architecture, the implementation, the history, and the plans for expansion of Expert Witness. It begins with a functional overview of the Expert Witness in the *Functional Description*Section. After the functional description, some typical cases are described in the *Example Cases* Section. In the *Implementation, History, and Future Directions* Section, the implementation of the current system in CLIPS and C is described, as is the history of the project, and the future directions. Results and conclusions are given in the *Results and Conclusions* Section.

FUNCTIONAL DESCRIPTION

The current domain of expertise for Expert Witness is obstetrical malpractice. The overall context in which Expert Witness determines the extent of medical malpractice is shown in Figure 1. To determine the fault of medical personnel, Expert Witness directs a paralegal in the search for relevant medical facts from patient records and patient interviews. Such information includes the family history, the patient history, the history of the mother prior to birth, the events and medical procedures performed at birth, and subsequent tests and treatment. Expert Witness builds a case file for each client. This multiple client feature allows the paralegal to start and stop data collection corresponding to the availability of information and access to the client. When sufficient data has been collected, a narrative transcript and a fact summary is produced. The narrative transcript is similar to what the medical expert would have produced. It marks the important details, such as confirming or disconfirming evidence, presents reasoning chains based on evidence, suggests further tests, and derives conclusions regarding the viability of the case. The transcripts and the fact summaries are used by the attorney and the medical expert to make the final decision whether malpractice contributed to the client's condition, and also to determine what additional data need collected. The general philosophy embedded in Expert Witness's knowledge is to only make conservative conclusions, based on practice that is well accepted in the field.

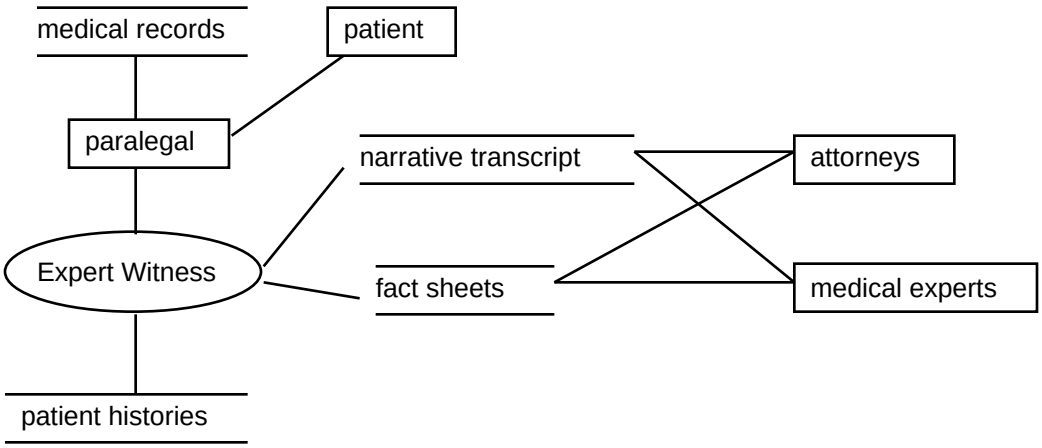


Figure 1. Context of Expert Witness

EXAMPLE CASES

The following cases are summaries of two actual cases handled by Expert Witness.

Case I

An infant was born with apgar scores of 8 and 9. The birth weight was six pounds. During the mothers labor, monitoring indicated that the baby was in distress. In response to the data suggesting distress, the physician treated the mother and reviewed the mother’s medications. It

was found that one of the medications that the mother was taking is known to create false positive findings of fetal distress. Normally, the distress patterns would have lead to a cesarean section. By reviewing the data correctly, the physician avoided an unnecessary surgery which carries added risks for the mother. The Expert Witness program analyzed the data and advised the user that the physician had acted appropriately based upon the facts presented. This analysis prevented a potentially frivolous law suite.

Case II

A child is known to be mentally retarded. The child was born with apgar scores of 2 and 5. During labor, the mother had a biophysical profile which was abnormal. After delivery, the infant developed low blood sugar and seizures. Family history revealed that the mother has a nephew by one of her sisters who is also mentally retarded. The Expert Witness program analyzed the data and advised the user that there appeared to be some improprieties on the part of the physician during the mother's labor that could have contributed to the child's present condition. It also noted, however, that there may have been a pre-existing condition which may be the main contributor to the child's problems. It suggested that further analysis is necessary. This is a case that deserved further in depth analysis by actual expert witness physicians.

IMPLEMENTATION, HISTORY, AND FUTURE DIRECTIONS

Expert Witness is used cyclically to build up a patient file. Within each cycle there are two stages, data collection and data inference. Data collection is done interactively as the paralegal presents known information to the system through a text based user interface written in C. Once all known information is provided, the inference phase begins, and the known data are analyzed to determine what conclusions are able to be made and what they are. When more information is needed, additional data are suggested in the transcript. The medical expert may also direct the paralegal to obtain more information. The next cycle of data collection/inference process allows direct entry of any additional information, and produces a more complete narration.

The inference part of the system is written in CLIPS 4.3.[1] Over 600 rules constitute the knowledge base. The basic architecture is an elaboration of the heuristic classification model.[2] Initial data are abstracted from quantitative values to qualitative categories. Matching, based directly on CLIPS rule matching, is used to determine the first level of solutions in the form of direct conclusions in the narrative transcript. Additional reasoning is performed to produce the next level of conclusions, based on the initial level. In contrast to some heuristic classifiers which seek to produce one conclusion and may take the first one that is satisfactory, Expert Witness makes all conclusions that it can. It uses a mix of reasoning methods, using some data to strengthen and some data to weaken conclusions. It does not use certainty factors or other approximate reasoning methods, since the qualitative representation of strength of belief using basic CLIPS was adequate for the conservative reasoning philosophy adopted for the system.

The performance of Expert Witness has been very good. The knowledge used has generally been localized, and the reasoning chains have been kept relatively short. Factoring of the rule base into a number of independent subsystems for determining the first level of conclusions has also helped. The second level conclusions are made using a rule base that is loaded after all first level conclusions have been made.

---

[1] CLIPS Release Notes, Version 4.3, NASA Johnson Space Flight Center, June 13, 1989.

[2] Clancey, W. J., "Heuristic Classification," *Artificial Intelligence*, **27:3**, December 1985, pp. 289-350.

Expert Witness was built over a period of 5 months beginning in 1991. The initial knowledge engineer and expert was Dr. Ray Lewandowski, a medical consultant and clinical geneticist. The user interface was constructed by David Perkins at Texas A&M University-Corpus Christi. The system has since been used by ten attorneys and their staff. Follow-up consultations are performed with Dr. Lewandowski. Plans are underway to increase the number of users. In the several years since being introduced in the field environment, no incorrect recommendations have been made, and much time has been saved.

Based on the success of the initial system and demands of the users for broadening the scope of application, additional experts are currently being interviewed in the areas of neonatology, expanded obstetrical coverage, and hospital practices and procedures. Additional modules beyond those are in the planning stage. No significant changes to the structure of the knowledge base are expected. Knowledge should remain localized, and the performance penalty should grow linearly with the number of systems. Each system will be incorporated so that it can function as a stand-alone or integrated component of the entire system.

RESULTS AND CONCLUSIONS

The system has since been used continuously since its development by ten attorneys and their staff. In the several years since being introduced in the field environment, no incorrect recommendations have been made, and much time has been saved. Based on this extended success, plans are underway to increase the number of users and the scope of the system's coverage.

A critical success factor for Expert Witness, aside from the quality of the knowledge base, has been the need for it to run on a generic hardware platform. The use of CLIPS has allowed us to keep the system small, while maintaining speed and ease of programming, both because the inference component is small and because it easily interfaced with a compact C user interface.

The second critical success factor derived from CLIPS is the suitability of the forward reasoning and matching to the application and representation of the knowledge. Although CLIPS would have allowed it, no meta-level reasoning was necessary. This simplicity allowed the knowledge base to grow to over 600 rules without greatly affecting the structural complexity of the knowledge or the cost of using it. On the face of it, the plainness of the knowledge representation as rules speaks against this system when compared to more complicated knowledge structures and control regimes, but in reality, the degree of fit between the knowledge and the inference system has allowed us to create and maintain a reasonably large knowledge base cheaply and reliably. This simplicity is crucial for us when we consider expanding the knowledge base as much as fivefold, which we intend to do.

A CLIPS-BASED EXPERT SYSTEM FOR THE EVALUATION AND SELECTION OF ROBOTS

Mohamed A. Nour
Felix O. Offodile
Gregory R. Madey
(gmadey@synapse.kent.edu)
Administrative Sciences
Kent State University
Kent, Ohio 44242
USA

ABSTRACT

This paper describes the development of a prototype expert system for the intelligent selection of robots for manufacturing operations. The paper first develops a comprehensive, three-stage process to model the robot selection problem. The decisions involved in this model easily lend themselves to an expert system application. A rule-based system, based on the selection model, is developed using the CLIPS expert system shell. Data about actual robots is used to test the performance of the prototype system. Further extensions to the rule-based system for data handling and interfacing capabilities are suggested.

INTRODUCTION

Many aspects of today's manufacturing activities are increasingly being automated in a feverish pursuit of quality, productivity, and competitiveness. Robotics has contributed significantly to these efforts; more specifically, industrial robots are playing an increasingly vital role in improving the production and manufacturing processes of many industries [6].

The decision to acquire a robot, however, is a nontrivial one, not only because it involves a large capital outlay that has to be justified, but also because it is largely complicated by a very wide range of robot models from numerous vendors [6]. A non-computer-assisted (manual) robot selection entails a number of risks, one of which is that the selected robot might not meet the task requirements; even if it does, it might not be the optimal or the most economical one. Mathematical modeling techniques, such as integer programming, are rather awkward and inflexible in tackling this problem. The reason for this is that the robot selection process is an ill-structured and complex one, involving not only production and engineering analysis, but also cost/benefit analysis and even vendor analysis. Its ill-structured nature does not readily lend itself to tractable mathematical modeling. Therefore, nontraditional approaches, such as expert systems (ES) or artificial neural networks (ANN), seem intuitively appealing tools in these circumstances.

When the decision maker (DM) is charged with making the selection decision, he or she is being called upon to play three roles at the same time, namely (1) financial analyst, (2) robotics expert, and (3) production manager. In other words, the decision maker would need to make three different (albeit related) decisions: (1) choosing the best robots that match the task requirements at hand, (2) choosing the most cost effective one(s) from those that meet the requirements, and (3) deciding from which vendor to order the robot(s). We shall call these decisions *technical*, *economic*, and *acquisitional*, respectively. Clearly, these are very complex decisions all to be made by the same decision maker. Supporting these decisions (e.g., by a knowledge-based system) should alleviate the burden from the decision maker and bring some consistency and confidence in the overall selection process. The success of the ES technology in a wide range of application domains and problem areas has inspired its use as a vehicle for automating decisions in production and operations management [1, 19], as well as the robot selection decision [16, 17].

In this paper, a three-stage model is presented for the robot selection process. The model is *comprehensive* enough to include the major and critical aspects of the selection decision. It is implemented in a CLIPS-based prototype expert system. The rest of the paper is organized as follows. In the following section, we review previous work and, in the third section, we present our three-stage model to robot selection. In the fourth section, the implementation of the prototype expert system is discussed. Limitations of, and extensions to the prototype expert system with database management (DBMS) capabilities are provided in section five. We conclude the paper in section six.

MOTIVATION AND RELATED WORK

The application of knowledge-based systems in production and operations management has been investigated by a number of researchers [1]. In particular, the application of ES in quality control [3], in job shop scheduling [18, 19], and industrial equipment selection [11, 20] has been reported in the literature. The robot selection problem is prominent in this line of research [8, 15, 17].

In an earlier paper by Knott and Getto [9], an economic model was presented for evaluating alternative robot systems under uncertainty. The authors used the present value concept to determine the relative worthiness of alternative robot configurations. Offodile, *et al.* [14, 15] discuss the development of a computer-aided robot selection system. The authors developed a coding and classification scheme for coding and storing robot characteristics in a database. The classification system would then aid the selection of the robot(s) that can perform the desired task. Economic modeling can then be used to choose the most cost-effective of those selected robots. Other related work includes Offodile *et al.* [16], Pham and Tacgin [17], and Wang, *et al.* [21].

A review of the above literature indicates that these models are deficient in at least one of the following measures:

- *Completeness:* We suggested earlier that the robot selection problem involves three related decisions. The models presented in the literature deal invariably with at most two of these decisions. The other aspects of the selection decision are thus implicitly assumed to be unimportant for the robot selection problem. Experience suggests that this is not the case, however.

- *Generality:* the models presented are restricted to specific production functions, e.g., assembly. Many of today's production functions are not monolithic but rather a collection of integrated functions, i.e., welding, assembly, painting, etc. In particular, industrial robots are by design multi-functional and a selection model should be robust enough to evaluate them for several tasks.

- *Focus:* The focus in the literature is often more on robot characteristics (robot-centered approach) than on the task the robot is supposed to perform (task-centered approach). We posit that task characteristics should be the primary focus in determining which robot to choose, not the converse.

We propose a three-stage model that captures the overall robot selection process, with primary emphasis being given to the *characteristics* and *requirements* of the *task* at hand. The proposed task-centered model is *comprehensive* in the sense that it covers the robot selection problem from the task identification, through robot selection, to vendor selection and, possibly, order placement. The model is also *general* in the sense that it applies to a wider range of industrial robot applications. While this selection model is different from previous approaches, it

incorporates in a systematic manner all the critical decisions in any sound robot selection process. The sequential order of these decisions, and the related phases, is important from a logical as well as an efficiency standpoints. We cannot, for example, separate the technical decision from the economic decision, for a robot that is technically well suited to do the job might not be economical; and vice versa. We shall present our model in the following section and in the subsequent sections discuss its implementation in a knowledge-based system.

A THREE-STAGE MODEL FOR ROBOT SELECTION

Figure 1 depicts the three-stage robot section scheme proposed in this paper. We present a general discussion of the scheme in the subsequent subsections.

- *Technical decision:* This is the first and the most critical decision to be made. It is the formal selection of one or more candidate robots that satisfy the minimum requirements and characteristics of the task to be performed. It is technical in the sense that it would normally be made by the production or process engineer upon a careful analysis of the technical characteristics of both the task and the robot. This decision is the most difficult of the three. A thorough analysis is required to arrive at the initial set of feasible robots.

- *Economic decision:* This is a decision involving the economic merit of the robot. More specifically, it is a decision about the most cost-effective robot alternative(s) considering both initial cost (purchase price) and operating costs. The purpose of the initial and operating costs is twofold: (1) to allow for a rough justification for the robot, and (2) to allow for a choice to be made among rival robots. Suppose, for example, that we had a choice of two robots from Stage One (to be described shortly)—one which is adequate for the task and costs within reasonable range; the other is more technologically advanced but costs well beyond what is considered economical for the task in question. Clearly the estimated cost of the latter would force us to choose the former robot.

- *Acquisitional decision:* This is simply deciding which vendor to acquire the robot(s) from. The choice of a vendor is based not only on purchase price, but also on service and quality.

The following three stages implement the above decisions in a systematic manner.

Stage One: *Technical Decision*

The purpose of this first stage is to determine a (possibly set of) robot(s) that most closely matches the task requirements. The starting point is the application area, or more specifically, the task itself for which the robot is needed. Thus, we need to determine in this stage the following:

1. The application area, e.g., assembly, welding, painting, etc.
2. The task within the application area, e.g., small parts assembly.
3. The task requirements, e.g., precision, speed, load.
4. The robots that most fully satisfy these requirements.
5. Whether human workers can perform the task.
6. Whether to go with robots or humans, if 5 above is true.

Identifying Application:

There is a wide range of applications, across various industries, for which industrial robots may be engaged. Both the application area and the narrow task within that application area should be identified. Thus, within welding, for example, we would identify spot welding and arc welding.

Identifying Task Characteristics:

This phase requires identification of all the task characteristics that influence the decision to employ humans or robots, and the selection among alternative, technically feasible robots. These characteristics will include, for example, the required degree of precision and accuracy; whether it is too hazardous, dangerous, or difficult for humans; and whether significant increases in productivity and/or cost savings would result.

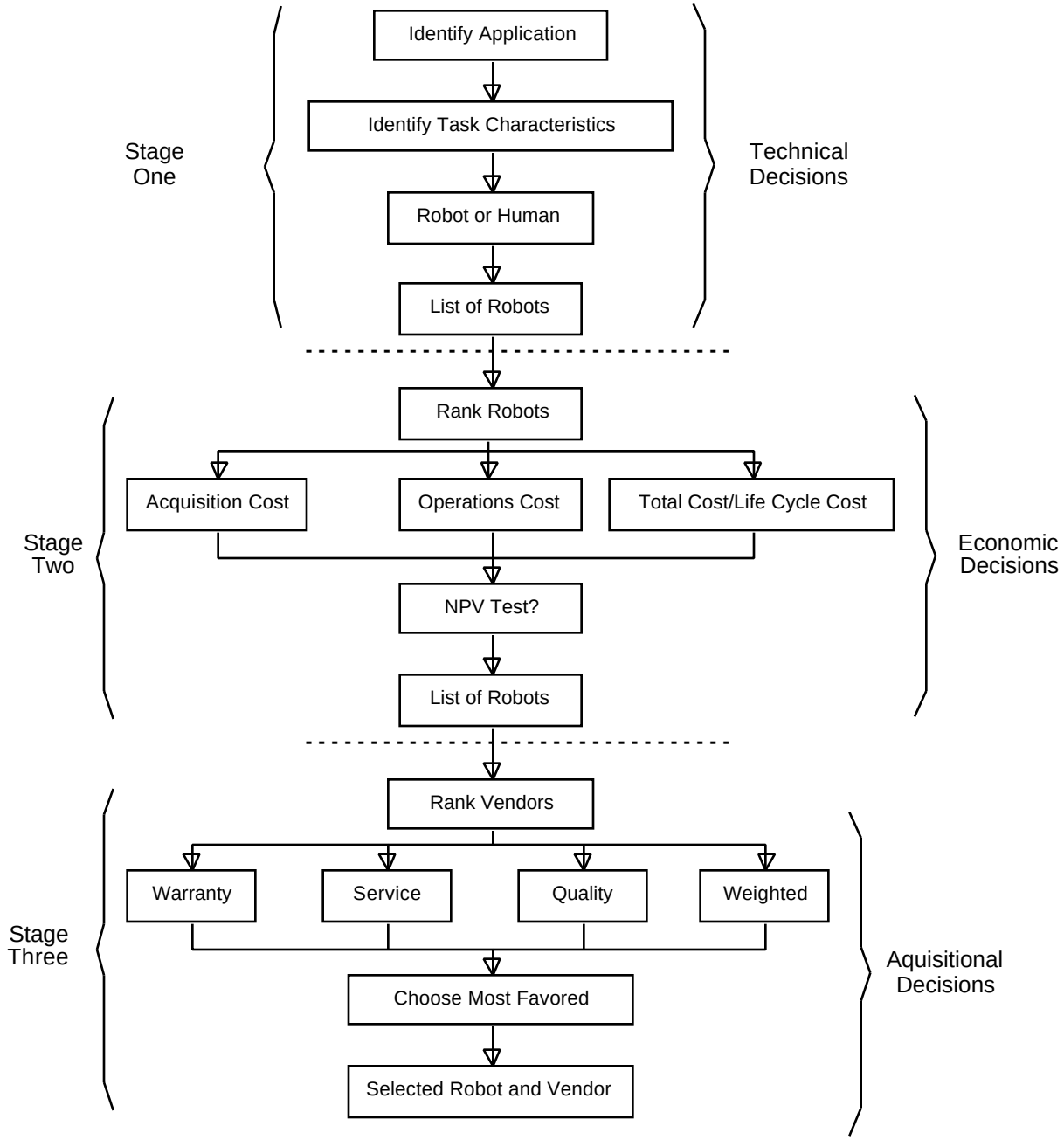


Figure 1. Three-Stage Robot Selection Model

A precise task definition might also require a task classification scheme, more fine-tuned than the one suggested by Ayres and Miller [2]. Since the desired robot is a function of the complexities of the task in question, we suggest the development of a task/robot grid (TRG) to associate specific task characteristics with relevant robot attributes. Let C_{ij} denote value j of task characteristic i, and A_{ij} denote value j of robot attribute i, i=1, 2, ..., m; j=1, 2, ..., n. Here C_{ij} is said to be compatible with A_{ij}, for particular values of i and j, if A_{ij} satisfies C_{ij}. For brevity, we denote this relationship by C_{ij} A_{ij}. Thus specifying task characteristic C_{ij} would automatically call for a robot attribute A_{ij} such that A_{ij} at least satisfies the task requirement C_{ij}, i,j.

Stage Two: *Economic Decision*

This stage can be called a cost justification stage. It utilizes the output from the first stage, which is one or more robots suited to the task at hand. The primary role of this stage is the identification of those robots that make economic sense to invest in; a present value approach is followed by the knowledge-based system to exclude all robots with net present value (NPV) less than their net cost (i.e., purchase price and operating costs). A ranking on the basis of the cost factor is then applied to the remaining robots, if any, i.e., to those passing the economic viability test. Thus, net present value analysis is used to determine whether it is profitable to employ any robot, given its net cost and the economic benefits (e.g., incremental cash flows) expected to accrue as a result of employing the robot to perform the task.

By the end of Stage Two we will have identified a subset of robots that are the most favorable in terms of performance as well as cost. Since a large number of vendors may be available, it is important to be able to get the "best" deal possible. This implies not only a good competitive price, but also acceptable quality, warranties, and a promise of support services.

Stage Three: *Acquisitional Decision*

In Stage three we have to rank, for every vendor, every robot that meets the choice criteria in Stages 1 and 2. The factors that are involved in these rankings are many. For example, Hunt [6] indicates that a certain study revealed the following factors as critical in the purchasing decisions of robots: design, performance, cost, maintenance, warranties, financial terms, and delivery terms. These can conveniently be grouped into four categories: (1) cost (purchase price and operating costs), (2) warranties, (3) quality (performance and design), and (4) service (support, financial and delivery terms). Maintenance is part of operating cost which is accounted for in Stage Two. Quality, services, warranties, and purchase price are the relevant factors in vendor selection. Purchase price has also played a role in the economic decision to determine the viability of each robot. Here, prices are used to compare vendors and rank robots accordingly. Therefore, for each vendor we rank the relevant robots on the basis of purchase price and the other three criteria (quality, warranties, services) and choose the most favorable vendor/robot combination.

THE KNOWLEDGE-BASED SYSTEM

We implemented the prototype knowledge-based system using the CLIPS expert system tool [12]. CLIPS is a forward-chaining rule-based language that resembles OPS5 and ART, two other widely known expert system tools [5]. Developed by NASA, CLIPS has shown an increasing popularity and acceptance by end users [10]. The two main components of this prototype, the knowledge base and the database, are discussed below.

The Knowledge Base

The primary focus of the selection model is the task characteristics, since it is these characteristics that determine what type of robot is needed. This emphasis is reflected in the knowledge base (KB) (or rule base) which captures the knowledge about different task requirements that are needed to justify the use of robots or humans and to specify a particular matching robot or robots. Thus, given certain task characteristics or requirements, the expert system will specify the most suitable robot configurations for performing the task.

The KB also includes knowledge about robot costs and how to manipulate these costs to justify (economically) the use of the respective robots and rank those robots that are considered justifiable. Thus, given operating and purchase costs for each robot, the expert system ranks them on the combined cost criterion. Finally, the KB also includes knowledge to help vendor selection. Subjective multiple criteria are used to compare vendors with associated robots of interest.

As a rule-based shell, CLIPS stores the knowledge in rules, which are logic-based structures, as shown in Figure 2. Figure 3 is a natural English counterpart of the rule in Figure 2.

```
;Rule No 29.
(Defrule find-robots-3rd-pass  “Technical Features”
     ?f <- (robot-2 ?robot)
          (Features (Accuracy ?v11)
                  (Repeat ?v12)
                  (Velocity ?v13)
                  (Payload ?v14)
          )
          (Robot    (ID ?robot)
                  (Accuracy ?v21&:(<= ?v21 ?v11))
                  (Repeat ?v22&:(<= ?v22 ?v12))
                  (Velocity ?v23&:(<= ?v23 ?v13))
                  (Payload ?v24&:(<= ?v24 ?v14))
          )
          =>
          (retract ?f)
          (assert (robot-3 ?robot)
)
```

Figure 2. Example Rule in CLIPS

```
Rule No 29: Finds robots matching given technical
                features.

IF       There is a robot for the application with the
         required grippers,

AND      This robot meets (at a minimum) the following
         technical features: Accuracy, Repeatability,
         Velocity, and Payload as specified by user

THEN     Add this robot to the set of currently feasible
         robots.
```

Figure 3. Example Rule in Natural English

The Database

The database is a critical resource for the ES; all details for robot configurations are contained in the database. The type of information stored for each robot includes:

- Robot class or model
- Performance characteristics
- Physical attributes
- Power characteristics

A full-fledged system would include the following additional information to permit proper comparisons among competing robots.

- Environment requirements
- General characteristics
- Operating costs

Figure 4 shows the information stored in the database for a typical robot using CLIPS syntax. As the figure indicates, each robot has a set of physical and performance characteristics (e.g., type of end effectors, number of axes, repeatability, etc.) and a set of application tasks within its capability. All of this information (and more) is supplied by robot vendors.

```
(Robot (ID RT001)
        (Control S2)
        (Power E)
        (Axes 6)
        (Accuracy .2)
        (Repeat .05)
        (Velocity 145)
        (Payload 6)
        (Effectors adjust-jaw)
        (Jobs ML PT SA EA IN)
        (Vendor “IBM Corp.”)
)

(Vendor (ID VD001)
         (Name “IBM Corp.”)
         (Robot-Info RT001 28500)
            (Service .95)
            (Warranty .8)
            (Quality .83)
)
```

Figure 4. “Facts” Stored as Frames in a CLIPS Database

Also contained in the database are vendor attributes such as service record, warranties, quality, robots carried and purchase prices (see Figure 3). The first three attributes are represented in the database by subjective scores (ratings), on a scale of 0 to 10. A “0” may indicate, for example, an F rating, “10” an A++ rating. This information could come from industry analysts and experts in the robotics industry.

Illustration

In this section we shall provide the results of a consultation with the prototype expert system using actual robotics data obtained from Fisher [4]. The first step in Stage One is to describe the application. For lack of space, we skip the dialogue that allows the decision maker to describe the task and its suitability for robots or human workers. On the basis of the information provided in that dialogue advice will be given as to the choice between a robot solution or human workers for the task.

Next, in Stage Two, economic analysis is performed, using information elicited through a similar dialogue as in Stage one, to determine the economic merit of each robot passing the technical test. This involves calculating a net present value (NPV) for the net incremental benefits resulting from employing robotics in the task under consideration. The NPV is then compared to the net cost (price plus operating cost) of each robot, and only robots whose net cost is less than or equal to the NPV are chosen. If no robot is found to meet this test, the system offers two reasons for the failure:

1. that the task is not worth the required investments in robots, or
2. that the database includes insufficient number of robots.

In the last stage, Stage Three, the system elicits subjective input from the decision maker regarding the importance of vendor attributes, such as service or warranties. Again, the rating is on a scale of 0 to 10; "0" indicates unimportant and "10" maximally important. This information is then used to compute a subjective score for each vendor, by weighting the analyst's ratings of each vendor with the input from the decision maker. Now, robots can be ranked by both price and vendor weighted score. Figure 5 shows the final results of this consultation.

```
Indicate the importance of each of the following,
 on a scale from 0 to 10:
  1. Vendor service quality :7
  2. Vendor warranties :8
  3. Product quality :9
  4. Price :6
.....
Rank by Subjective score (S) or by Price (P)? p

Robot Index    Vendor Name             Price    Score
-----------    -----------             -----    ----
RT005          ASEA, Inc.              $60000    17
RT007          Bendix                  $70000    13
RT011          Cincinnati Milacron     $90000    17
RT006          Automatix, Inc.         $95000    18
RT008          Bendix                  $95000    13
RT016          Kuka                   $125000    13

The net present value of cashflows:   $129036.6
```

Figure 5. Subjective Values and Final Results

LIMITATIONS AND EXTENSIONS

The description of the task as allowed by the current prototype provides only a broad definition of the nature of the task to be performed; it does not provide specific details or "tolerances." For example, to increase the chances of a match, the user may be tempted to supply larger (less tighter) values for positional accuracy and repeatability. However, this may result in a large number of robots being selected and the prototype system allows ranking only through price and vendor attributes. To rank robots for each and every attribute, however, would probably be both unwieldy and unrealistic.

Moreover, a knowledge-based robot selection system should provide a friendly interface that allows the decision maker to input English phrases to describe a particular application; the system would then use the task definition thus provided by the user to suggest applicable robot(s). Therefore, the crucial task of the knowledge-based system would be to make sense out of the English phrases supplied by the decision maker to describe the task. This implies that the knowledge-based system would have to have some natural language processing capability to properly associate the task description with meanings represented in the knowledge base. This, then, would pave the way for processing the applicable knowledge to reach a choice of a set of robots.

Additionally, as mentioned earlier, the database component of the expert system needs to store a wealth of information about a wide range of robots and vendors. This information can be not only very detailed, but volatile as well, as the robot technology advances and as competition among vendor produces changes in vendor profiles. What all this amounts to is that the expert system needs to be able to survive the test of time and handle this voluminous data in a graceful manner. Current expert systems exhibit elementary data management capabilities which are inadequate for this inherently complex database. As Jarke and Vassiliou [7] indicate, a generalized Database Management System (DBMS) integrated in the ES may be necessary to deal with this database effectively. These authors sketch out four ways to extend expert systems with DBMS capabilities, not all of which are relevant in any given circumstances.

CONCLUSION

We presented in this paper a robot selection model based on a three-stage selection process; each stage feeds its output into the next stage until a final robot/vendor combination is selected. We implemented this model in a prototype knowledge-based system using the CLIPS expert system language. The prototype indicated that a full fledged expert system will be practical and can be extremely useful in providing a consistent and credible robot selection approach.

Further work is needed to improve the granularity and natural language processing capability of the system. Also needed is research into possibilities of extending the database management capabilities of the robot selection system by coupling it with a database management system.

REFERENCES

1. Anthony, Tery B. and Hauser, Richard D., "Expert Systems in Production and Operations Management: Research Directions in Assessing Overall Impact," *International Journal of Production Research*, Vol.29, No.12, 1991, pp. 2471-2482.

2. Ayres, Robert U. and Miller, Steven M., *Robotics: Applications and Social Implications,* Ballinger Publishing Co., 1983.

3. Fard, Nasser S. and Sabuncuoglu, Ihsan, "An Expert System for Selecting Attribute Sampling Plans," *International Journal of Production Research*, Vol.3, No.6, 1991, pp. 364-372.

4. Fisher, E.L., "Industrial Robots Around the World," in Nof, S.Y., ed., *Handbook of Industrial Robots*, John Wiley & Sons, New York, 1985. pp. 1305-1332.

5. Giarratano, Joseph and Riley, Gary. *Expert Systems: Principles and Programming*, PWS-Kent Publishing Company, Boston, 1989.

6. Hunt, V. Daniel, *Understanding Robotics*, Academic Press Inc., 1990.

7. Jarke, Matthias and Vassilliou, Yannis, "Coupling Expert Systems With Database Management Systems," *Expert Database Systems: Proceedings of the First International Workshop*, L. Kerschberg, (Ed.), Benjamin/Cummings, 1986, pp. 65-85.

8. Jones, M. S., Malmborg, C. J. and Agee, M. H., "Decision Support System Used for Robot Selection," *Industrial Engineering*, Vol. 17, No. 9, September, 1985, pp. 66-73.

9. Knott, Kenneth, and Getto, Robert D., "A model for evaluating alternative robot systems under uncertainty," *International Journal of Production Research*, Vol. 20, No. 2, 1982, pp. 155-165.

10. Martin, Linda and Taylor, Wendy, *A Booklet About CLIPS Applications*, NASA, Lyndon B. Johnson Space Center, 1991.

11. Malmborg, C., Agee, M. H., Simons, G. R. and Choudhry, J.V., "A Prototype Expert System For Industrial Truck Selection," *Industrial Engineering*, Vol. 19, No. 3, March 1987, pp. 58-64.

12. NASA, Lyndon B. Johnson Space Center, *CLIPS Basic Programming Guide*, 1991.

13. Nof, S.Y., Knight, J. L., JR, and Salvendy, G., "Effective Utilization of Industrial Robots-A Job and Skills Analysis Approach," *AIIE Transactions*, Vol. 12, No. 3, pp. 216- 224.

14. Offodile, Felix O. and Johnson, Steven L., "Taxonomic System for Robot Selection in Flexible Manufacturing Cells," *Journal of Manufacturing Systems*, Vol. 9, No.1, 1987, pp. 77-80.

15. Offodile, Felix O., Lambert, B.K. and Dudek, R.A., "Development of a computer aided robot selection procedure (CARSP)," *International Journal of Production Research*, Vol. 25, No. 8, 1987, pp. 1109-1121.

16. Offodile, F. O., Marcy, W. M. and Johnson, S.L., "Knowledge base design for flexible robots," *International Journal of Production Research*, Vol. 29, No. 2, 1991, pp. 317-328.

17. Pham, D.T. and Tacgin, E., "DBGRIP: A learning expert system for detailed selection of robot grippers," *International Journal of Production Research*, Vol. 29, No. 8, 1991, pp. 1549-1563.

18. Pluym, Ben Van Der, "Knowledge-based decision-making for job shop scheduling," *International Journal of Computer Integrated Manufacturing*, Vol. 3, No. 6, 1990, pp. 354-363.

19. Shah, V., Madey, G., and Mehrez, A., “A Methodology for Knowledge-based Scheduling Decision Support,” *Omega: International Journal of Management Sciences*, Vol. 20, No. 5/6, 1992, pp. 679-703.

20. Trevino, J., Hurley, B.J., Clincy, V. and Jang, S.C., “Storage and industrial truck selection expert system (SITSES),” *International Journal of Computer Integrated Manufacturing*, Vol. 4, No. 3, 1991, pp. 187-194.

21. Wang, M-J., Singh, H. P. and Huang, W. V., “A decision support system for robot selection,” *Decision Support Systems*, Vol. , 1991, pp. 273-283.

THE DESIGN AND IMPLEMENTATION OF EPL: AN EVENT PATTERN LANGUAGE FOR ACTIVE DATABASES[1]

G. Giuffrida and C. Zaniolo

Computer Science Department
The University of California
Los Angeles, California 90024
giovanni@cs.ucla.edu

ABSTRACT

The growing demand for intelligent information systems requires closer coupling of rule-based reasoning engines, such as CLIPS, with advanced Data Base Management Systems (DBMS). For instance, several commercial DBMSs now support the notion of triggers that monitor events and transactions occurring in the database and fire induced actions, which perform a variety of critical functions, including safeguarding the integrity of data, monitoring access, and recording volatile information needed by administrators, analysts and expert systems to perform assorted tasks; examples of these tasks include security enforcement, market studies, knowledge discovery and link analysis. At UCLA, we designed and implemented the Event Pattern Language (EPL) which is capable of detecting and acting upon complex patterns of events which are temporally related to each other. For instance, a plant manager should be notified when a certain pattern of overheating repeats itself over time in a chemical process; likewise, proper notification is required when a suspicious sequence of bank transactions is executed within a certain time limit.

The EPL prototype is built in CLIPS to operate on top of Sybase, a commercial relational DBMS, where actions can be triggered by events such as simple database updates, insertions and deletions. The rule-based syntax of EPL allows the sequences of goals in rules to be interpreted as sequences of temporal events; each goal can correspond to either (i) a simple event, or (ii) a (possibly negated) event/condition predicate, or (iii) a complex event defined as the disjunction and repetition of other events. Various extensions have been added to CLIPS in order to tailor the interface with Sybase and its open client/server architecture.

INTRODUCTION

A growing demand for information systems that support enterprise integration, scientific and multimedia applications has produced a need for more advanced database systems and environments. In particular, active rule-based environments are needed to support operations such as data acquisition, validation, ingestion, distribution, auditing and management —both for raw data and derivative products. The commercial DBMS world has sensed this trend and the more aggressive vendors are moving to provide useful extensions such as *triggers* and *open servers*. These extensions, however, remain limited with respect to functionality, usability and portability; thus, there remains a need for an enterprise to procure a database environment that is (1) more complete and powerful, and thus supports those facilities not provided by vendors, and (2) is more independent from specific vendor products and their releases and helps the enterprise to manage their IMS and cope with multiple vendors, data heterogeneity and distribution.

A particularly severe drawback of current DBMS is their inability of detecting patterns of events, where an event is any of the possible database operation allowed by the system; typically they are: *insertion*, *deletion* and *updating*. Depending on the application, sequences of events

[1] This work was done under contract with Hughes Aircraft Corporation, Los Angeles, California.

temporally related to each other, might be of interest for the user. In addition to basic database events, management of long transactions and deferred actions may be involved in such patterns. Practical examples of such meaningful patterns of events are:

- Temperature which goes down for three consecutive days;
- 2 day delayed deposit for out-of-state checks;
- 30 days of inactivity on a bank account;
- IBM shares increased value consecutively within the last week;
- Big withdrawal from a certain account followed by a deposit for the same amount on another account.

These and similar situations may require either a certain action to take place (e.g.: buy IBM shares) or a warning message to be issued (e.g.: huge transfer of money is taking place.) EPL gives the user the ability to handle such situations.

The purpose of this paper is to propose a rule-based language and system architecture for data ingestion. The first part of this paper describes the language, then the system architecture is discussed.

EPL DESCRIPTION

An EPL program consists of several named modules; modules can be compiled, and enabled independently. The head of each such module defines an **universe** of basic events of interest, which will be monitored by the EPL module. The basic events being monitored can be of the following three types:

```
insert(Rname), delete(Rname), update(Rname)
```

where *Rname* is the name of a database relation.

A module body consists of one or more rules having the basic form:

```
event-condition-list
 →
action-list
```

The left hand side of the rule describes a certain pattern of events. When such a pattern successfully matches with events taking place in the database the set of actions listed in the right hand side are executed.

For instance, assume that we have a database relation describing bank accounts whose schema is: **ACC(Accno,Balance).** We want to monitor the deposits of funds in excess of more than $100,000 into account 00201. In EPL, this can be done as follows:

```
begin AccountMonitor
   monitor update(ACC), delete(ACC), insert(ACC);

   update(ACC(X)),
   X.old_Accno = 00201,
```

```
   X.new_Accno = 00201,
   X.old_Balance - X.new_Balance > 100000
    ->
   write( "suspect withdraw at %s", X.evtime)
end AccontMonitor.
```

The lines “begin AccountMonitor” and “end AccountMonitor” delimit the module. Several rules may be defined within a module (also refereed as “monitor”.) The second line in the example define the *universe* for the module, in this case any update, delete and insert on the **ACC** table will be monitored by this module. Then the rule definition comes. Basically this rule will be fired by an update on the **ACC** table, The variable *X* denotes the tuple being updated. The EPL system makes available to the programmer both the new and the old attribute values of X, these are respectively refereed by means of prefixes “new_” and “old_”. An additional attribute, namely “evtime”, is available. This contains the time when the specific event occurred.

In the previous rule, the event-condition list consists of one event and three conditions. The action list contains a single write statements. In general one or more actions are allowed, these actions are printing statements, execution of SQL statements, and operating system calls.

The previous rule can also be written in a form that combines an event and its qualifying conditions, that is:

```
update(ACC(X),
   X.old_Accno = 00201,
   X.new_Accno = 00201,
   X.old_Balance - X.new_balance > 100000)
 ->
write("suspect withdraw at %s", X.evtime)
```

In this second version, the extent of parentheses stresses the fact that conditions can be viewed as part of the qualification of an event. A basic event followed by some conditions will be called a **qualified event**.

The primitives to monitor sequences of events provided by EPL are significantly more powerful than those provided by current database systems. Thus, monitoring transfers from account 00201 to another account, say 00222, can be expressed in a EPL module as follows:

```
update(ACC(X), X.old_Accno = 00201,
    X.old_Balance - X.new_Balance > 100000)
update(ACC (Y), Y.old_Accno = 00222,
   X.old_Balance - X.new_Balance = Y.new_Balance - Y.old_Balance)
 ->
write("suspect transfer")
```

Thus, the succession of two events, one taking a sum of money larger than $100.000 out of account 00201 and the other depositing the same sum into 00222, triggers the printing of a warning message.

For this two-event rule to fire, the deposit event must *follow immediately* the withdraw event. (Using the concept of *composite events* described later, it will also be possible to specify events that need not immediately follow each other.)

The notion of *immediate following* is defined with respect to the universe of events being monitored in the module. Monitored events are arranged in a temporal sequence (a history). The

notion of universe is also needed to define the negation of *b*, written *!b*, where *b* stands for a basic event pattern. An event *e1* satisfies *!b* if *e1* satisfies some basic event that is not *b*.

We now turn to the second kind of event patterns supported by EPL: **clock events**. Each clock event is viewed as occurring immediately following the event with the time-stamp closest to it. But a clock event occurring at the same time as a basic event is considered to follow that basic event. For example, say that our bank make funds available only after two days from the deposit of a check. This might be accomplished as follows:

```
insert( deposit( Y), Y.type = "check"),
clock( Y.evtime + 2D)
 ->
   action to update balance ...
write( "credit %d to account # %d", Y.amount, Y.account).
```

In this rule the "clock" event makes the rule waiting for two days after the check deposit took place.

The EPL system assumes that there is some internal representation of time, and makes available to the user a way to represent constant values expressing time. In particular, any constant number can be followed by one of the following characters: 'S', 'M', 'H', 'D', which stand, respectively, for seconds, minutes, hours and days. In the previous example the constant *2D* stands for two days. A value for time is built as the sum of functions that map days, hours, minutes and seconds to an internal representation. Thus *2D+24H+61M* will map to the same value of time as *3D+1H+1M*. Thus, EPL rules are not dependent on the internal clock representation chosen by the system. Observe that a clock can only take a constant argument —i.e., a constant, or an expression which evaluates to a constant of type time.

Patterns specifying (i) basic events, (ii) qualified events, (iii) the negations of these, and (iv) clock events are called *simple event patterns*. Simple patterns can always be decided being true or false on the basis of a single event. *Composite event patterns* are instead those that can be only satisfied by two or more events. Composite event patterns are inductively defined as follows. Let $\{p_1, p_2, ..., p_n\}$, n>1 be event patterns (either composite or simple.) Then the following are also composite event patterns:

1. $(p_1, p_2, ..., p_n)$: a sequence consisting of p_1, immediately followed by p_2, ..., immediately followed by p_n.

2. $n{:}p_1$: a sequence of n>0 consecutive p_1's.

3. $*{:}p_1$: a sequence of zero or more consecutive p_1's.

4. $[p_1, p_2, ..., p_n]$: $(p_1, *{:}!p_2, p_2, ..., *{:}!p_n, p_n)$.

5. $\{p_1, p_2, ..., p_n\}$ denotes that at least one of the p_i must be satisfied for $(1 \le i \le n)$.

Using the composite patterns we can model complex patterns of events. For instance, we can be interested to "the first sunny day after at least three consecutive raining days." Assuming that we have a "weather" table which is updated every day (the "type" attribute contains the weather type for the specific day.) Our rule will be:

```
[ (3:insert( weather(X), X.type = 'rain')),
  insert( weather(Y), Y.type = 'sun')]
->
```

```
write("hurrah, eventually sun on %d\n", Y.evtime).
```

This rule fires even though between the three raining days and the sunny one there are days whose weather type is different from “rain”. In case we are interested to “the first sunny day immediately following three or more raining days,” we should rewrite the rule as:

```
 (3:insert( weather(X), X.type = ‘rain’),
  *:insert( weather(Z), Z.type = ‘rain’),
  insert( weather(Y), Y.type = ‘sun’))
->
 write("hurrah, eventually sun on %d\n", Y.evtime).
```

Note the different use of the *relaxed sequence* operator “[...]” in the first case and the *immediate sequence* one “(...)” in the second rule. By combination of the available composite operators, EPL can express very complex patterns of events.

EPL ARCHITECTURE

EPL’s architecture combines CLIPS with a DBMS that supports the *active* rule paradigm. The current prototype runs on Sybase [3] and Postgres [10] — but porting to other active RDBMS [6, 8] is rather straightforward.

Sybase rule system presents some drawbacks like:

- Only one trigger is allowed on each table for each possible event;
- The trigger fires immediately after the data modification takes place;
- Trigger execution is set oriented, which means the triggers are executed only once regardless the number of tuples involved in the transaction;
- Only SQL statements are allowed as actions.

EPL tries to both overcome to such limitations and allow the user to model patterns of meaningful events.

Event Monitoring Mechanism

For each event which needs to be monitored by an EPL rule a trigger and a table (also referred as Event monitor Relation, ER) have to be created on Sybase. The trigger fires on the event which can be either an insert, delete or, update. This trigger will copy the modified tuple(s) into the corresponding ER.

As an example, say that we want to monitor the insertion on the **ACC** relation previously defined. As soon as the EPL monitor is loaded into the system the fact *(universe acc_mon i acc)*, will be asserted into the CLIPS working memory. This new fact triggers a CLIPS rule which creates a new Sybase relation called “ERacc_ins” having the following schema: *(int accno, int balance, int evtime)*. Moreover, a Sybase trigger is created by sending the following command from CLIPS to Sybase:

```
create trigger ETacc_ins on acc for insert as
begin
  declare @d int
```

```
  select @d = datediff( second, '16:00:00:000 12/31/1969', getdate())
  insert ERacc_ins select distinct *, @d from inserted
end
```

The event time is computed as the number of seconds since the 4pm of 12/31/1969. This is a standard way to represent time on UNIX systems. Any ER name starts with the prefix “ER” followed by both the monitored table name and the type of event. The correspondent trigger has “ET” as prefix.

As later explained, the module **EPL-Querier** performs the communication between CLIPS and Sybase.

EPL rules as finite-state machines

As previously discussed, any EPL rule is modeled by a finite state automaton which is implemented by a set of CLIPS rules. Transitions between automaton states take place when the following conditions occur:

- a new incoming event satisfies the current pattern;
- a pending clock request reaches its time;
- a predicate is satisfied.

On a transition the automaton can take one of the following actions:

- Move to the next **Acceptance** state where it will wait for the next event;
- Move to a **Fail** state. Here, the automaton instantiation, with relative information, is removed from the memory;
- Move to a **Success** state. Here, the actions specified in the right hand side of the rule are executed.

Each EPL rule is transformed to a set of CLIPS rules which implement its finite state machine. Several instantiations of the same EPL rule can be active at the same time.

Architecture overview

EPL is basically built on top of CLIPS in two ways: 1) Some new functions, implemented in C, have been added to CLIPS in order to support EPL; 2) CLIPS programs have been written to implement the EPL rule execution system. Figure 1 depicts the entire EPL system.

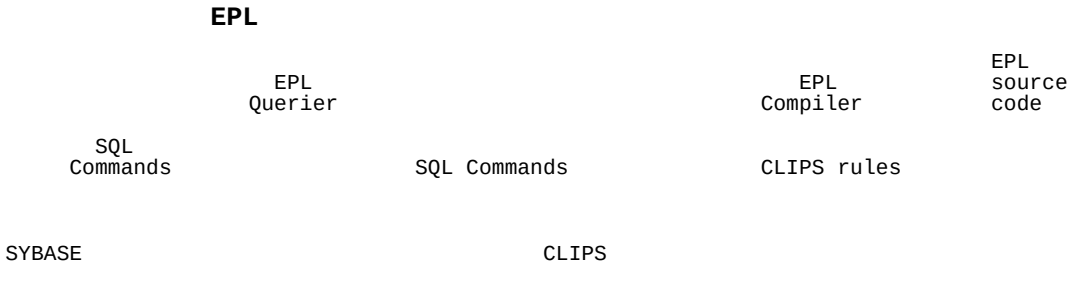


Figure 1. EPL Architecture

EPL-Compiler is a rule translator, it takes an EPL program as input and produces a set of CLIPS rules which implement the EPL program. **EPL-Polling**, at regular intervals, transfers the ERs from Sybase to the CLIPS working memory. This process requires some data type conversion in order to accommodate Sybase tuples as CLIPS facts. The **EPL-Querier** sends SQL commands to Sybase server when either (1) Sybase triggers or ERs have to be created (or removed) or when (2) an SQL command is invoked on the action side of an EPL rule. The **EPL-User-Interface** accepts commands from the user and produces output to either the screen or a file. At low level, EPL commands are executed by asserting a fact in the CLIPS working memory. Such an assertion triggers a rule which executes the desired command. This loose coupling allows an easy design of the user interface whose only task is to insert a new fact depending on the user action.

EPL-Demons is a CLIPS program which implements the EPL rule execution system. Basically this set of CLIPS rules monitors the entire EPL execution. The EPL demons, together with the CLIPS rules produced by the EPL-compiler, form the entire CLIPS program under normal execution time. The CLIPS facts on which these rules work are those periodically produced by the EPL-Polling, and those asserted by the EPL user interface as a consequence of user actions.

CONCLUSIONS

This document has described the design and the architecture of EPL, a system which provides sophisticate event pattern monitoring and triggering facilities on top of commercial active databases, such as SYBASE. EPL implementation is based on CLIPS and the design of an interface between SYBASE and CLIPS represented one of the most critical tasks in building EPL. EPL rules are translated into a set of CLIPS rules which implement the finite state machine needed to execute such EPL rules.

This paper provided an overview of the language definition and a brief description of the system implementation neglecting various implementation details for lack of space.

Future work is required to provide language extensions and interfacing with other active database systems.

ACKNOWLEDGMENTS.

This report builds upon a previous one authored by S. Lau, R. Muntz and C. Zaniolo. The authors would also like to thank Roger Barker for several discussions on EPL.

REFERENCES

1. ISO-ANSI Working Draft of SQL-3.

2. R. Coleman. “Pulling the Right Strings”, Database Programming and Design, Sept. 1992, pp. 42-49.

3. Sybase Inc. Sybase's Reference Manual.

4. C.L.Forgy. “RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem” on Artificial Intelligence 19, 1982.

5. “CLIPS User's Guide”, Artificial Intelligence Section.

6. G.Koch, “Oracle: the Complete Reference,” Berkeley, Osborne, McGraw-Hill, 1990.

7. J.L Lassez, M.J. Maher, K. Marriot. “Unification Revisited”, Lecture Notes in Computer Science vol.306, Springer-Verlag, 1986.

8. J.Melton, A.R.Simon. “Understanding the new SQL: A Complete Guide,” San Mateo, California, Morgan Kaufmann Publishers, San Francisco, California, 1993.

9. S.Naqvi, S.Tsur “A Logical Language for Data and Knowledge Bases,” Computer Science Press, New York, 1989.

10. J. Rhein, G. Kemnitz, POSTGRES User Group, The POSTGRES User Manual, University of California, Berkeley.

11. M. Stonebraker, The POSTGRES Storage System, Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

12. M. Stonebraker, The integration of rule systems and database systems IEEE Transactions on Knowledge and Data Engineering, October 1992.

EMBEDDING CLIPS INTO C++

Lance Obermeyer
Tactical Simulation Division
Applied Research Laboratories
University of Texas at Austin

Daniel P. Miranker
Department of Computer Sciences
University of Texas at Austin

This paper describes a set of extensions to CLIPS that achieve a new level of embeddability with C++. These extensions are the include construct and the defcontainer construct.

The include construct, (include <c++-header-file.h>), allows C++ functions to be embedded in both the LHS and RHS of CLIPS rules. The defcontainer construct, (defcontainer <c++-type>), allows the inference engine to treat C++ class instances as CLIPS deftemplate facts.

The header file in an include construct is precisely the same header file the programmer uses for his own C++ code, independent of CLIPS. Consequently, existing C++ class libraries may be transparently imported into CLIPS. These C++ types may use advanced features like inherience, virtual functions and templates.

The implementation has been tested with several class libraries including Rogue Wave Software's Tools.h++, GNU's libg++, and USL`s C++ Standard Components.

EXPERT SYSTEM SHELL TO REASON ON LARGE AMOUNT OF DATA

G. Giuffrida
Computer Science Department
The University of California Los Angeles
giovanni@cs.ucla.edu

ABSTRACT

The current DBMSs do not provide a sophisticated environment to develop rule based expert system applications. Some of the new DBMSs come along with some sort of rule mechanism, these are *active* and *deductive* database systems. However, both of these are not featured enough to support full implementation based on rules. On the other hand, current expert system shells do not provide any link with external databases. That is, all the data are kept into the system working memory. Such working memory is maintained in main memory. For some applications the reduced size of the available working memory could represent a constraint for the development. Typically these are applications which require reasoning on huge amounts of data. All these data do not fit into the computer main memory. Moreover, in some cases these data can be already available in some database systems and continuously updated while the expert system is running.

This paper proposes an architecture which employs knowledge discovering techniques to reduce the amount of data to be stored in the main memory, in this architecture a standard DBMS is coupled with a rule-based language. The data are stored into the DBMS. An interface between the two systems is responsible for inducing knowledge from the set of relations. Such induced knowledge is then transferred to the rule-based language working memory.

INTRODUCTION

Current trends in database research area are about making smarter and more friendly current DBMS. Traditionally, a DBMS is mostly a repository of information to be later retrieved by an ad-hoc query language. If some data are currently unavailable in the system, or the query is not properly issued, the system can return either an error or a non-answer. This is not really the case when a specific question is asked to a human expert, in this case the expert digs in his or her memory looking for an answer, if no proper answer is found the expert perhaps starts looking for alternative, and approximate, responses. Actually, the expert uses his or her knowledge in a sort of "cooperative way" trying to make the user as more satisfied as possible. To answer such a query the expert employs his or her knowledge in a much more productive way than performing a simple linear scan over the known set of tuples in his or her mind: **meta-knowledge** was fruitfully employed in order to formulate the answer.

In an artificial system such meta-knowledge can be known a priori (read: Tied up to the system at database design time) or induced by an already available set of relations. The purpose of *induced knowledge* [3, 15] is to describe the current state of the database by means of sets of rules; a rule induction algorithm is responsible to *discover* such induced knowledge. Basically, induced rules are oriented to summarize the way the different attributes are related to each other. Artificial reasoning can be built on top of this meta-knowledge.

This paper proposes an architecture which integrates two public domain systems, namely Postgres and CLIPS, into a unique environment. The proposed architecture offers a shell for developing expert systems able to work on large amounts of data. Database relations are stored on Postgres while the artificial expert is implemented on top of CLIPS. The artificial expert can specialize the rule induction algorithm in order to focus the induction only on the necessary data.

Postgres comes along with a rule system to implement active database features. By the way, implementing an expert system using only Postgres has some drawbacks. The Postgres rule mechanism has been designed for triggering over updates on the database. We always need to perform a secondary storage update in order to fire a Postgres rule. This can be too expensive in many cases when a little fact stored in main memory can perform the same task. Moreover, the rule system in Postgres does not implement all the facilities of a special purpose rule-based language such CLIPS.

On the other side CLIPS does not provide any access to an external database. This can cause serious limitations in designing expert systems reasoning on large amounts of data stored on a database.

CURRENT SYSTEMS

Latest developments in databases are oriented to enrich traditional systems with some sort of rule-based mechanism. Novel additional features are so added to traditional DBMS by means of rules. New classes of systems have appeared on the market, they are: *Active* [6, 7, 14] and *Deductive* [8, 10, 11] databases. The following sections give some details about these new systems. The suitability of those as expert system shell is also investigated. Advantages and drawbacks for each system are drawn. Also, the limitations of current rule-based languages as developing shell for expert system oriented to reason on large amounts of data are discussed.

Active Database

More and more active database systems are everyday appearing on the market. The transition of such system from research stages to the market was fast. They materialized in systems like Sybase [7], Postgres [13], Starburst [6], and more. However, the active feature of databases is still under investigation. Additional research still needs to be done.

The active feature extends the traditional systems with a rule-based mechanism oriented to allow implementation of *demons*. The purpose of these demons is basically to overview operations on the database and, when certain conditions are satisfied, invoke a corrective action over the database. The system can so be extended with sort of animated entities; data integrity can be enforced by means of active rules.

Typically an active rule has the following format:

```
on <event>
if <condition>
then <action>
```

This rule model is also referred as E-C-A (Event-Condition-Action) rule. Simpler rule model can utilize an E-A pattern. **event** is one of the possible operations oriented to tuple handling; typical events are: *insert*, *delete*, *update* or, *retrieve*. The specified **condition** is a condition which should hold on the data involved in the current event in order to fire the rule. **action** is the set of actions that take place when the specified condition is satisfied.

In the following a Postgres rule is reported as example of active rule (copied from [12]):

```
define rule r1 is
  on replace to EMP.salary
     where current.name = 'Joe'
  do replace EMP (salary = new.salary)
```

```
        where EMP.name = 'Sam'
```

Rule r1 will fire every time the salary of Joe is updated. For such update the salary of Sam will be set to the same value of the new salary of Joe. In other words, Sam will always get the same salary of Joe. Note that the inverse is not true, the same rule will not take care to update Joe's salary when Sam's salary is replaced by some new value.

Evaluation of database active feature is based upon (1) the domains of possible events which can be monitored; (2) the formalism to specify the condition; (3) the language to execute actions. Other features, like rule execution model, have also to be considered for a sound evaluation. Postgres [13], is actually one of the most advanced active databases currently available.

Active database systems present some drawbacks as expert system shells. The most significant ones are hereafter listed:

- Firing only upon database events, no "Working memory" is available. All the data are kept in the secondary memory storage. Main memory usage might, in several cases, be preferable to a secondary memory one.

- The language to specify actions is usually restricted to the database domain. Typical rule-based languages allow on the right-hand side complex operations (print, while loop, user inputs, etc.) to be performed.

- Not well defined rule execution model is available.

Other criticisms to these systems are about the rule redundancy required in some cases. A set of sort of complementary rules might be defined in order to ensure some data integrity. For instance, let us suppose we want to make sure that each employee works for one and only one department. A set of rules needs to be defined in order to enforce this constraint: One will take care when a new employee is hired, another when the employee is transferred, another when a department is dropped, and so on. This is not really the case of an equivalent rule-based implementation where a single rule can handle all the cases. The different behavior between active DBMS and RBL is actually based on different architectures. In an active database system the rules are tied up to the database operations, so a trigger must be defined for each possible operation on the interested data. In a rule-based system, rules are fired straight from the data in the working memory. The operations which update the data are hidden to the triggered rules. This model reduces the number of required rules. Moreover, the maintenance cost for a set of rules is of course higher than the one for a single rule.

The previous considerations should convince the reader that implementing an expert system on top of an active database is not a straightforward task; they lack the power for that purpose.

Deductive Databases

Another challenging and relatively new research direction in database area is about *Deductive databases*. These systems are essentially based on a Prolog-like development language. Some of these are LDL++ [10], NAIL! [9] and CORAL [11]. Here facts can be either stored on main memory or secondary memory storage. However, in the author's opinion, deductive databases are still not at a stable stage. Further investigation needs to be spent on those. This belief is also enforced from the lack of any commercial deductive database system on the market at the time of this writing.

Development of deductive databases raised several both theoretical and practical issues that are still looking for answers. Research is still in progress in this direction.

Some of the most significant points of deductive systems are:

- Prolog-like rule-based development environment;
- Recursive rule invocation allowed;
- Easily definable virtual relations (*views* in database jargon);
- Hybrid forward/backward chaining rule execution model. The method chosen depends on the current task to be performed. It is chosen automatically by the system itself. The user does not have any responsibility on that;
- Restricted usage of negation. Only certain classes of negation (namely *stratified*) are allowed. Basically, no negation on recursive calls can be used.
- Ability of interfacing with already available DBMSs.

Deductive databases are definitively a proper environment for developing expert reasoning on large amounts of knowledge. However, as already said, they still need some time before reaching a steady stage.

Expert System Shells

Somewhat current expert systems shells present the opposite problem of the previously discussed database systems. In this case all the facts are kept into the system working memory. No connection with any external storage system is provided. This can be a constraint difficult to be taken around when huge amount of knowledge is employed as basis for the reasoning. In this case the main memory cannot be big enough to accommodate this large number of tuples. In another possible scenario, reasoning has to be developed on data which are already stored on some database systems. Perhaps these data are still evolving while the artificial reasoning is performed. The only possible solution to this problem is represented by a periodic transfer of the new updated data from the database to the expert system shell.

Current solutions to this problem are oriented to create an ad-hoc interface between an expert system shell and a DBMS. Such solutions are often properly tailored for the application itself. They do not provide a generic solution to the problem.

KNOWLEDGE INDUCTION ALGORITHM

The process of knowledge induction [2, 15] aims to build useful semantic data starting from the available relations in a database. In this section, an overview of the knowledge induction algorithm is given while keeping an eye open on our specific implementation.

Somehow the knowledge induction process aims to “capture” the semantics of the stored data and model it by means of *induced rules*. Different forms of rule may exist, we are mainly interested in rules which correlate two attributes between themselves. The correlation of interest for us is the definition of a domain for the first attribute which identifies a unique value for the

second attribute. We are interested to the *range form* which is: "*if* $x_1 \leq X \leq x_2$ *then* $Y = y$ *with Probability P and Coverage C."* The **probability** value expresses the *certainty degree* of the given rule while the **coverage** refers to the number of tuples used to build such rule. Additional details can be found in [15], [2] and [3]. Several different forms and attributes can be combined to express induced rules.

The set of induced rules represents part of the knowledge about the given databases. Storing these rules most probably requires less space than the corresponding table format. Induced rules have to change dynamically reflecting the database updates. Every update operation on the database has to affect the set of induced rules accordingly.

In the proposed system induced rules are stored as CLIPS facts whose form is:

```
(ik <arg1> <lowerBound> <upperBound> <arg2> <value> <prob> <cover>)
```

The first atom is to classify the set of facts describing the induced knowledge. The remaining atoms are the information about the specific induced rule. The set of all these facts forms the induced knowledge. Semantics of most of the arguments is straightforward. **prob** is the probability factor and **cover** is the number of tuples covered by the rule (look at [1] for a complete description of these parameters.) For instance, the rule "*if 62000 ≤ SALARY ≤ 85000 then TYPE = 'MANAGER' with Probability=0.6, Coverage=3"* is stored in the following CLIPS fact:

```
(ik SALARY 62000 85000 TYPE MANAGER 0.6 3)
```

As an example, let us suppose to have the following table, namely *emp*, in our Postgres database:

NAME	TYPE	SALARY
John	STAFF	30000
Mary	MANAGER	62000
Eva	STAFF	32500
Bob	MANAGER	85000
Mark	MANAGER	76000
Judy	MANAGER	83000
Kia	STAFF	56000
Tom	MANAGER	50000
Phil	STAFF	54000

Table 1. EMP

By applying the induction algorithm on these tuples the set of CLIPS facts will be:

```
(ik emp.SALARY 62000 85000 emp.TYPE MANAGER 0.8 4)
(ik emp.SALARY 50000 50000 emp.TYPE MANAGER 0.2 1)
(ik emp.SALARY 30000 32500 emp.TYPE STAFF 0.5 2)
(ik emp.SALARY 54000 56000 emp.TYPE STAFF 0.5 2)
```

The induced rule schema is only a first attempt which is used more as framework for this paper. A real design should take into more consideration other parameters to be included in the schema.

The induced rules need to be dynamically updated reflecting operations performed over the database. For instance, let us suppose the following new tuple is inserted into the system:

```
Betty  MANAGER  60000
```

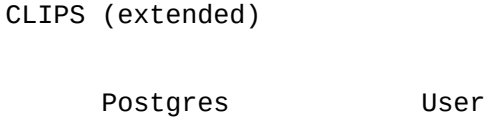
The knowledge base has to be updated to the following set of facts:

```
(ik emp.SALARY 60000 85000 emp.TYPE MANAGER 0.83 5)
(ik emp.SALARY 50000 50000 emp.TYPE MANAGER 0.16 1)
(ik emp.SALARY 30000 32500 emp.TYPE STAFF 0.5 2)
(ik emp.SALARY 54000 56000 emp.TYPE STAFF 0.5 2)
```

The new tuple affected the values in the first fact, the lower bound, probability and coverage values changed accordingly. Similar changes will take place in case of deletion or updating.

PROPOSED SYSTEM OVERVIEW

In the proposed system architecture, Postgres and CLIPS are being integrated. The interface between them represents the key point for the design. An expert system can be developed on top of CLIPS. This latter is extended with features to make available summarized data stored in Postgres. From an expert system shell's point of view the new system can be seen as in the following figure:



Postgres presence is actually hidden to the end-user. The application built on top of CLIPS takes advantage of the meta-knowledge induced from the stored database. The expert system being developed on CLIPS has to be aware of the following:

- Database schema;
- Database statistics: number of tuples, number of induced rules, rule coverage, popularity, probability, etc.

CLIPS has to be extended with dedicated constructs to inquire for such information. Depending on both the database information and the application being implemented, the meta-knowledge extraction process is properly tailored and executed. From CLIPS several actions can be taken in order to model the induction algorithm to best fit the current application.

The knowledge induction process executes in a dynamic fashion, that is, once the rule schema has been defined, any update on the monitored Postgres relations will be reflected on the induced rules. The rule induction manager then propagates these induced rules to the CLIPS working memory in the fact form previously discussed. Eventually, these facts trigger some CLIPS rules.

The interface between CLIPS and Postgres then can be defined by the following set of functions available on the CLIPS side:

- Inquire for the database schema;
- Inquire for database statistics;

- New rule induction schema definition: $R_1.X \rightarrow R_2.Y$, where R_1 and R_2 are relations while X and Y are attributes;

- Modify rule induction parameters (cut-off factor, etc.);

- Retrieve current rule induction schema;

- Remove induced rule;

The system extends the set of built-in CLIPS functions to include these new services.

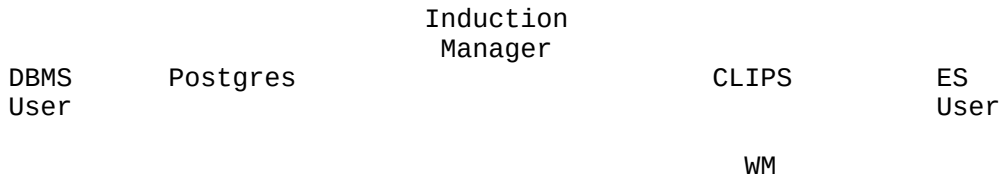
DESIGN

The system will be presented to the expert system developer as a more featured shell which includes ability to access summarized data. The key points of all design are essentially the following:

- Interface between CLIPS and Postgres;

- Induced knowledge management.

The system architecture is the following:



The *Induction Manager* is responsible for the communication between CLIPS and Postgres. CLIPS utilizes services offered by this module to tailor the induction algorithm to its needs. Responses from Postgres to CLIPS are also passed through the induction manager. The link between Postgres and WM (the CLIPS working memory) stands for a straight access from Postgres to CLIPS working memory. That is, once the rule schema has been defined, updates on Postgres will result in updates on the induced rules. These rules are then written straight to CLIPS' working memory without requiring the intervention of the induction manager. In other words, the induction manager can be seen as a set of function to be invoked from CLIPS, while the link from Postgres to WM represents an asynchronous flow of data toward CLIPS. Once the rule induction schema has been defined, this flow of data will be dynamically maintained, that is, any update to any monitored Postgres relation can cause some rule induction instances updates, these will be promptly propagated to CLIPS through this link. As better explained in the following, actually the induction manager is scattered between Postgres and CLIPS. Some Postgres triggers and CLIPS rules (together with some external Postgres procedures) form the complete block.

The entire system works in a dynamic fashion. That is, at the same time the CLIPS application is running people can keep on working on Postgres. Postgres updates affect the set of induced rules on the CLIPS side. Such updates can trigger some CLIPS rules to execution. Under this light the

entire system can be seen as an extension of the rule-based mechanism of Postgres: Some Postgres rule activations eventually trigger some CLIPS rules.

CONCLUSIONS

This paper discussed the design of a system where expert shell techniques are combined together with knowledge discovering techniques. Two public domain systems, namely CLIPS and Postgres, have been combined into a unique one. Purpose of this design is to provide an expert system shell with ability to reason over a large amount of data. Current expert system shells lack the ability of accessing data stored on external devices. On the other side database technology is not yet well refined to provide a featuring shell to develop expert systems. Proper combinations of these techniques may lead to interesting results.

The possible applications for such a system are those where reasoning on large volume of data are required. For instance, think about the complexity of reasoning on the millions of customers of a phone company or a frequent flyer program, in this case the complexity is due to the enormous amount of data to reason on.

REFERENCES

1. W.W.Chu, “Class notes CS244-Spring 1994”, University of California in Los Angeles, 1994.

2. W.W.Chu, R.Lee, “Semantic Query Processing Via Database Restructuring,” Proceedings from the 8th International Congress of Cybernetics and Systems, 1990.

3. W.W.Chu, R.Lee, Q.Chen, “Using Type Inference and Induced Rules to Provide Intensional Answers,” Proceedings of the 7th International Conference on Data Engineering, 1991.

4. C.L.Forgy, “RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem,” Artificial Intelligence 19, 1982.

5. J.C.Giarratano, “CLIPS User's guide,” Artificial Intelligence Section, Lyndon B. Johnson Space Center, June 1989.

6. L.Haas, W.Chang, G.M.Lohman, J.McPherson, P.F.Wilms, G.Lapis, B.Lindsay, H.Pirahesh, M.Carey, E.Shekita, “Starburst midflight: as the dust clears,” IEEE Transactions on Knowledge and Data Engineering, March 1990.

7. D.McGoveran, C.J.Date, “A Guide to SYBASE and SQL Server,” Addison-Wesley Publishing Company, 1992.

8. J.Minker “Foundation of Deductive Databases and Logic Programming,” Morgan-Kaufmann, Los Altos, CA, 1988.

9. K.Morris, J.Ullman, A.Van Gelder, “Design overview of the NAIL! system,” proceedings of the 3rd Int. Conference on Logic Programming, Springer-Verlag LNCS 225, New York, 1986.

10. S.Naqvi, S.Tsur, “A Logical Language for Data and Knowledge Bases,” Computer Science Press, 1989.

11. R.Ramakrishan, D.Srivastava, S.Sudarshan, “CORAL: A Deductive Database Programming Language,” Proc. VLDB '92 Int. Conf. 1992.

12. J. Rhein, G. Kemnitz, POSTGRES User Group, The POSTGRES User Manual, University of California, Berkeley.

13. M. Stonebraker, “The POSTGRES Storage System,” Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

14. M. Stonebraker, “The Integration of Rule Systems and Database Systems,” IEEE Transactions on Knowledge and Data Engineering, October 1992.

15. Quinlian, J.R., “Induction Over Large Data Bases,” STAN-CS-79-739, Stanford University, 1979.

AI & WORKFLOW AUTOMATION: THE PROTOTYPE ELECTRONIC PURCHASE REQUEST SYSTEM

Michael M. Compton
compton@ptolemy.arc.nasa.gov
(415) 604-6776

Shawn R. Wolfe
shawn@ptolemy.arc.nasa.gov
(415) 604-4760

AI Research Branch / Recom Technologies, Inc.
NASA Ames Research Center
Moffett Field, CA 94035-1000

ABSTRACT

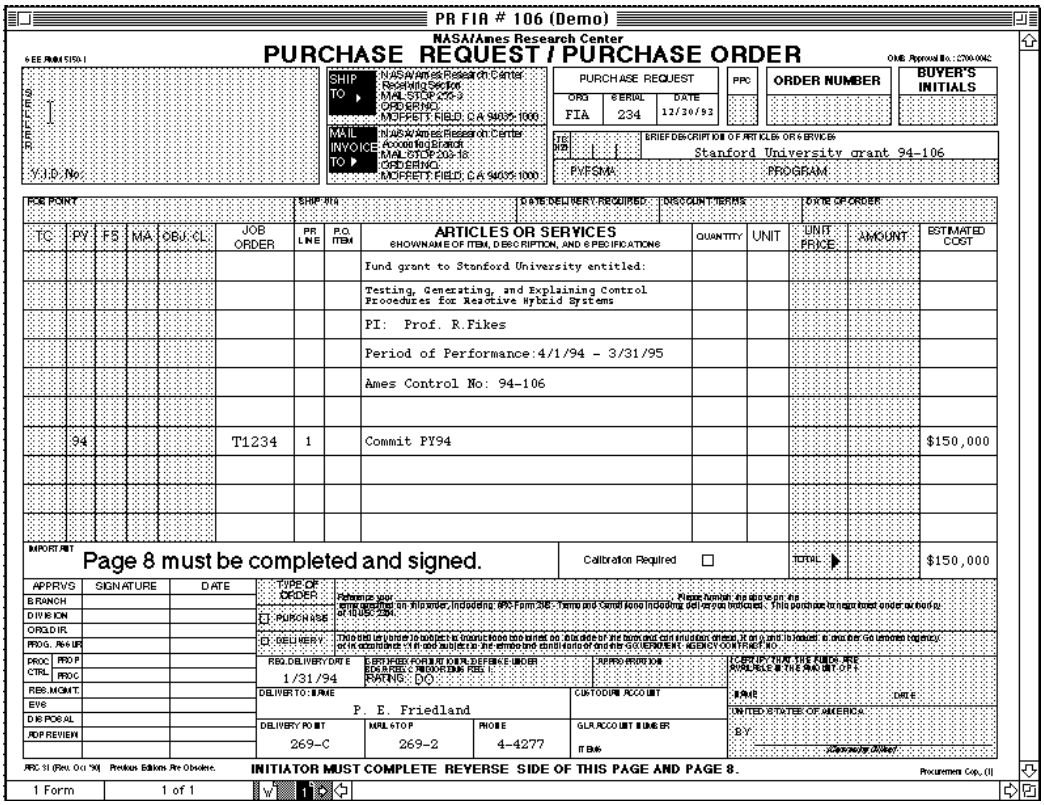
Automating "paper" workflow processes with electronic forms and email can dramatically improve the efficiency of those processes. However, applications that involve complex forms that are used for a variety of purposes or that require numerous and varied approvals often require additional software tools to ensure that 1) the electronic form is correctly and completely filled out, and 2) the form is routed to the proper individuals and organizations for approval. The Prototype Electronic Purchase Request (PEPR) system, which has been in pilot use at NASA Ames Research Center since December 1993, seamlessly links a commercial electronic forms package and a CLIPS-based knowledge system that first ensures that electronic forms are correct and complete, and then generates an "electronic routing slip" that is used to route the form to the people who must sign it. The PEPR validation module is context-sensitive, and can apply different validation rules at each step in the approval process. The PEPR system is form-independent, and has been applied to several different types of forms. The system employs a version of CLIPS that has been extended to support AppleScript, a recently-released scripting language for the Macintosh. This "scriptability" provides both a transparent, flexible interface between the two programs and a means by which a single copy of the knowledge base can be utilized by numerous remote users.

INTRODUCTION

The Procurement Division at NASA Ames Research Center processes up to twenty thousand purchase requests (PRs) every year. These PRs, which all use a common form, are used to procure virtually anything used at the Center: computers, hazardous chemicals, office equipment, scientific instruments, airplane parts, and even funding for external research projects. PRs can be submitted by any civil servant employee at the Center, and must be approved by anywhere from three to twenty different individuals and offices. The average time required to submit a PR and obtain the necessary approvers' signatures is eighteen business days. Worse yet, approximately half of the PRs that are submitted are either incorrectly filled out, lack some required additional paperwork, or are routed to the wrong group for approval and must be returned to the originator. This not only delays procurement of the requested items but also burdens the system with a significant amount of paper flowing in the "wrong direction". In addition, the paper system lacks any mechanism for tracking a submitted PR, so people who originate these purchase requests often try to track them manually by picking up the telephone and calling around until they find where the PR is in the approval process. This, along with the numerous "walk-though" PRs, contribute significantly to the delays involved in processing the requests.

In 1991, the AI Research Branch at NASA Ames undertook a “weekends and evenings” effort to see whether a knowledge-based system, in conjunction with other advanced computing tools, could help expedite the process by which purchase requests are submitted, routed, and approved. The resulting system, called the Prototype Electronic Purchase Request system (PEPR), combines a commercial electronic forms package with a knowledge-based system that both ensures that the submitted forms are correct and complete, and generates an electronic “routing slip”, based on the content of various fields on the form, that reflects the approvals that particular PR requires. The PEPR system currently operates in a Macintosh environment and takes advantage of several new collaborative features of the latest release of the Macintosh OS, including digital signatures and “OS-level” electronic mail.

The system is now being used by several different groups at Ames to process a particular class of PR, namely those that apply to the funding of external research at colleges and universities. Initial results indicate that the system can dramatically reduce the time required to originate and process PRs and their supporting paperwork by ensuring that PRs entering the system are error-free and automatically routing them to the proper individuals. The system also provides a tracking capability by which the originator of a PR can query the system about the status of a particular PR and find out exactly where it is in the approval process.


PR FIA # 106 (Demo)

NASA/Ames Research Center

PURCHASE REQUEST / PURCHASE ORDER

PURCHASE REQUEST ORG: FIA SERIAL: 234 DATE: 12/30/93

PPC | ORDER NUMBER | BUYER'S INITIALS

SHIP TO: NASA/Ames Research Center, Receiving Section, MAIL STOP 205-3, ORDERING, MOFFETT FIELD, CA 94035-1000

MAIL INVOICE TO: NASA/Ames Research Center, Accounting Branch, MAIL STOP 205-18, ORDERING, MOFFETT FIELD, CA 94035-1000

BRIEF DESCRIPTION OF ARTICLES OR SERVICES: Stanford University grant 94-106

P/F/SMA | PROGRAM

TC	PY	FS	MA	OBJ. CL.	JOB ORDER	PR. LINE	P.O. ITEM	ARTICLES OR SERVICES (SHOW NAME OF ITEM, DESCRIPTION, AND SPECIFICATIONS)	QUANTITY	UNIT	UNIT PRICE	AMOUNT	ESTIMATED COST
								Fund grant to Stanford University entitled:					
								Testing, Generating, and Explaining Control Procedures for Reactive Hybrid Systems					
								PI: Prof. R.Fikes					
								Period of Performance:4/1/94 - 3/31/95					
								Ames Control No: 94-106					
	94				T1234	1		Commit PY94					$150,000

IMPORTANT: Page 8 must be completed and signed. Calibration Required ☐ TOTAL $150,000

APPRVS	SIGNATURE	DATE
BRANCH		
DIVISION		
ORG.DIR.		
PROG. APP.		
PROC. CTRL	PRO P	
	PROC	
REQ. MGMT.		
EVS		
DIS POSAL		
ADP REVIEW		

TYPE OF ORDER: ☐ PURCHASE ☐ DELIVERY

REQ. DELIVERY DATE: 1/31/94 | RATING: DO

DELIVER TO: NAME: P. E. Friedland | CUSTODIAL ACCOUNT

DELIVERY POINT: 269-C | MAIL STOP: 269-2 | PHONE: 4-4277 | GLA ACCOUNT NUMBER / ITEM

I CERTIFY THAT THE FUNDS ARE AVAILABLE IN THE ABOVE ACCT. NO.

NAME | DATE

UNITED STATES OF AMERICA

BY:

INITIATOR MUST COMPLETE REVERSE SIDE OF THIS PAGE AND PAGE 8.

1 Form | 1 of 1

Figure 1. An Example Purchase Request

Figure 1 shows an example PR. Because of its size and complexity, we have focused on the Ames Purchase Request form for the development of the PEPR system. However, our implementation is largely form-independent, and can be applied to other forms that require “complex routing”. We have also applied the PEPR system to approximately six other types of forms and are actively pursuing other potential applications of the system both inside and outside of NASA Ames.

WHY AI?

The knowledge-based component of the PEPR system utilizes a fairly straightforward rule- and object-based mechanism to provide its validation and routing capabilities (although we are investigating machine learning techniques to ease the knowledge-acquisition problem -- see the section entitled Future Plans, below). There are two main reasons that a knowledge-based approach is appropriate to the problem of validation and routing of electronic forms.

First, the knowledge required to ensure the PR's correctness and completeness is quite diverse and very widely distributed among the various groups at Ames. Different validation "rules" come into play depending on what items or services are being ordered and what offices are required to approve a particular purchase. Early on in the project it became clear that these validation rules would have to be acquired and revised incrementally. In addition, the fact that different validation rules come into play at different stages of the approval cycle meant that the validation mechanism had to be both "item-sensitive" and "context-sensitive". By adopting a rule-based approach, we were able to design and implement a general mechanism for applying validation rules of varying complexity and then add and/or refine form-specific validation rules as they were discovered.

Second, the knowledge that we required to generate a correct "approval path" for a particular PR was not well-defined and distributed among a wide variety of people. We also recognized that in order to guarantee that the system would always generate a correct set of approvers, we would need to be able to incrementally add routing knowledge as "holes" in the existing routing knowledge became apparent. The inherent separation of "inference engine" and "knowledge base" in rule-based systems offered a clear advantage over a conventional procedural approach.

WHY CLIPS?

We decided to use the CLIPS shell to implement the knowledge-based portion of the PEPR system for a variety of reasons. First, the data-driven nature of forms processing seemed to suggest that a forward-chaining inference engine would be appropriate. Second, CLIPS runs in a Macintosh environment, which is the platform of choice among our targeted pilot users. Third, the availability of CLIPS source code meant that we could tailor it to our specific needs (see [2] for a more detailed description of the modifications we made to CLIPS). Several other projects within our branch had successfully applied CLIPS to a variety of problems, so there was a fair amount of local expertise in its use. Also, the fact that it is available to government projects at no cost made it particularly appealing.

KEY DESIGN REQUIREMENTS

To evaluate the suitability of a knowledge-based system in an automated workflow environment, it was of course necessary to provide other components of the workflow system. As a result, certain design issues and requirements were identified early in the project. The following represent key assumptions and design desiderata that probably apply to any automated workflow system:

- **Familiar user interface:** The electronic version of the form had to look very much like the paper form with which the users were already familiar. Also, the electronic form needed to perform the rudimentary operations that users have come to expect from any automated application (printing support, automatic calculation of numeric fields, etc.)

- **Reliable data transport mechanism:** In order to get the forms from user to user, the system had to utilize an easy-to-use and reliable electronic mail system.

- **User authentication:** Once users are expected to receive and process sensitive data electronically, they must be assured that the people who sent the data are who they claim to be. Therefore, our system needed to ensure authentication of its users.

- **Data integrity assurance:** Likewise, the users needed to be sure that the data they received had not been altered, either accidentally or intentionally, while in transit.

- **Seamless integration:** We wanted the operation of the knowledge-based component of the system to be completely invisible from the user and have its output appear as data on the form.

- **Tracking capability:** Enabling a user to determine where in the process a particular form is at any particular moment, without having to bother other users, is very important to the acceptance of an automated workflow system. We wanted our users to be able to determine the status of their submitted forms automatically.

Of course, we did not want to have to develop all of the mechanisms required to meet these key needs. Thankfully, most of the requirements described above had been provided by recently-released commercial products. Therefore, our goal in the development of the PEPR system was to make use of commercially-available technology to fulfill these requirements whenever possible, and to integrate the various software components as cleanly as possible. While this approach required us to make use of pre-release versions of some software components of the system (with many of the frustrations inherent in "beta testing"), it enabled us to focus on the development and integration of the knowledge-based component and also led to mutually beneficial relationships with the vendors whose products we utilized.

SYSTEM COMPONENTS

Because of the workflow-enabling capabilities of the latest release of the operating system, the availability of workflow-related products, and the relative popularity of the platform at Ames, we chose to implement the first version of the PEPR system on the Apple Macintosh. The PEPR system is comprised of several commercial software tools:

- **Expert System Shell:** As described above, we selected CLIPS with which to implement the knowledge-based portion of the PEPR system.

- **Electronic Forms Package:** The Informed™ package from Shana Corporation is used to produce high-fidelity electronic forms. This package is comprised of the Informed Designer™ program, which permits a forms designer to define the layout and functionality of the form, and the Informed Manager™ program which permits filling out the form by end-users.

- **Scripting Language:** The various software modules that comprise the PEPR system share data by means of AppleScript™, a scripting language for the Macintosh that allows the programs to interact with each other and share data, even across an AppleTalk network.

- **DBMS:** The PEPR system currently utilizes 4th Dimension™, a "scriptable" data base management system from ACIUS, to hold data associated with the routing and tracking of forms as they are sent from user to user.

These applications all operate together under version 7.1.1 of the Macintosh operating system (also known as "System 7 Pro") which provides system-level capability for electronic mail,

digital signatures (for user authentication and data integrity assurance) as components of Apple's PowerTalk™ software product. The PowerShare™ system, which provides the store-and-forward mail service and user catalog support, operates on a centrally-located server system and supports all client users.

Each user of the system is required only to be running System 7 Pro and the Informed Manager application. CLIPS and 4th Dimension reside on a central "server" and can be accessed remotely by all users.

KNOWLEDGE BASE STRUCTURE

The PEPR knowledge base is comprised of four main modules; the Validator, the Classifier, the Approval-path Generator, and the Organization "data base". In addition, each form that the PEPR system supports has its own set of form-specific validation rules that are loaded dynamically as the form is processed.

- **Validator:** The PEPR validator is responsible for ensuring that the various fields on the form are correct and complete. The validation rules are represented as CLIPS classes, and are organized hierarchically with their own "apply" methods. Actual form-specific validation rules are represented as instances of these classes and are loaded dynamically from disk files when a particular form type is to be validated. If a validation rule is violated, the validator creates an instance of an error object with a suitable error message that eventually gets returned to a field on the form.

- **Classifier:** If the validator finds no errors on the form, the Classifier is invoked. The Classifier uses the contents of specific fields on the form to construct hypotheses about potential categories to which the specific form might belong. The Classifier loads a group of form-specific "clues" that are comprised of a text string, a field name, a classification category, and a certainty factor. These clues are evaluated in turn; if the clue's text string is found within the associated field, then membership in the given category is established with the given certainty factor. These certainty factors can be positive or negative, and are combined using the CF calculus defined by Shortliffe *et al* for the MYCIN system. If the resulting certainty associated with a certain hypothesis exceeds a threshold value, then the form is said to belong to that category.

- **Approval-path Generator:** Once all of the applicable categories for a given PR have been determined, the approval-path generator looks at specific fields on the form and determines the originating organization. It then loads the form-specific routing rules, and determines both the "management" approvals that are required (which depend upon the originating organization and, often, the total dollar amount associated with the PR) and the "special" approvals that are required (which are dependent on the classification categories to which the PR was assigned). These approvals are represented as the various organizations that must approve the form. The approval-path generator then looks up the "primary contact" for each of these organizations in the "organization data base" and inserts that person's name in the forms electronic "routing slip". (Note that by updating the "primary contact" for an organization periodically allows forms to be routed to designated alternates should the real primary person be on vacation or otherwise unavailable).

- **Organization Data Base:** This portion of the knowledge base contains CLIPS objects that correspond to the various managerial groups and hierarchies at Ames, and is used to help generate approval paths, as described above. (Of course, this module is currently a very good candidate for re-implementation in some other format as an external data base,

and the PEPR team is currently negotiating with other Ames groups who maintain similar data bases for other purposes).

DEVELOPMENT HISTORY

The PEPR system has been under development on a part-time basis for the past three years. Since then, the system has undergone various changes, both in its architecture and functionality. In Early 1991, work began in investigating the problems associated with the procurement process and the potential applicability of software tools to help address those problems. The team identified a useful sub-class of purchase requests on which to begin work, namely those PRs associated with the funding of research at external universities (this sub-class had the advantages of being reasonably straightforward with respect to the routing required and of providing an immediate near-term benefit -- the AI Research Branch submits a substantial number of these PRs and would therefore be in a good position to evaluate the utility of such a system). In June of 1991, the forms required to support university grants were distributed to a small number of users. These forms included only the evaluation forms (not the PR), and did not utilize the knowledge-based component. Early in 1992, the electronic forms were re-implemented in Informed, which proved to be a superior forms package to the that which had been used previously. These forms (except the PR) were given to numerous users around the Center, and were well-received. The knowledge-based validator, although working, was not deployed to end-users because we lacked a mechanism to efficiently share data between the form and the knowledge system. By the fall of 1992, we had initial versions of both the validator and the approval-path generator working, but they were only usable as a demonstration because they were not well-enough integrated with the forms system to be given to end-users. This “integration” was by means of a popular keyboard-macro package that allowed the two applications to clumsily share data via a disk file. However, this approach had two serious drawbacks. First, the keyboard macro package merely simulated the manipulation of the user interface, and so the user would have been subjected to a very distracting flurry of dialog boxes and simulated mouse-clicks. Second (and more importantly), that integration required that both the forms package and the knowledge base be running on the user’s machine. That was an unacceptable limitation and would have undoubtedly “turned off” more users that it would have helped. We were, however, able to give the end-users electronic versions of the grant evaluation forms, which were somewhat helpful to the more experienced users even without the knowledge-based components. In early 1993, the team signed on as pre-release users of AppleScript, and modified the CLIPS shell to be “scriptable”. This not only enabled a more “seamless” and less distracting integration between the forms package and the knowledge base, but more importantly it enabled us to set up a single copy of the knowledge system on a central server and permit users to access it over the network. By the summer of 1993, we became pre-release users of the new operating system software (part of the Apple Open Collaboration Environment) that provided support for digital signatures, system-level electronic mail, and other workflow-facilitating features. With these new features came the ability not only to give real users access to the knowledge base validation and routing capability, but also the data integrity assurance that would be required to support electronic submission of the sensitive data contained on financial instruments such as a purchase request form.

In December 1993, the PEPR system “went live”, and is now in daily use within the Aerospace Systems Directorate at Ames for the electronic submission, approval, and routing of purchase requests and university grant evaluation forms. The system is even being used to electronically send grant award forms to selected universities, something that had previously been done manually by the University Affairs Office at Ames.

FUTURE PLANS

The PEPR team is currently supporting the University Grant pilot testers, and is in the process of making small refinements to the system as the users report problems and suggest improvements. In the coming months, we expect to be able to expand both the user base of the system and the scope of the purchase requests to which it is applied. We are also investigating other related workflow applications, both within Ames and at other government laboratories and within industry.

The PEPR team is also working very closely with researchers at Washington State University who are applying machine learning techniques to electronic forms. Our hope is that as our data base of "correct and complete" forms grows, we will be able to utilize these techniques to automatically generate new validation and routing rules.

REFERENCES

1. Compton, M., Wolfe, S. 1993 Intelligent Validation and Routing of Electronic Forms in a Distributed Workflow Environment.. Proceedings of the Tenth IEEE Conference on AI and Applications.

2. Compton, M., Wolfe, S. 1994 CLIPS, Apple Events, and AppleScript: Integrating CLIPS with Commercial Software. Proceedings of the Third Conference on CLIPS.

3. Compton, M., Stewart, H., Tabibzadeh, S., Hastings, B. 1992 Intelligent purchase request system, NASA Ames Research Center Tech. Rep. FIA-92-07

4. Shortliffe, E. H. 1976 Computer-based medical consultations: MYCIN. New York: American Elsevier.

5. Hermens, L.A., Schlimmer, J.C. 1993 Applying machine learning to electronic form filling. Proceedings of the SPIE Applications of AI: Machine Vision and Robotics

A KNOWLEDGE-BASED SYSTEM FOR CONTROLLING AUTOMOBILE TRAFFIC

Alexander Maravas* and Robert F. Stengel**

Department of Mechanical and Aerospace Engineering
Princeton University
Princeton, NJ 08544

ABSTRACT

Transportation network capacity variations arising from accidents, roadway maintenance activity, and special events, as well as fluctuations in commuters' travel demands complicate traffic management. Artificial intelligence concepts and expert systems can be useful in framing policies for incident detection, congestion anticipation, and optimal traffic management. This paper examines the applicability of intelligent route guidance and control as decision aids for traffic management. Basic requirements for managing traffic are reviewed, concepts for studying traffic flow are introduced, and mathematical models for modeling traffic flow are examined. Measures for quantifying transportation network performance levels are chosen, and surveillance and control strategies are evaluated. It can be concluded that automated decision support holds great promise for aiding the efficient flow of automobile traffic over limited-access roadways, bridges, and tunnels.

INTRODUCTION

U.S. automobile traffic has been growing by 4 percent a year to its current level of 2 trillion vehicle-miles, and it is expected to double to 4 trillion vehicle-miles by 2020. According to Federal Highway Administration, if no significant improvements are made in the highway system, congestion delays will increase by as much as 400 percent [1]. According to IVHS America, the annual cost of congestion to the U.S. in lost productivity is estimated at over $100 billion [2]. In many areas there is very little that can be done to increase road capacity. There is not adequate right-of-way next to existing roads, and in many cases the cost of a new highway is prohibitively expensive. It is therefore imperative that new ways be sought to make better use of existing infrastructure.

In 1987 the Federal Highway Administration formed Mobility 2000, a joint effort between the government, industry and academia. This led to the formation of an organization called the Intelligent Vehicle Highway Society of America, or IVHS America [1]. IVHS America aims at improving the level of transportation services that are currently available to the public by integrated systems of surveillance, communications, computer and control process technologies [4].

IVHS technologies have been grouped into four generic elements: Advanced Transportation Management Systems (ATMS), Advanced Driver Information Systems (ADIS), Automated Vehicle Control (AVC), and Commercial Operations [3]. This paper concentrates on ATMS, which involves the management of a transportation network. Implementation of such systems requires development of real-time traffic monitoring and data collection techniques. More precisely, an Advanced Traffic Management System should have the following characteristics, as specified by the proceedings of the Mobility 2000 conference [3,4]:

- real time operation

---

* Student
** Professor

- responsiveness to changes in traffic flow
- surveillance and detection
- integrated systems
- collaboration of jurisdictions involved
- effective incident control strategies

Effective incident control strategies will be a crucial part of this project. Contrary to widespread belief, not all congestion is due to rush hour traffic; 56% of costs incurred by congestion are due to non-recurrent events or incidents. Incidents include vehicle accidents, unfavorable weather conditions, highway maintenance, and road reconstruction [3]. It is essential to determine the nature and scope of an incident as quickly as possible. The control center should be informed about the incident either through the police or other jurisdictional agencies. A more advanced approach would be visual validation of incidents with the use of camera surveillance systems. Effective detection and verification of incidents will lead to lower disruption of traffic flow [3]. Drivers could be routed to alternate paths to avoid unnecessary tie-ups and frustration due to long delays [3].

Traffic control centers would need real-time information about the network condition. An intelligent vehicle/highway system would have to monitor traffic throughout the day. For the near future, sensors will include inductive loops buried just below the surface and ultrasonic sensors mounted overhead. These devices will be able to count the number of vehicles passing a certain point and will gauge their speed. Another alternative would be for image-processing computers to extract traffic data from television pictures from cameras on the arterial network. Recent tests have provided very promising results about the accuracy of inductive loop detectors [1]. Ultimately, improvement of network surveillance technologies and link-time estimation techniques will be crucial in the implementation of an intelligent vehicle/highway system [5]. In the future, vehicles equipped with navigational systems could communicate directly with the control center, giving information about the drivers' locations, speeds, and destinations.

Advanced Traffic Management Systems will have to be integrated with Advanced Traveler Information Systems (ATIS) to ensure higher efficiency of the control system. Drivers will be informed about congestion, roadway conditions, and alternate routes through audio-visual means in the vehicle and through variable message signs at strategic points of the network. Information provided might include incident locations, fog or snow on the road, restrictive speeds, and lane conditions. Two-way real-time communication between vehicles and the control center could be facilitated by radio communications, cellular systems, and satellite communications [4].

Among the benefits of IVHS will be reduction in traffic congestion, reduction in the number of accidents , improved transit service, less fuel wasted, and fewer emissions from idling engines [4]. Fully integrated ATMS/ATIS combinations could reduce congestion in urban areas from 25 to 40%. Unchecked traffic congestion is the largest contributor to poor air quality and wasted fuel consumption. IVHS will not solve all problems in transportation, but it will increase the level of services rendered.

FUNDAMENTALS OF TRAFFIC FLOW MODELING

Evaluating traffic performance requires a thorough understanding of traffic flow characteristics and analytical techniques. The most important macroscopic flow characteristics are flow rate, density, and speed [7]. The flow rate q past a point is expressed as [8]:

$$q = \frac{n}{T} \qquad (1)$$

where

q: flow rate past a point
n: number of vehicles passing point in time interval T
T: time interval of observation

It is important to recognize that q is sensitive to the selected time interval T during which the measurement began and ended. Whatever the value of T, the most common units for q are vehicles/hr.

Density (or concentration) can be expressed as [7,8]:

$$k = \frac{n}{L} \qquad (2)$$

where

k: density
n: number of vehicles on road
L: length of road

Density is an instantaneous traffic measurement, and its units are usually vehicles/lane-mile. In most cases, one mile and a single line of vehicles are considered. Traffic densities vary from zero to values that represent vehicles that are completely stopped. The upper limit of k is called the jam density and is on the order of 185 to 250 vehicles per lane-mile, depending on the length of the vehicles and the average distance between vehicles.

Speed is another primary flow variable. Space-mean speed is the average speed of the vehicles obtained by dividing the total distance traveled by total time required, and it is expressed as [8]:

$$u = \left(\sum_{i=0}^{n} s_i \right) / \left(\sum_{i=0}^{n} m_i \right) \qquad (3)$$

where

u: space-mean speed
s_i: distance traveled by vehicle i on roadway
m_i: time spent by vehicle i on roadway

A very important relationship exists between the three fundamental stream flow variables that have been defined above. This flow relationship is [7]:

$$q = ku \qquad (4)$$

A linear speed-density relation has been assumed to simplify the presentation. The relation can be expressed as [7]:

$$u = u_f - \left(\frac{u_f}{k_j} \right) k \qquad (5)$$

This relationship indicates that as speed approaches free flow, speed density and flow approach zero. An increase in density causes a decrease in speed until flow is maximized at q_m and speed and density reach their optimum values (u_o, k_o). Further increases in density cause density to reach its maximum value (k_j) and speed and flow to approach zero [7]. A relationship between flow and density can be obtained by substituting equation (5) into (4). This yields [7]:

$$q = u_f k - \left(\frac{u_f}{k_j}\right) k^2 \qquad (6)$$

Under low-density conditions, flow approaches zero and speed approaches free-flow speed (u_f). At optimum density, flow is maximized, and speed attains an optimum value. Maximizing the objective function (6) by setting its derivative equal to zero ($dq / dk = 0$), we find that optimum density occurs at half the jam density ($k_o = k_j / 2$). This is true only for a linear speed density relationship, but the same reasoning can be applied to other non-linear relationships [7].

The optimum speed is half the free flow speed ($u_o = u_f / 2$). Because $q_m = k_o u_o$, it is evident that $q_m = u_f k_j / 4$. Once again, it is important to remember that these values are only true for a linear speed-density relation [7]. The flow-density relationship often serves as a basis for highway control. Density is used as the control parameter, and flow is the objective function. At low densities, demand is being satisfied and level of service is satisfactory, but as density increases control is needed to keep densities below or near the optimum density value [7]. Other models have been proposed by transportation planners. This is an optimization problem with traffic flow as the objective function and traffic density as the control parameter. The technology to measure traffic density exists, and its knowledge can help us estimate traffic flow, average speed, travel time, and level of service.

ASSUMPTIONS OF SURVEILLANCE AND CONTROL PROCEDURES

Every morning there is a large pool of people west of the Hudson river who want to cross it. Every member of this group of "intelligent agents" is part of a continuously changing environment, holding an individual behavioral response to the evolution of the dynamic system.

It has been observed that commuters usually choose the route with the shortest travel time. Consequently, drivers tend to divide themselves between two routes in such a way that the travel times between them are identical. Finally, an equilibrium point is reached in which commuters do not have much choice between two routes since the user cost (travel time) of traversing the two links has stabilized [8].

In many cases, an incident can disturb the user equilibrium. In such instances transportation planners can help establish a new equilibrium where no route is underutilized. By dispensing traffic advisories to the right people, one can be assured that misallocation of transportation resources can be avoided. Computer simulations have proven that the use of route guidance systems can reduce travel time to all drivers by up to 20% [9]. Route guidance was found to be more helpful as the duration of incidents increased, in which cases more drivers had to be routed to achieve an optimal traffic assignment [9]. A corresponding surveillance procedure is illustrated in Figure 1.

In selecting between alternate routes, a user-optimal system chooses a route that minimizes the travel time of the individual driver. However, a system-optimal system selects a set of routes that

minimizes the overall travel time of all drivers [11]. Routing decisions should not be made independently because every decision effects the whole network: if many drivers choose a certain route at the same time, that route may become congested and non-optimal [10].

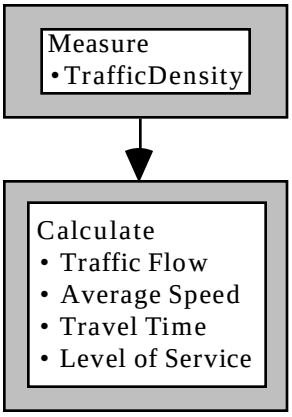


Figure 1. Proposed Surveillance Procedure.

A DECLARATIVE FRAMEWORK FOR TRAFFIC CONTROL

Traffic control can be partitioned into reflexive, procedural, and declarative functions. Reflexive functions operate at the subconscious level of human thought; they are instantaneous reactions to external stimuli. Procedural actions also operate on a subconscious level, but they are more complex than reflexive functions, following a set of basic rules and actions that represent skilled behavior. Declarative functions operate at the conscious or preconscious level of thought. On the conscious level they require attention; on the preconscious level they require intuition and conceptual formulations. They involve decision making and provide models for system monitoring, goal planning and system/scenario identification [12,13].

A traffic management system requires goal planning and system monitoring. At every instant, all alternatives must be considered and decisions must be made through a deductive process [13]. A declarative model makes such decisions in a process that is similar to human reasoning [6]. All our knowledge, beliefs, and experience of traffic modeling are placed in a declarative framework to provide a system that reasons in an intelligent manner.

This paper focuses on declarative traffic controls. Rules that reflect the controllers knowledge of the response of the control system are established. Programming is implemented as a CLIPS expert system that supports various programming paradigms. CLIPS was chosen because it can be easily integrated with C programs. The rule-based programming paradigm is useful in modeling traffic incidents. In procedural programming, the order in which all the commands are executed is pre-specified. In reality, traffic incidents are often unpredictable. By establishing the appropriate heuristics and rules, the system becomes "intelligent." When it is faced with a certain problem, it uses pattern matching that is appropriate to the existing facts.

Computer systems often are organized in a modular structure to increase computational efficiency. Time can be saved by looking at rules and facts that are relevant at that instant. Modular representation allows partitioning of the knowledge base into easily manageable segments, thus making the system easily expandable. CLIPS incorporation of modules is similar to the blackboard architecture of other systems. Knowledge sources are kept separate and independent, and different knowledge representation techniques can be used. Communication of all sources takes place through the blackboard [15].

Task definition is an important factor in the development and design of such rule-based systems. The ultimate goal is to develop an expert system of expert systems, which is a hierarchical structure that reasons and communicates like a team of cooperating people might [13].

A knowledge-based system called the Traffic Information Collator (TIC) has been developed at the University of Sussex. The TIC receives police reports on traffic incidents and automatically generates appropriate warning messages for motorists. The system operates in real-time and is entirely automatic. The TIC is comprised of five processing modules each holding its own knowledge base [16,17].

Research in air traffic control has been conducted in a similar manner. Cengeloglu integrated an Air Traffic Control Simulator (written in C) with a traffic control decision framework (implemented in CLIPS). The simulator creates a virtual reality of airspace and continuously updates the information (knowledge base) of the decision framework. Several scenarios containing unexpected conditions and events are created to test the prototype system under hypothetical operating conditions [15].

A complete transportation model should consist of a traffic simulator and a traffic manager. This paper focuses on the decision process of managing and controlling traffic.

Prediction algorithms could be employed to predict the evolution of traffic. This additional knowledge could be used in the control process. Smulders created a model to study the use of filtering on freeway traffic control [18]. Once the actual values of the density and mean speed in all sections of the freeway are available, his model generates predictions for the evolution of traffic over short time periods (e.g., 10 minutes). A state vector contains all the information about the present state of the system. The estimation of a certain state from the measurement vector is done through the use of a Kalman filter [18].

The real-time operation of knowledge-based systems in actual or simulated environments is attained through the use of a cyclic goal-directed process search, with time-sliced procedure steps. Prior to a search, the value of parameters is either *known* or *unknown*. After the search, the status of parameter values may be changed. Repetitive knowledge-base initialization sets the value of every parameter to its default value after each search cycle; all information that was attained in the previous search is deleted. Thus the controller “forgets” the information it acquired in a former search prior to solving a new problem.

The CLIPS knowledge base can be initialized by resetting the program. All the constructs remain unchanged and do not have to be reloaded. This process allows the accommodation of time-varying data in the system [14]. In this sense, real-time application implies some parallel execution features. Several domains of the research space must be searched concurrently [19]. It is noteworthy that not all cyclic search processes of the system have to be synchronous.

USING A KNOWLEDGE-BASED SYSTEM FOR DECISION AIDING IN TRAFFIC CONTROL

The aim of this project is to design an expert system that evaluates traffic conditions and dispenses travel advisories to commuters and traffic control centers. A decision-support system has been written in the CLIPS programming environment to illustrate several traffic control procedures (Fig. 2). The program is geared toward illustrating a method of traffic control rather than solving a particular problem. Some traffic parameters have been approximated and certain simplifying assumptions have been made to avoid unnecessary computational complexity.

A small network was constructed for initial program development. The network consists of a section of the New Jersey Turnpike and the routes connecting Exits 14, 16, and 18 to the Holland

Tunnel, Lincoln Tunnel, and George Washington Bridge. Node and link data for all the roads in New Jersey and New York has been accumulated by the Civil Engineering Department at Princeton University. Future research could be geared toward applying the declarative control procedures developed in this project to a network of a larger scale.

Actual implementation of this system is based on it receiving real-time information about traffic conditions of a network using sensor measurements at different points on the network. The current program uses scenario files, with hypothetical traffic densities from all roads in the network. The program reads these values (as well as the pertinent time period of the day) and creates the appropriate data constructs, which are asserted as facts. The system then matches these facts based on pre-specified heuristics. Once it has concluded a declarative search, it gives certain advisories and recommendations at the CLIPS command prompt. All advisories are made with the idea that they would be broadcast on changeable signs at several roadway locations; they could also be displayed at traffic management centers or on the displays of suitably equipped vehicles. Alternative scenario files test the system under several hypothetical operating conditions.

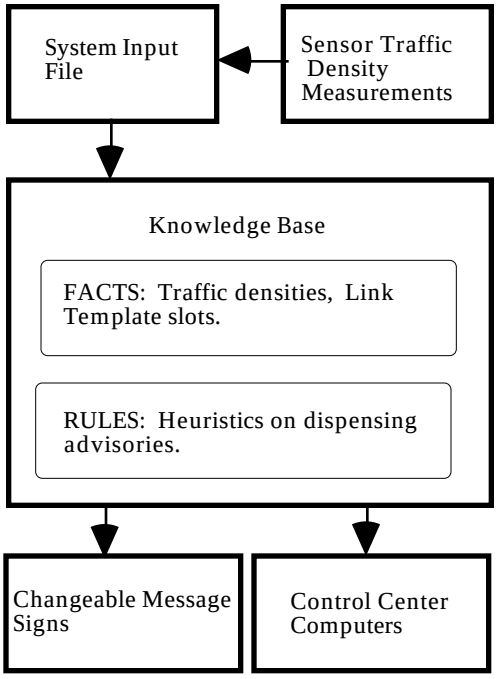


Figure 2. Schematic Representation of System Architecture.

IMPLEMENTATION OF THE TRAFFIC MANAGEMENT SYSTEM

The decision-support system, programmed with CLIPS 6.01, supports several procedural functions, that are necessary for the computation of relevant information. Once the density is known, the average travel speed, flow rate, service rate, and level of service can be calculated from the appropriate models and equations.

Historical data of expected traffic demand for the Hudson river crossings have been stored in a function that returns the expected number of vehicles at a certain time period. The use of an appropriate filter would make this function redundant. It provides information about the probable evolution of traffic and is sufficient for the preliminary development of the system. It makes the system "forward looking" and capable of predicting congestion buildup. Historical data could be

stored as facts; however, this clogs up the facts list, and the system is forced to look at irrelevant facts.

After reading initial data from the scenario file, the roadway densities are asserted as facts, and they are put onto the facts list. The initialization rules create the appropriate data constructs based on these basic facts and the appropriate procedural functions. Most of the program's knowledge is stored in templates, which are similar to structures in C. Thus every link of the network has it own template, containing information such as speed, density, flow, operational lanes, service rate, and accident-status. Templates are convenient for storing data and can be readily accessed and modified by the user. After all the values of the slots of the templates have been calculated, the templates are asserted as facts. Thereafter, decisions are made by pattern matching on the values of the templates' slots. Data storage is compact so as not to overflow the CLIPS fact list.

The system is readily adjustable to lane closures due to maintenance. For instance, if one of the lanes of the Lincoln Tunnel is closed due to maintenance, then when the knowledge base is initialized, it takes this fact into account. In the case of bad weather, such as a snowstorm, the values of the free-flow speed and the jam density should be reevaluated. Since these deviations from normal operating conditions are incorporated into the system once it is initialized, all decisions made thereafter are adjusted accordingly. System initialization at every time increment ensures that the link-node data is augmented to reflect current conditions. Thus the system is very flexible and can include all foreseeable traffic situations.

Broadcasting travel advisories is a challenging part of the program. For instance, the fact that there is an accident on the route from Exit 14 to the Holland Tunnel is useful for people approaching that exit, but not for people traveling away from it. Thus the system is faced with the decision of whether to make a broadcast and to whom it should be made. Due to the geometry of the network, the broadcast heuristics are different for every decision node. Even in such a small network, there are numerous broadcasting possibilities.

Automated communication between commuters and the control center can be achieved by the appropriate use of advisory rules. The data templates also contain information such as the presence and severity of accidents on the links of the network. Every link’s template has a slot that is called "accident status," which can be either TRUE or FALSE. The default value is FALSE, but once an accident occurs it is switched to TRUE. Thereafter, the system ensures that the appropriate people are informed of the incident. Templates also store the estimated time to restore the link to normal operating conditions and the degree of lane blockage at the accident site.

While radio advisories provide commuters with information about traffic delays and adverse traffic conditions, it is doubtful that they give them all the information they need at the right time. Since not all drivers are tuned to the same station and radio advisories may lack central coordination, the result of broadcasting may not be system-optimal. Changeable message signs ensure that the appropriate people get the right message at the right time. The system is more helpful under such circumstances, since it can help bring the network back to user equilibrium by ensuring that all links are properly utilized.

In general the factors that should be considered in any routing decision are: a) traffic density and velocity profile of main and alternate route, b) length of main and alternate route, c) percentage of divertible traffic volume, and d) demands at on-ramps on both routes [20]. Once the density of the link is known, an average speed can be computed using one of the traffic models described earlier in this paper. Dividing the length of the link by the average speed yields the current (experienced) travel time.

Routing between two alternate routes can be achieved by the use of a tolerance-level measurement. The difference between experienced travel times (or flow) between two routes can be calculated. If it is higher than some pre-specified level, then commuters should be routed to the underutilized roadway. The use of individual travel times is user-optimal, whereas the use of traffic flow is system-optimal. Optimizing commuters travel time can probably be done better with the use of shortest-path algorithms, whereas equilibrating flows can be incorporated in an expert system.

Traffic flow is a measure of highway productivity. Ensuring maximum utilization is essentially an optimization problem, with flow as the objective function and density acting as the control parameter. The flow of a link is maximized at the optimal density (k_o). For a linear model, the optimal operating density is half the jam density (k_j). The system has the expertise and knowledge to recognize how far the actual density measurements are from ideal conditions. If the traffic density of a road is less than optimal, then the road is under-utilized. If the density is higher then optimal, then it is over-utilized. An advanced intelligent highway system should be able to monitor and compare traffic flow in both directions of a link to see if the decision for a lane-direction change is warranted. The ultimate goal is to ensure that all routes are utilized properly. This section of the system is subject to a lot of development and improvement since the domain knowledge is uncertain. Currently there is no clear way to route traffic optimally.

The system searches its historical data to see if traffic demand for a link is expected to rise or fall in the next time period. If travel demand is expected to fall and the road is being underutilized, the system suggests that more vehicles be routed to that link. Diversion of traffic flow is an effective method of improving traffic performance and can be achieved by rerouting drivers with specific direction and destinations [20]. Advisories on the New Jersey Turnpike are different for people traveling North than for those traveling South.

The issue of how long a message should be broadcast is significant. Suppose that congestion can be avoided if 30% of drivers respond to the diversion sign. If only 15% of the drivers follow the diversion recommendation, congestion will not improve as expected. A feedback control system could take this into account by deciding to broadcast the diversion message for a longer time period until the desired utilization levels are reached. This approach compensates for all uncertainties in the percentage of divertible traffic flow [20]. The critical design issue is assuring that the system reaches stability quickly.

Knowledge base initialization at frequent time intervals, through the use of sensor measurements, makes this method of broadcasting a closed-loop feedback control process. Since the real-time implementation of this system would rely on cyclic search, a message would be sent every time the system decides that traffic should be diverted. Currently the program does not have real time measurements or simulated traffic demands, so it does not execute a cyclic search. It considers 24 one-hour periods over the span of a day. Real time implementation would require that the CLIPS knowledge be reinitialized at every time increment with the state measurements.

CONCLUSION

This project has made a step towards emulating an automated decision process for an Advanced Transportation Management System. This non-conventional approach to transportation modeling has examined the applicability of intelligent control techniques in management. Since a fully operational Intelligent Vehicle Highway System will require full integration of symbolic and numerical knowledge, declarative rules have been embedded with procedural code. A framework for modeling traffic incidents has been provided. The link information of the system is

augmented on a real-time basis, and advisories are issued based on the current state of the system.

Operation of this system requires that the traffic control center has knowledge of the traffic densities of all links in the transportation network. The technology for making such measurements exists and has been described earlier in this paper. Implementation of this system will require research into how programming software will be integrated with all the system sensors.

Before fully implementing such a control process, it is necessary to choose what degree of automation the system should have. The system should be allowed to run on its own, without human intervention, only when all possible errors have been removed from it.

It is difficult to foresee all the incidents that could happen. Even for such a small network there are many possibilities for issuing travel advisories. As the area that is monitored by sensors becomes larger, it is evident that human operators cannot check everything that is going on. However, the heuristic rules and frequent knowledge-base initialization make the system adaptive to many situations. The size of the knowledge base and the number of advisories are limited only by the available computer memory. Initially the system can be tested in a small area. Thereafter, additional rules for broadcasting to other locations can be added incrementally. Additional details of this research can be found in Ref. 21.

FUTURE WORK

Future work can be geared towards expanding the knowledge base by using the object-oriented programming paradigm offered by CLIPS. Node and link data of large networks could be stored compactly in an object-oriented format. Different classes of roads could be defined (e.g. arterial, expressway, intersection). Every link would then be an instance of these classes and it would inherit some of its properties from them.

The cyclic search must be implemented with the use of actual or simulated data, requiring the knowledge base to be reset (initialized) at frequent time intervals. It would be interesting to link CLIPS with a traffic simulator written in C. The simulator could generate traffic demands, and the CLIPS program could issue appropriate advisories.

This system knowledge base is subject to refinement. Additional rules must be added so the system knows what to do in the absence of certain density measurements, which could arise from malfunctioning sensors. Since not all transportation engineers use the same traffic models, it would be desirable to allow the user to use different traffic models or to be able to create his own model with an equation parser. The traffic routing technique and its relevant objective function needs to be reevaluated. Backlogs due to ramp-metering also should be examined. The use of estimators and prediction algorithms would enhance system performance significantly.

A learning control system would be able to learn from its daily experiences. Initially, it could be "trained" on a set of simulated data. Data archiving, on a daily basis, would increase the size of the system's knowledge base. Thereafter, it could evaluate how well it handled a previous accident. Hence, if it was in a similar situation it would use its acquired expertise to issue the appropriate advisories. An intelligent system could detect traffic incidents from unusual sensor readings. Eventually the system would be able to distinguish between weekday and weekend traffic patterns. Considering the recent technological advances in intelligent control systems, the era of the fully automated highway might not be very far away.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation, Grant No. ECS-9216450.

REFERENCES

1. Bernstein D., Ben Akiva M., Holtz A., Koutsopoulos H., and Sussman J., “The Case for Smart Highways," *Technology Review*, July 1992.

2. Anon., IVHS Strategic Plan Report to Congress, Department of Transportation, Dec. 18, 1992.

3. Proceedings of a Workshop on Intelligent Vehicle/ Highway Systems, *Mobility 2000*, San Antonio Texas, Feb. 15-17, 1989.

4. Euler G.W., “Intelligent Vehicle/Highway Systems: Definitions and Applications," *ITE Journal*, Nov. 1990.

5. Ervin, R.D., *An American Observation of IVHS in Japan*, Ann Arbor, Michigan, 1991.

6. Stratton D.A., *Aircraft Guidance for Wind Shear Avoidance: Decision Making Under Uncertainty*, Ph.D. Dissertation, Princeton University, Princeton, 1992.

7. May A.D., *Traffic Flow Fundamentals*, Prentice Hall, Englewood Cliffs, 1990.

8. Morlok E.K., *Introduction to Transportation Engineering and Planning*, McGraw-Hill, New York, 1978.

9. Rakha H., Van Aerde M., Case E.R., Ugge A., “Evaluating the Benefits and Interactions of Route Guidance and Traffic Control Strategies using Simulation”, IEEE CH2789, June 1989.

10. Solomon M., *New Approaches to the Visualization of Traffic Flows in Highway Networks*, Senior Thesis, Princeton University, Princeton, 1992.

11. Van Aerde M., Rakha H., “Development and Potential of System Optimized Route Guidance Strategies”, IEEE CH2789, June, 1989.

12. Chao T., *Design of an Intelligent Vehicle/Highway System Computer Simulation Model*, Princeton University, Mechanical and Aerospace Engineering Department Senior Independent Work Final Report, Princeton, 1994.

13. Stengel R.F., "Probability-Based Decision Making for Automated Highway Driving, to appear in *IEEE Trans. Vehicular Technology* (with A. Niehaus).

14. Handelman D.A., *A Rule-Based Paradigm for Intelligent Adaptive Flight Control*, Ph.D. Dissertation, Princeton University, Princeton, June 1989.

15. Cengeloglu Y., *A Framework for Dynamic Knowledge Exchange Among Intelligent Agents*, Masters Thesis in Electrical and Computer Engineering, University of Central Florida, Orlando Florida, 1993.

16. Evans R., Hartley A.F., “The Traffic Information Collator”, *Expert Systems*, Nov. 1990, Vol. 7, No. 4.

17. Allport D., “Understanding RTA's”, *Proceedings of the 1988 Alvey Technical Conference*, 1988.

18. Smulders S.A., “Modeling and Filtering of Freeway Traffic Flow”, *Transportation and Traffic Theory*, Elsevier, New York, 1987.

19. Le Fort N., Aboukhaled, D.Ramamonjisoa, “A Co-Pilot Architecture based on a multi-expert system and a real-time environment”, *1993 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 5, France Oct. 17-20, 1993.

20. Cremer M., Fleischmann S., “Traffic Responsive Control of Freeway Networks by a State Feedback Approach”, *Transportation and Traffic Theory*, Elsevier, New York, 1987.

21. Maravas, A., *A Knowledge-Based System for Optimal Control of Traffic Flow*, Princeton University, Mechanical and Aerospace Engineering Department Senior Independent Work Report, Princeton, 1994.

DEVELOPMENT OF AN EXPERT SYSTEM FOR POWER QUALITY ADVISEMENT USING CLIPS 6.0

A. Chandrasekaran and P.R.R. Sarma
Tennessee Technological University
Cookeville, TN 38501

Ashok Sundaram
Custom Power Distribution Program
Electric Power Research Institute
Palo Alto, CA 94303

ABSTRACT

Proliferation of power electronic devices has brought in its wake both deterioration in and demand for quality power supply from the utilities. The power quality problems become apparent when the users' equipment or systems maloperate or fail. Since power quality concerns arise from a wide variety of sources and the problem fixes are better achieved from the expertise of field engineers, development of an expert system for power quality advisement seems to be a very attractive and cost-effective solution for utility applications. An expert system thus developed gives an understanding of the adverse effects of power quality related problems on the system and could help in finding remedial solutions. The paper reports the design of a power quality advisement expert system being developed using CLIPS 6.0. A brief outline of the power quality concerns is first presented. A description of the knowledge base is next given and details of actual implementation include screen outputs from the program.

INTRODUCTION

The introduction of nonlinear loads and their increasing usage, have led to a point where the system voltage and current are no longer sinusoidal for safe operation of the equipment at both industrial and customer levels. Before the advent of electronic and power electronic devices, the current distortions were due to saturation of the magnetic cores in the transformers and motors, arc furnaces and mercury arc rectifiers. Even though the overall effect on the system was there, it had no serious effect on the performance of comparatively more rugged and insensitive equipment. Today, sensitive electronic and microprocessor based equipment are commonly used and they are susceptible to variations and distortions of the sinusoidal wave. Distorted sinusoidal waveforms of voltage and current are produced due to the nonlinear characteristics of the electronic components which are used in the manufacture of any electronic related equipment. The power quality problems arising in the system basically are impulses, surges, sags, swells, interruptions, harmonics, flicker etc. These power quality problems can be a potential threat to the satisfactory operation of the equipment. Thus there is a need to alleviate or eliminate the effects of poor power quality. Engineers are applying their experience with the causes of power quality impairment to recommend solutions to correct the problems.

Expert systems could be used for domain specific problems, such as power quality. The solutions to power quality may be many and it may not be possible to come to a single conclusion. Expert System approach can be useful to get a very successful approximate solution to problems which do not have an algorithmic solution and to ill-structured problems where reasoning may offer a better solution. The opinions of the experts may vary regarding a particular problem, but the level of expertise combined from several experts may exceed that of a single human expert, and this can be made use in an Expert System. The solutions given by an expert system are unbiased but the solutions from an expert may not always be the same. Databases can be accessed by anexpert system and algorithmic conclusions can be obtained wherever possible.

POWER QUALITY (PQ) IN POWER SYSTEMS

The impulses arising in the system are usually due to lightning and they cause insulator flashover, transformer failures, arrestor failures and damage to other substation equipment. Lightning strokes can pass through the transformer neutrals and cause damage to the customer equipment also. This is especially when the lightning strikes close to the distribution transformers. Oscillatory transients can be classified into three groups as low frequency, medium frequency and high frequency transients. Their range is from below 5 kHz to above 500 kHz. The main causes for these transients are capacitor switching, cable switching, back to back capacitor energization, travelling waves from lightning impulses and circuit switching transients. The impact of these transients are very high voltage levels on the secondary side of the transformers, especially due to capacitive coupling between primary and secondary windings, for medium frequency transients. Thus, these transients can also cause equipment failures and disruption of sensitive electronic equipment.

Sags and Swells can be classified into three groups namely instantaneous, momentary and temporary. They may last from 0.5 cycle to 1 minute. Sags may cause dropout of sensitive electronic equipment, dropout of relays, overheating of motors etc. Swells are usually due to single-line-to-ground faults occurring in the system. The extent of voltage rise varies with the type of grounding scheme adopted. An increase in the voltage of 73.2% is for ungrounded systems. The effect of swells could cause Metal Oxide Varistors to be forced into conduction.

Over-voltages and under-voltages last longer than 1 minute. They are caused by load switching, capacitor switching or due to bad system voltage regulation. The impact of theseis dropout of sensitive customer equipment which usually require constant voltage.

Harmonics are caused due to nonlinear characteristic of the loads of which converter circuits, arcing devices etc are some examples. Harmonics are multiples of fundamental frequency and are continuous in nature and distort the sinusoidal waveform. Interharmonics are caused by cycloconvertors and arc furnaces. The impact of harmonics can cause overheating of transformers and rotating machinery, maloperation of sensitive electronic equipment, maloperation of frequency sensitive equipment, metering errors, presence of neutral to ground voltage resulting in possible neutral overloading, capacitor failures or fuse blowing, telephone interference,increased power losses, etc.

Noise is caused by the range of components less than 200 kHz. Improper grounding and operation of power electronic equipment are the main causes of noise production. The impact of noise is on telephone interference. Notching is due to commutation in three phase inverters and converters. The number of notches depend on the converter and invertor configuration. The converters and invertor circuits can be operated in a six pulse, twelve pulse, eighteen pulse or more configurations. Flicker is usually caused by presence of components less than 25 Hz. Arc furnaces and intermittent loads like welding machines are some examples. These low frequency components may cause problems with lighting.

The various disturbances leading to power quality deterioration can stem from a wide variety of reasons that are often interdependant and quite complicated. A thorough analysis may be time consuming and inefficient. The solutions also depend upon experience of the system.

EXPERT SYSTEM APPROACH TO PQ

Expert systems came into existence as an outcome of the need for better problem solving methods where a closed form solution is unavailable. Since knowledge of power qualityproblems

is obtained more through study and experience, development of an Expert System is a viable approach.

An Expert system comprises of user interface, working memory, inference engine, agenda, and knowledge acquisition facility. The user interface is a mechanism by which the user and the expert system communicate. Working memory consists of a global database of facts used by the rules. An agenda is a list of rules with priority created by the inference engine, whose patterns are satisfied by facts or objects in the working memory. The knowledge acquisition facility is an automatic way for the user to enter knowledge in the system rather than by having the knowledge engineer explicitly code the knowledge.

Rule-based, object-oriented and procedural programming paradigms are supported by CLIPS 6.0. The basic components of CLIPS 6.0 are facts list, knowledge base and inference engine. The fact list contains the data on which inferences are derived, knowledge base contains all the rules and the inference engine controls the overall execution. Thus the knowledge base contains the knowledge and inference engine draws conclusions from the knowledge available. The user has to supply the expert system with facts and the expert system responds to the users' queries for expertise.

Knowledge can be represented by rules, semantic networks,parse trees, object-attribute-value triples, frames, logic etc. Each have their own limitation and a suitable field of usage [1]. Trees and lattices are useful for classifying objects because of hierarchical nature.

Decision trees can be used effectively in the development of power quality expert system due to hierarchical nature of power quality problems as usually one problem leads to several problems. A decision structure is both a knowledge representation scheme and a method of reasoning about it's knowledge. Larger sets of alternatives are examined first and then the decision process starts narrowing till the best solution is obtained. The decision trees shouldprovide the solution to a problem from a predetermined set of possible answers. The decision trees derive a solution by reducing the set of possible outcomes and thus getting closer to the best possible answer. Since the problems pertaining to power quality have different effects the solutions and remedial actions may be approximated by successive pruning of the search space of the decision tree.

A decision tree is composed of nodes and branches. The node at the top of the tree is called root node and there is no flow of information to the root node. The branches represent connection between the nodes. The other nodes, apart from the root node, represent locations in the tree and they can be answer nodes or decision nodes. An answer node may have flow of information to and from the other nodes and are referred as child nodes. The decision node is referred as leaf node and represents all possible solutions that can be derived from the tree. Thus the decision node terminates the program with solutions.

Broadly classifying a system, the power quality problems are felt at the distribution system level or a customer level. Thus this can be taken as the root node. At the distribution level the problem may be with the equipment at the substation or with the equipment in the distribution lines. Similarly the problem at the customer level could be with an industrial customer, a commercial customer or a residential customer. Thus these can be referred to as the child nodes. The problems faced at each of these levels may be understood better with each successive answer node, and thus probable answers could be arrived at, at the decision nodes. This approach could be useful if any numeric or monitored data pertaining to the system is not available. Also a probable answer can be concluded, depending on the answers given by the user only by observable impacts on the equipment, say the equipment maloperating or equipment failing or fuse failing, motor getting overheated etc. The efficiency or the level of certainty of solutions given will depend on the information provided by the user regarding the problem.

When monitoring equipment are connected at various points at the customer facility, data pertaining to the various disturbances due to surges, impulses, interruptions, variations in voltage, current etc can be collected or stored. Many such monitors are available and used by the utilities. PQ node program of Electric Power Research Institute (EPRI) uses a number of monitors for collecting disturbance data. The output from these analyzers can be used by the expert system for diagnosis. This approach could be very effective as the user may not be able to provide enough information from his knowledge. This will increase the efficiency of the expert system, as the user may give an incorrect answer or may not be able to answer some questions. The data files can be accessed into the expert systems, which is CLIPS 6.0, in our study and the data can be internally manipulated to come to a certain decision. If necessary, further information can be elicited from the user before arriving at a final solution.

Standard reports of the outputs with graphs of disturbances, possible solutions etc can be generated by the program by opening text files within the program.

IMPLEMENTATION OF THE EXPERT SYSTEM

EPRI is sponsoring a research project for the development of software modules for addressing power quality problems in the utilities. The center for Electric Power of the Tennessee Technological University is co-sponsoring the project. The major objectives of the project are the following :

- Design and test a prototype expert system for power quality advising.
- Develop dedicated, interactive analysis and design software for power electronic systems.
- Develop a concept of neural network processor for the power system disturbance data.

For developing the expert system, CLIPS software designed at NASA/Johnson SpaceCenter with the specific purposes of providing high portability, low cost and easy integration with other systems has been selected. The latest version CLIPS 6.0 for Windows provides greater flexibility and incorporates object-oriented techniques.

Figure 1 shows the configuration of the expert system blocks being analyzed for the power quality advisement. The rules are framed by the experts opinion and the number of rules can be increased and added to the program whenever possible. The monitored data from PQ nodes, user or both can supply the facts, which can be utilized to come to a probable solution. The inference engine and agenda fire the rules on the basis of priority. There is an explanation facility which gives reports and suggests the possible reasons for coming to a conclusion depending on the inputs given.

[Figure Deleted]

Figure 1.

The following is the pattern in which the questions are asked by the Expert System for the responses of the user. Here a batch file is run and CLIPS 6.0. response is given below.

```
CLIPS> (open "s.dat" s "w")
TRUE
CLIPS> (open "r.dat" r "w")
TRUE
CLIPS> (load "sept1.clp")
!!*****************************************************************
TRUE
```

```
CLIPS> (reset)
CLIPS> (run)

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
**
*Expert System  For Power Quality Advisement*
*Developed By*
*Dr. A. Chandrasekaran   and   P.R.R. Sarma*
*Center For Electric Power*
*Tennessee Technological University*
*Cookeville, Tennessee.*
**
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


   Where is the complaint from?

   1. Distribution-system
   2. Customer
1

   Where is the problem occurring ?

   1. Circuits
   2. Sub-Station
2

   What is the complaint ?

   1. Service-Interruption
   2. Device-Maloperation
   3. Equipment-Failure
2

   Which of these has the problem ?

   1. Transformer
   2. Circuit-Breaker
   3. Control-Circuitry
1

   Is the system voltage normal ?

   1. Yes
   2. No
1

   Is the hum of the transformer excessive ?

   1. Yes
   2. No
1

   Is the transformer getting overheated ?

   1. Yes
   2. No
1
```

```
   Is the transformer getting overheated even
   under normal load conditions ?

   1. Yes
   2. No
1

   Is the transformer supplying power to
   heavy lighting and power electronic loads ?
   1. Yes
   2. No
1

   Are there capacitor banks in the substation ?

   1. Yes
   2. No
1

   Is the substation provided with harmonic
   filters to filter harmonics ?

   1. Yes
   2. No
2


   Probable cause:

   Place filters to prevent saturation
   and overheating of the transformers
   when it supplies power to power
   electronic loads.

CLIPS> (exit)
```

The following is the symptoms file "s.dat", which was storing the symptoms pertaining to the problem is given below.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
*THIS EXPERT SYSTEM IS BEING*
**
*DEVELOPED BY*
**
*Dr. A.CHANDRASEKARAN & P.R.R. SARMA*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


SUMMARY OF SYMPTOMS OBSERVED:

** The complaint is from the distribution system.

** The trouble is in the substation.
** Complaint is device maloperation.

** The problem pertains to the transformer.
```

```
** The hum of the transformer is excessive.

** The transformer is getting overheated.

** The transformer is supplying power to lighting and
   power electronic loads.

** The transformer is getting heated under normal load conditions.
```

The following is the possible reasons file "r.dat", which was storing the possible reasons pertaining to the problem is given below.

```
** The transformer hum could be more due to loose core bolts, presence of
harmonics etc.

** Transformer may get overheated due to overloading, saturation,
insignificant faults in winding etc.

** Transformer can get overheated under normal load conditions due to
harmonics, insignificant faults in transformer etc.

** Harmonics may be in the system when the power is being supplied to
lighting and power electronic loads.

** Harmonics may be present if harmonics are not filtered out.
```

The text files generated can be imported into Word perfect, Winword etc for making the final reports. At present, efforts are being made to integrate the data obtained from the PQ analyzer into the CLIPS program as a part of the EPRI project. The monitored data reports from the PQ analyzer can be used to arrive at conclusions about the power quality problems especially with regard to the origin of the disturbances from the users' queries. Also the reports could incorporate disturbance graphs obtained from the data specified in the input files of the monitored data.

The concepts of the theories of uncertainty and Fuzzy Logic can be utilized using FuzzyCLIPS in the development of the power quality expert system, especially in cases where the questions like “Is the hum of transformer excessive?” have to be answered. The possibilities for ‘excessive’ may be more than normal, high, very high etc. FuzzyCLIPS may be used for answers of this sort and the efficiency of the solutions thus can be increased.

CONCLUSION

The development of the power quality expert system shows that expert system usage is definitely a viable alternative. Analysis of monitored data can be used to identify the sources causing power quality deterioration. Suitable conclusions can be drawn to recommend mitigating the power quality problem sources to the customers on one hand and the equipment manufactures on the other. An expert system approach can also be useful to educate customers on actions that can be taken to correct or prevent power quality problems.

REFERENCES

1. Giarratano Joseph and Riley Gary, “Expert Systems, Principles and Programming,” 2nd edition, PWS Publishing Company, Boston, 1993.

2. Adapa Rambabu, “Expert System Applications in Power System Planning and Operations,” IEEE Power Engineering Review, February 1994, pp. 12 to 14.

3. Liebowitz Jay, DE salvo A. Daniel, “Structuring Expert Systems,” Yourdon Press, Prentice Hall Building, Engelwood Cliffs, N.J. 07632, 1989.

4. “A Guide to Monitoring Power Distribution Quality,” phase 1, Electric Power Research Institute Project 3098-01, Report TR-103208, Palo Alto, California, April 1994.

QPA-CLIPS: A LANGUAGE AND REPRESENTATION FOR PROCESS CONTROL*

Thomas G. Freund

Pratt & Whitney
400 Main Street
Mail Stop 118-38
East Harford, CT 06108

ABSTRACT

QPA-CLIPS is an extension of CLIPS oriented towards process control applications. Its constructs define a dependency network of process actions driven by sensor information. The language consists of 3 basic constructs: TASK, SENSOR and FILTER.. TASKs define the dependency network describing alternative state transitions for a process. SENSORs and FILTERs define sensor information sources used to activate state transitions within the network. *deftemplate*'s define these constructs and their run-time environment is an interpreter knowledge base, performing pattern matching on sensor information and so activating TASKs in the dependency network. The pattern matching technique is based on the repeatable occurrence of a sensor data pattern. QPA-CLIPS has been successfully tested on a SPARCStation providing supervisory control to an Allen-Bradley PLC 5 controller driving molding equipment.

INTRODUCTION - THE NEED

Process control is the science and, at times, art of applying and holding the right setpoint value and/or mixing the right amount of an ingredient at the right time. But, when is the time right ? And, if we know enough about the material or mixture, what is the right setpoint value and/or amount that has to be mixed ?

Materials scientists and engineers have spent years of painstaking experimentation and analysis to characterize the behavior of metals, plastics, and composites. Along the same vein, manufacturing engineers and factory floor operators have developed many person-years worth of practical "know-how" about material behavior under a variety of processing conditions.

In addition, though there is always room for improvement, significant strides have been made in useful sensor technology for acquiring information on material behavior. As the types of materials used in everyday products increase in complexity, the demand increases for embedding a better understanding about material behavior directly into the control of their manufacturing processes.

QPA-CLIPS, the language and its run-time environment, is a means to directly tie our best understanding of material behavior to the control of material forming or curing processes. This is accomplished by intelligent mapping of sensor information on material behavior to changes in process parameter values. The changed parameter values, in turn, are used to directly drive process equipment .

* This work was supported by a grant from the National Center for Manufacturing Sciences, Ann Arbor, MI 48108 and in cooperation with Allen-Bradley Co. (Milwaukee, WI) and Erie Press Systems (Erie, PA).

THE PROBLEM

Material forming processes transform a prescribed volume of material(s) into a desired net shape. In curing, material is processed in a way that changes material properties to a set of desired characteristics. In either case, the proper choice of tooling along with a properly defined process cycle, or "recipe", are crucial to the consistent production of quality parts. Variations in material behavior from an expected norm must be accounted for in both the initial conditions of the material and, particularly, the cycle or "recipe" driving the process equipment.

Controlling a forming or curing process

Attaining a desired final form and material characteristics in forming or curing processes requires effective control of force and heat application. Control of force involves application of mechanical pressure to distort or redistribute a set volume of material. within a constraining volume, defined by the tooling (i.e., dies) used in the process.

In the case of forming processes, the material usually requires to be more pliable than its natural state at room temperature. Heat application then becomes an integral part of the pressure application. For curing processes, the application of heat itself, with some pressure being applied in certain cases, is at the core of the processes. In either case, heat control, then, must be *synchronized* with force control and the state of the material must be monitored in-process to determine the right point in the process cycle where heat and/or force application is needed.

One must also take into account the physical limitations imposed by the machinery used for the process. Repeated extreme rates of heat or force application can lead to frequent machine breakdowns and, consequently, make the process economically undesirable.

"Listening" to the material

In-process monitoring is not the act of collecting streams of historical data for off-line analysis; though this is an important step in creating effective process control. When we collect information during a dialogue, we don't necessarily capture every single word and nuance. Rather, we normally record essential points and supporting information that capture the theme or goal of what is being exchanged.

In a similar way, analyzing process data involves the ability, as in listening, to differentiate between steady or normal process behavior and patterns , or *events*, indicating the onset of a change in material state.

As a simple example, Figure 1 shows a plot of sensor data over time during a curing process. Point A indicates the onset of the peak, point C, in the sensor data; while B indicates that we are past the region around C. If C is indicator of an optimum material condition for heat or force application, understanding data patterns in A and/or B can be used to trigger that application.

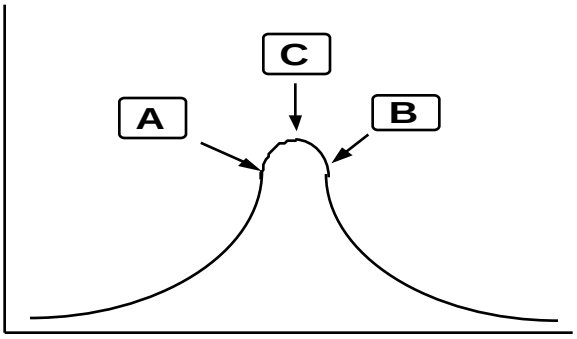


Figure 1

So that, creating a process “recipe” becomes a matter of identifying those key patterns or events, understanding their relationship to heat or force application, and finally linking that application to the occurrence of these events.

A SOLUTION : QPA-CLIPS

QPA-CLIPS, or Qualitative Process Automation CLIPS, is a language with a supporting run-time environment used to describe and run process “recipes” as a set of causal linkages between a specific sensor event(s) and application of a heat or force. The concept of qualitative process analysis was developed as a technique for intelligent control based on interpretation of sensor information [1]. QPA-CLIPS is an implementation of this concept in the CLIPS environment.

Architecture

The execution model of QPA-CLIPS is basically a traversal across a dependency network. The nodes in that network describe one or more sensor events, their related application actions, and one or more pre-conditions which must be satisfied in order to activate or traverse the node.

Traversal through a node consists of 3 phases:

(1) satisfaction of pre-conditions,

(2) detection of sensor events,

(3) performing required heat or force application.

Transition to the next phase cannot occur until successful completion of the current phase. Once phase 3 is completed, one of the remaining nodes in the network is chosen for traversal using the same 3-phase method. If all nodes have been traversed, traversal is halted. Pre-conditions are either traversal through another node, unconditional (i.e. START node), or the successful diagnostic checks on a sensor. The flow chart in Figure 2 summarizes the traversal process.

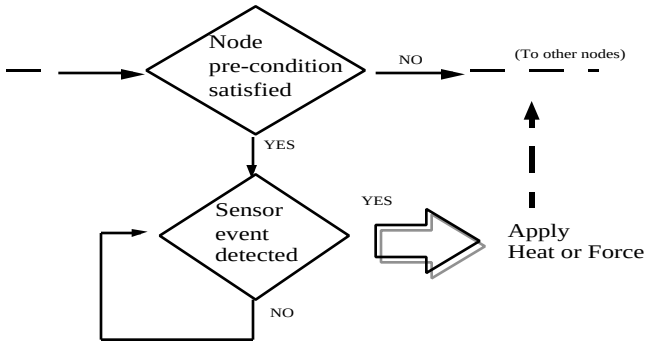


Figure 2

Node traversal is implemented as an interpreter knowledge base exploiting the underlying CLIPS search methods.

There are currently 3 constructs making up the QPA-CLIPS language: TASK, SENSOR, and FILTER. They are defined within the interpreter knowledge base through a set of *deftemplate* 's [2] . The contents of a node are described by the TASK. SENSOR defines sources of raw sensor data; while FILTER applies mathematical functions to SENSOR data. A collection of TASKs describing a dependency network, along with the required SENSORs and FILTERs, is referred to as a *process model* and is consistent with the GRAFCET standard based on Petri Nets [3].

TASK

TASKs encapsulate the pre-conditions for node traversal, sensor event descriptions, and the prescribed heat or force application. In BNF notation form, this translates to:

```
(TASK
   (name <name>)
   (start-when <pre-condition>)
   (look-for <sensor-event>)
   (then-do <application>) )
```

<name> is a unique label used whenever another TASK refers to this TASK throughout the process model. <pre-condition> is a string type that can be either:

```
<task-name> COMPLETE,
```

or

```
<sensor> OK
```

or

```
START
```

In the first form, the pre-condition becomes the completion or traversal through another node, or TASK. The second form checks completion of a diagnostic check for a sensor. Sensor diagnostics are not currently an explicit construct in QPA-CLIPS. Rather, they are embedded in the sensor I/O functions integrated with CLIPS. The third form of <pre-condition> is makes that TASK the initial traversal point in the dependency network.

<sensor-event> is a string type that can have multiple occurrences of the form:

```
<source> <direction> <nominal> <tolerance> <repeat>
```

<source> is the label for a SENSOR or FILTER providing data identifying this event. <direction> can be RISES, DROPS, or HOLDS. The remaining items are numbers describing respectively a threshold value that must be reached, the tolerance band around the nominal value, and the number of times that SENSOR or FILTER data must meet this threshold-and-tolerance-band occurrence without interruption.

So for example, the sensor event clause of

```
flow RISES 50 0.01 3
```

translates to data from SENSOR flow must rise at least 0.01 above 50 cc/min. for 3 consecutive times in order to be identified as this sensor event.

<application> is of the form :

```
<parameter> <amount>
```

where <parameter> is a label for a process parameter and <amount> is a new value or setting for that parameter. An example is :

```
PR 5
```

or set pressure to 5 tons. Application of heat or force then is basically a change in the value of a memory location which, in turn, drives the process.

SENSOR

SENSORs are one of 2 possible sources of data for a sensor event defined within a TASK and defined as follows:

```
(SENSOR
   (name <name>)
   (max-allowed <value>)
   (min-allowed <value>) )
```

<name> is a unique label used whenever a TASK or FILTER refer to this SENSOR throughout the process model. *max-allowed* and *min-allowed* provide the allowable range of values for the sensor data. As an example, a flowmeter within a process model can be defined as:

```
(SENSOR
   (name flowmeter)
   (max-allowed 100)
   (min allowed 5) )
```

The rate of the flowmeter can range between 5 and 100 cc/min. Currently , retrieval of sensor data is accomplished through user-defined functions embedded within CLIPS. Association between a SENSOR and these functions is performed through the QPA-CLIPS interpreter. The implementation of SENSOR assumes use of one sensor for the process. This suffices for current applications of QPA-CLIPS. However, for multiple sensor input, the SENSOR construct will be modified to include a reference to a specific function, referred to a the SENSOR source, or:

```
(SENSOR
   (name <name>)
   (max-allowed <value>)
   (min-allowed <value>)
   (source <function>) )
```

FILTER

FILTERs are the other source of data for a sensor event and is defined as:

```
(FILTER
   (name <name>)
   (max-allowed <value>)
   (min-allowed <value>)
   (change <sensor>)
   (using <formula>) )
```

<name> is a unique label used by a TASK whenever referring to this FILTER in order to identify a sensor event. Both *max-allowed* and *min-allowed* are used in the same manner as in SENSOR. But, here, they refer to the calculated value created by this FILTER. <sensor> is the label referring to the SENSOR supplying raw sensor data to this FILTER.

<formula> is a string describing the combination of mathematical functions used to translate the raw sensor data. so, for example, a formula string of “SLOPE LOG” is the equivalent of :

d(log S)/dt

where S is the raw sensor data.

Future enhancements to the FILTER construct will be the ability for expressing formulas in more natural algebraic terms. So, “SLOPE LOG”, for example, will take on a form like dLOG / dt.

Building a process model

The first step in developing a process model is a through description of how the process runs, what information can be extracted about the process during a cycle run, what sensors are available to extract this information, and how is the sensor information related to application of heat and/or force to the material. These relationships can be investigated through techniques such as Design of Experiments or Taguchi methods. Once they are verified, a dependency network can be build from these relationships in order to specify alternative paths for running the process cycle; depending on the sensor data patterns being extracted. This network can then be encoded as a process model.

As a simple example, a molding process for a panel in a cabinet enclosure is now switching to a new material, requiring that pressure be triggered based on material temperature. For this material, a constant heat setting must be applied throughout the cycle. A sensor, called a thermowidget, was selected to retrieve in-process material temperature. Its operating range is 500˚F and 70˚F. So, its SENSOR definition is:

```
(SENSOR
   (name thermowidget)
   (max-allowed 500)
   (min-allowed 70))
```

Two pressure applications are required by the process: an initial setting to distribute material throughout the mold and a final setting to complete the part. The initial setting must be applied when the material temperature reaches a value of 100˚F and the final setting when the rate of change of the material temperature is 0. We first need a FILTER to define the slope of the temperature or:

```
(FILTER
   (name thermoslope)
   (max-allowed 10)
   (min-allowed -10)
   (change thermowidget)
   (using "SLOPE") )
```

We achieve the pressure application through two TASKs:

```
(TASK
   (name first-setting)
   (start-when "START")
   (look-for "thermowidget RISES 100 0.1 5")
   (then-do "PR 2") )

(TASK
   (name final-setting)
   (start-when "first-setting COMPLETE")
   (look-for "thermoslope DROPS 0 0.01 3")
   (then-do "PR 5") )
```

Another TASK can be added to complete the cycle and release the finished part.

RUN-TIME ENVIRONMENT

A process model is executed through a QPA-CLIPS interpreter: a CLIPS knowledge base consisting of the *deftemplate*'s defining the TASK, SENSOR, and FILTER constructs, a rule set which carries out the interpretation cycle, and *deffunction*'s supporting the rule set. Underlying this knowledge base are the user-defined functions integrating the process model with sensors and process equipment.

The Interpreter

At the core of the interpreter are a set of rules which cycle through the currently available TASKs in a process model and implement the 3-phase method for node traversal mentioned above. When the QPA-CLIPS environment along with the appropriate process model is invoked, control transfers to the TASK containing the pre-condition of START. From that point on, the node traversal method takes over. The QPA-CLIPS interpreter knowledge base can be ported to any CLIPS-compatible platform.

Sensor and controller integration

Sensors are integrated through user-defined functions embedded within CLIPS. They currently interact with sensor hardware through serial (RS-232-C) communications. These functions are the only platform dependent components within QPA-CLIPS.

Integration of heat and force application is done through a memory mapping paradigm between sensor events and parameter settings where TASK actions, as defined in the then-do clause, are mapped to the memory location of another control system (i.e. a PLC) which, in turn, directly

drives process machinery (see Figure 3). These new settings then trigger signals to heaters or motors driving the equipment.

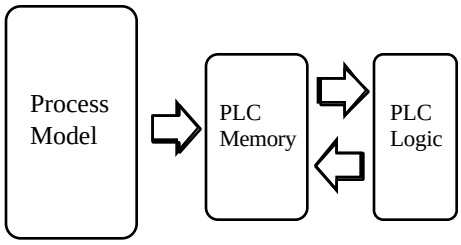


Figure 3

CURRENT STATUS

A version of QPA-CLIPS has been demonstrated on a composite molding application. The system consists of a SPARCStation 2 running QPA-CLIPS linked to an Allen-Bradley PLC5/40 driving a molding press. It has successfully created a number of production quality parts for a jet engine.

FUTURE ENHANCEMENTS

Though the current version of QPA-CLIPS has been demonstrated to work successfully, several opportunities for enhancements have been mentioned in the description of its constructs. In addition, other opportunities for enhancements exist in the development and run-time environments of QPA-CLIPS.

6.1 A GUI for QPA-CLIPS

Currently, a process model is created through the use of a text editor, such as emacs or textedit. The completed model is tested on a simulated run-time environment. Taking a cue from the visual development tools such as those found under Microsoft Windows, a graphical development environment for a process model will greatly ease the development of a process model. The syntax of the QPA-CLIPS constructs are simple enough for process engineers to work with. Nevertheless, as process models grow in complexity, a need will arise for easy-to-use model navigation and debugging tools seamlessly connected to the simulated run-time environment.

Automating process model creation

Experience with developing process models has shown that the bulk of the time is actually spent in experimentation; trying to establish the relationships between sensor data and heat/force application points. What is needed to streamline this aspect of the model creation process is the partial or complete automation of the experimentation phase leading to the automatic creation of a process model; since a new material system will require a new process model. The work on DAT-GC by Thompson et al [4] offer a good start in that direction.

Exploiting fuzzy linguistics

In the definition of the *look-for* clause of a TASK construct, one needs a rather accurate description of not only the goal or threshold value to be reached, but also a tolerance band around that value as well as a repeatability count on the number of times in a row that the event must

appear. There are cases where a less exacting or *fuzzy* [5] description will suffice. So, using our example model from 3.2 above, the *look-for* clause for the final pressure application can then be:

```
(look-for “thermoslope dropping IMMEDIATELY to ZERO”)
```

IMMEDIATELY is a fuzzy term similar to the repeatability count. ZERO, on the other hand, describes a fuzzy term associated with a range of precise values for a SENSOR or FILTER, which can be described with a new QPA-CLIPS construct such as:

```
(BEHAVIOR
   (name <name>)
   (for <source>)
   (range <value-list>) )
```

where <name> is the fuzzy term, <source> is the label for the SENSOR or FILTER, and <value-list> contains the values defining the membership function [5] for this BEHAVIOR.

Exploiting parallelism

It goes without saying that node traversal within a process model is inherently a method that can easily converted to parallel processing. Having a parallel version of CLIPS operating in an affordable parallel processing platform will enable QPA-CLIPS to easily operate in a process control scenario requiring multiple sensors.

CONCLUSIONS

QPA-CLIPS, an extension of CLIPS for process control, is a proven tool for use by process engineers in developing control models for forming or curing processes. Its simple syntax and integration into CLIPS provides a powerful, yet straightforward, way to describe and apply control of a process driven by sensor events, or distinct patterns in sensor data. Several enhancements have been suggested and currently strong interest exist in the automatic generation of process models and fuzzifying the description of sensor events.

REFERENCES

1. Park, Jack, Toward the development of a real-time expert system, 1986 Rochester FORTH Conference Proceedings, Rochester NY, June 1986, pp. 23-33

2. CLIPS Reference Manual, Vol. I, Basic Programming Guide, CLIPS Version 5.1, Sept. 1991

3. David, Rene and Hassane Alla, Petri nets for modeling of dynamic systems - a survey, *Automatica*, Vol. 30, No. 2, pp.175-202, 1994

4. Levinson, Richard, Peter Robinson, and David E. Thompson, Integrated perception, planning and control for autonomous soil analysis, Proc. CAIA-93

5. Terano, Toshiro, Kiyoji Asai, and Michio Sugeno, Fuzzy Systems Theory and its applications, Academic Press, 1992

AN IMPLEMENTATION OF FUZZY CLIPS AND ITS APPLICATIONS

Thach C. Le
The Aerospace Corporation
Information Technology Dept., M1-107
2350 E. El Segundo Bl.
El Segundo, CA 90245-4691

Fuzzy Logic expert systems can be considered as an extension of traditional expert systems. Their capability to represent and manipulate linguistic variables is a desirable feature in the systems, such as rule-based expert systems, whose design is to capture human heuristic knowledge. The C-Language Integrated Production System (CLIPS) developed by NASA, is a boolean expert system shell, which has a large user base and been used to develop many rule-based expert systems. Fuzzy CLIPS (FCLIPS), developed by The Aerospace Corporation, is a natural extension of CLIPS to include fuzzy logic concepts. FCLIPS is a superset of CLIPS, whose rules allow mixture of fuzzy and boolean predicates. This paper discusses the current implementation of FCLIPS, its applications, and issues for future extensions.

UNCERTAINTY REASONING IN MICROPROCESSOR SYSTEMS USING FUZZYCLIPS

S.M. Yuen And K.P. Lam
Department of Systems Engineering
The Chinese University of Hong Kong
Shatin, New Territories, Hong Kong

A diagnostic system for microprocessor system design is being designed and developed. In microprocessor system diagnosis and design, temporal reasoning of event changes occurring at imprecisely known time instants is an important issue. The concept of time range, which combines the change-based and time-based approaches of temporal logic, has been proposed as a new time structure to capture the notion of time imprecision in event occurrence. According to this concept, efficient temporal reasoning techniques for time referencing, constraint satisfaction and propagation of time ranges have been developed for embedding domain knowledge in a deep-level constraint model. The knowledge base of the system parameters and the temporal reasoning techniques are implemented in CLIPS. To handle the uncertainty information in microprocessor systems, fuzzy logic is applied. Each imprecision of time contributes to some uncertainties or risks in a microprocessor system. The fuzzyCLIPS package has been used to determine the risk factor of a design due to the uncertain information in a microprocessor system. A practical MC68000 CPU-memory interface design problem is adopted as the domain problem. The sequence of events during a read cycle is traced through an inference process to determine if any constraint in the model is violated.

NEURAL NET CONTROLLER FOR INLET PRESSURE CONTROL OF ROCKET ENGINE TESTING

Luis C. Trevino
NASA-MSFC, Propulsion Laboratory
Huntsville, Alabama

ABSTRACT

Many dynamical systems operate in select operating regions, each exhibiting characteristic modes of behavior. It is traditional to employ standard adjustable gain PID loops in such systems where no apriori model information is available. However, for controlling inlet pressure for rocket engine testing, problems in fine tuning, disturbance accommodation, and control gains for new profile operating regions (for R&D) are typically encountered [2]. Because of the capability of capturing i/o peculiarities, using NETS, a back propagation trained neural network controller is specified. For select operating regions, the neural network controller is simulated to be as robust as the PID controller. For a comparative analysis, the Higher Order Moment Neural Array (HOMNA) method [1] is used to specify a second neural controller by extracting critical exemplars from the i/o data set. Furthermore, using the critical exemplars from the HOMNA method, a third neural controller is developed using NETS back propagation algorithm. All controllers are benchmarked against each other.

INTRODUCTION

An actual propellant run tank pressurization system is shown in Figure 1.1 for liquid oxygen (LOX). The plant is the 23000 gallon LOX run tank. The primary controlling element is an electro-hydraulic (servo) valve labelled as EHV-1024. The minor loop is represented by a valve position feedback transducer (LVDT). The major or outer loop is represented by a pressure transducer (0-200 psig). The current controller is a standard PID servo controller. The reference pressure setpoint is provided by a G.E. Programmable Logic Controller. The linearized state equations for the system are shown below:

$$x_1 = x_2 - (0.8kg+c)x_1 \quad (1.1)$$

$$x_2 = 5kg\ au - (0.8kg\ c+d)x_1 + x_3 \quad (1.2)$$

$$x_3 = 5abkg\ u - (0.8kg\ d+f)x_1 + x_4 \quad (1.3)$$

$$x_4 = -0.8kg\ fx_1 \quad (1.4)$$

where kg=1, servo valve minimum gain. Based on previous SSME test firings, the average operating values for each state variable are determined to be

$x_1 = P_B$:0–76 psig

$x_2 = T_u$:150–300R˚

$x_3 = V_u$:250–350 ft^3

x_4 = L:0–1 inch

where

P_B = bottom tank pressure
T_u = ullage temperature
V_u = ullage volume
L = valve stem stroke length

Using those ranges, the following average coefficients are algebraically determined:

a = 120.05
b = 89.19
c = 214.30
d = 5995.44
f = 14.70

METHODOLOGY

1. Using a developed PID-system routine from [2], an i/o histogram is established in the required format per [1] for a select cardinality. Figure 2.1 portrays the scheme. A ramp control setpoint signal (from 0-120 psig) served as the reference trajectory.

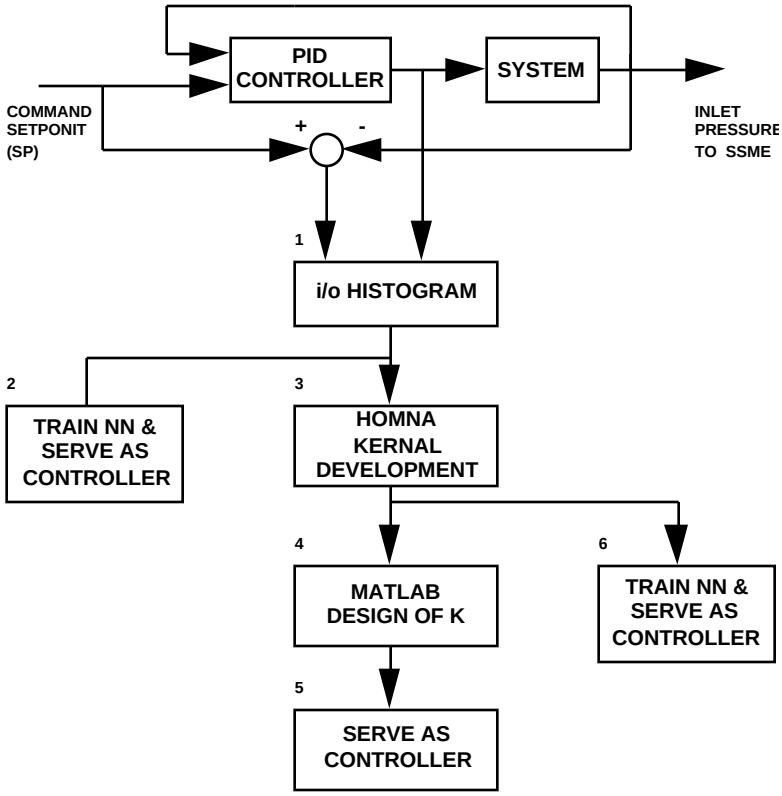


Figure 2.1. Scheme For Building i/o Histogram and Training Set.

2. Using the captured i/o data set and NETS back propagation algorithm, a neural network is next established. The trained network is next simulated as the controller for the system. Figure 2.2 illustrates the simulation scheme.

3. Using a developed HOMNA (KERNALS) algorithm [1], a reduced training i/o set is specified. The input portion of the set, "S", will provide the mapping of real time system

inputs to the neural net controller (NNC). The output segment of the set is represented by the last column vector of the i/o set.

4. After configuring the reduced i/o set into the needed formats, Using MATLAB, the gain equation is executed per [1].

$$K = YG^{-1} = Y(SS^{*}) \qquad (2.1)$$

where

K = neural gains for single neuron layer
Y = NNC controller output signature vector
S = established matrix set of part 3, above
= any (decoupled) operation: exponential, etc.

For this project, was identical with that used in the literature of [1], namely the exponential function. “K” serves as a mapping function of the input, via “S” to the NNC output, u(j). Here, u(j) serves as control input to the system and is determined by equation (2.2) [1].

$$u(j) = K(Sx(j)) \qquad (2.2)$$

where x(j) represents the input. For this project,a five dimensional input is used and is accomplished using successive delays. The overall HOMNA scheme is embedded in the neural controller block of Figure 2.2

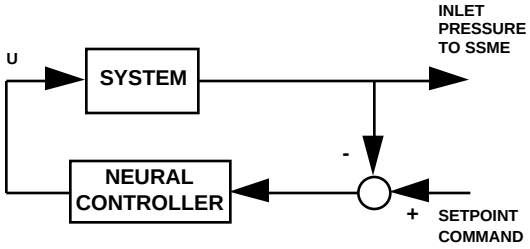


Figure 2.2. Simulation Scheme for NNC and System

5. For select cases to be presented, integral control was presented according to the following scheme of [1].

$$= \frac{1}{N} \sum [y(j) - \bar{Y}(j)] \qquad (2.3)$$

where

N = window (sampling) size
y(j) = current system output
$\bar{y}$(j) = desired output, or command setpoint, sp

RESULTS

Case	System	Command Setpoint	Noise	Integral Control
1	Neural Net	Ramp	None	n/a
2	HOMNA	Ramp	None	None
3	HOMNA	Ramp	None	None
4	PID	Ramp	None	Present
5	Neural Net	Ramp	Present	n/a
6	HOMNA	Ramp	Present	Present
7	PID	Ramp	Present	Present
8	Neural Net	Profile	None	n/a
9	HOMNA	Profile	None	None
10	HOMNA	Profile	None	Present
11	PID	Profile	Present	Present
12	Neural Net	Profile	Present	n/a
13	HOMNA	Profile	Present	None
14	HOMNA	Profile	Present	Present
15[1]	Neural Net	Ramp	None	n/a

Table 1. Case Summary

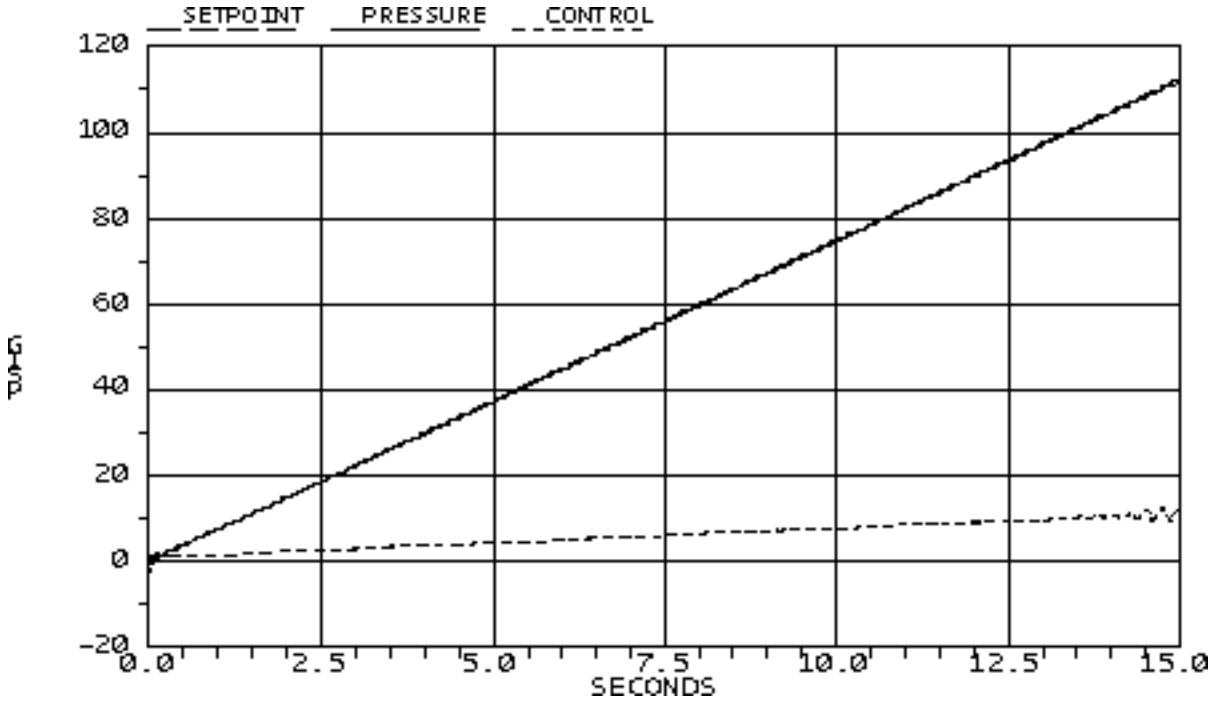


Figure 3.1. Case 1, 3, and 4 Simulation Results

[1] Case for procedural step 6.

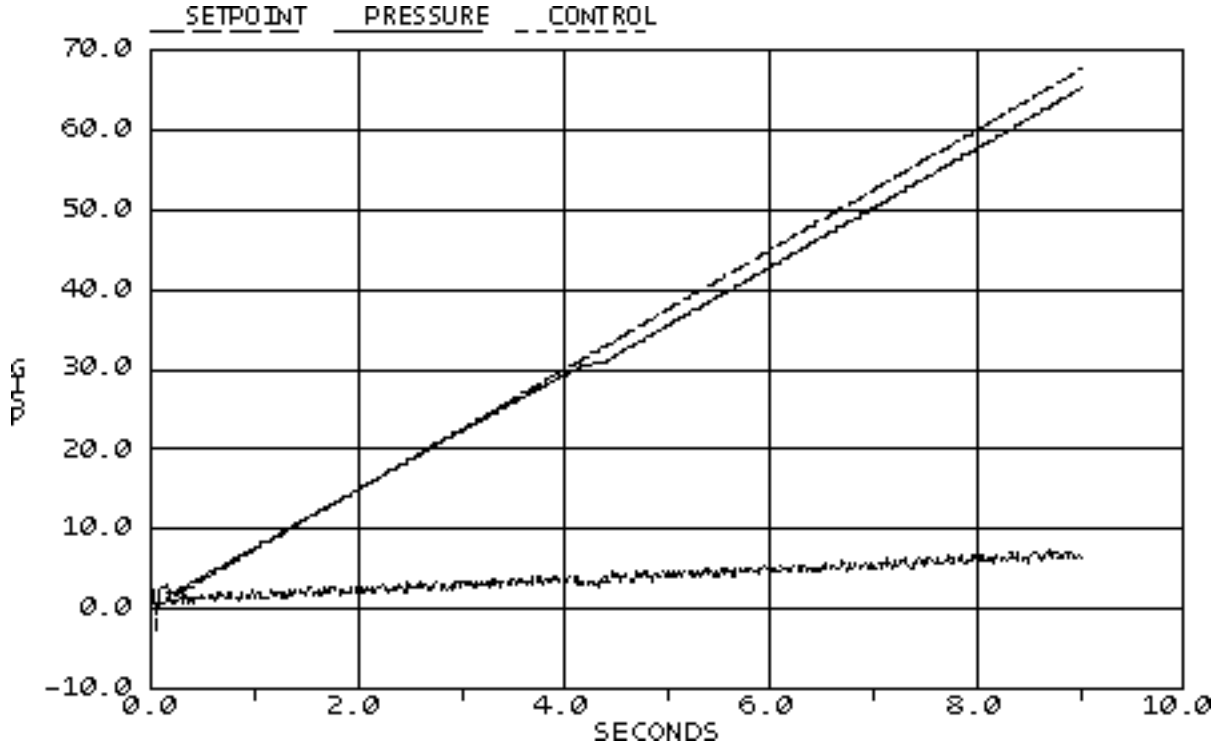


Figure 3.2. Case 2 Simulation Results

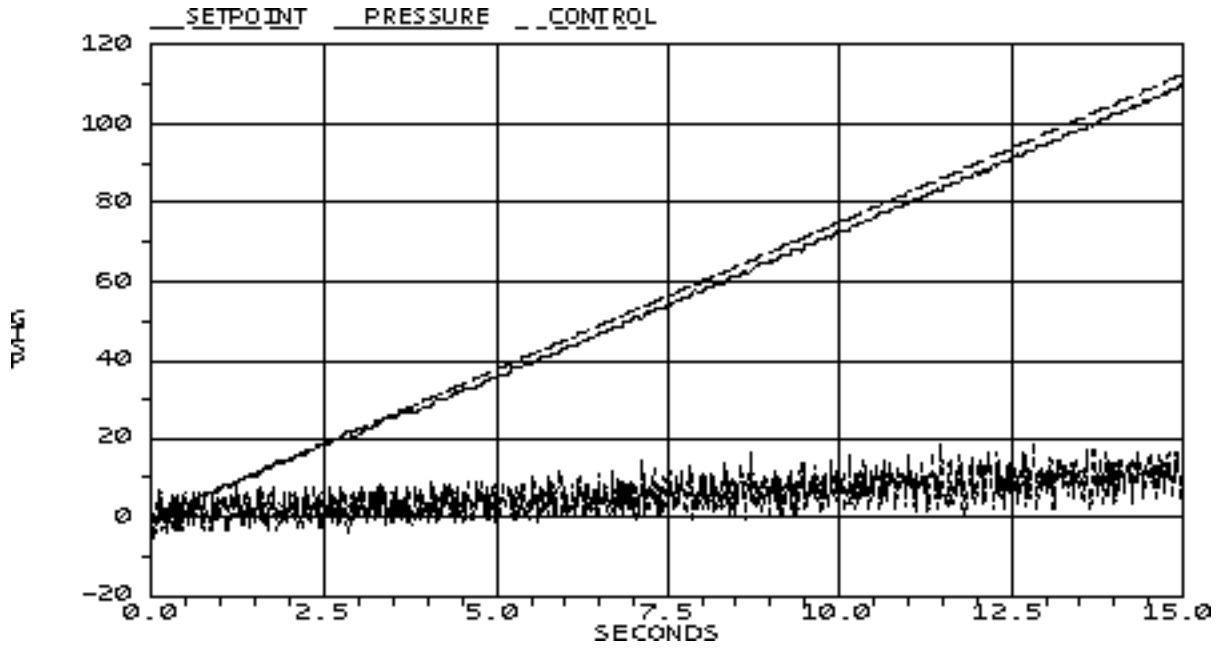


Figure 3.3. Case 5 Simulation Results

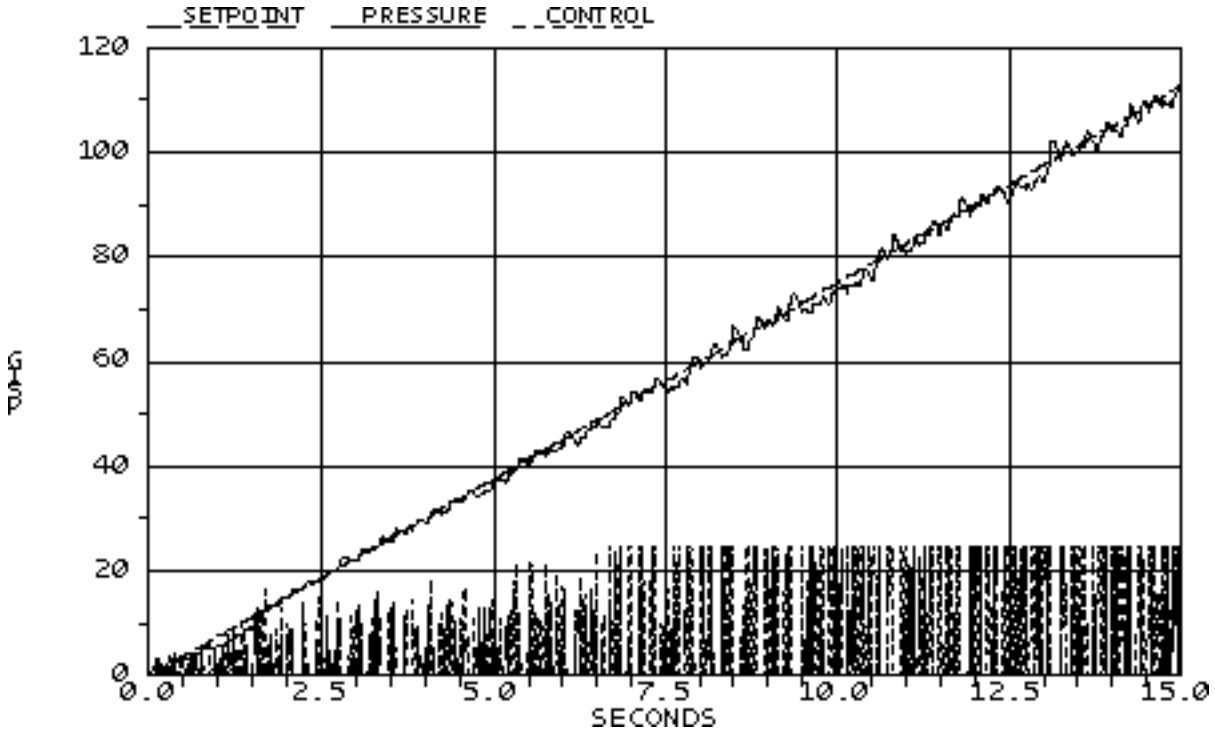


Figure 3.4. Case 6 Simulation Results

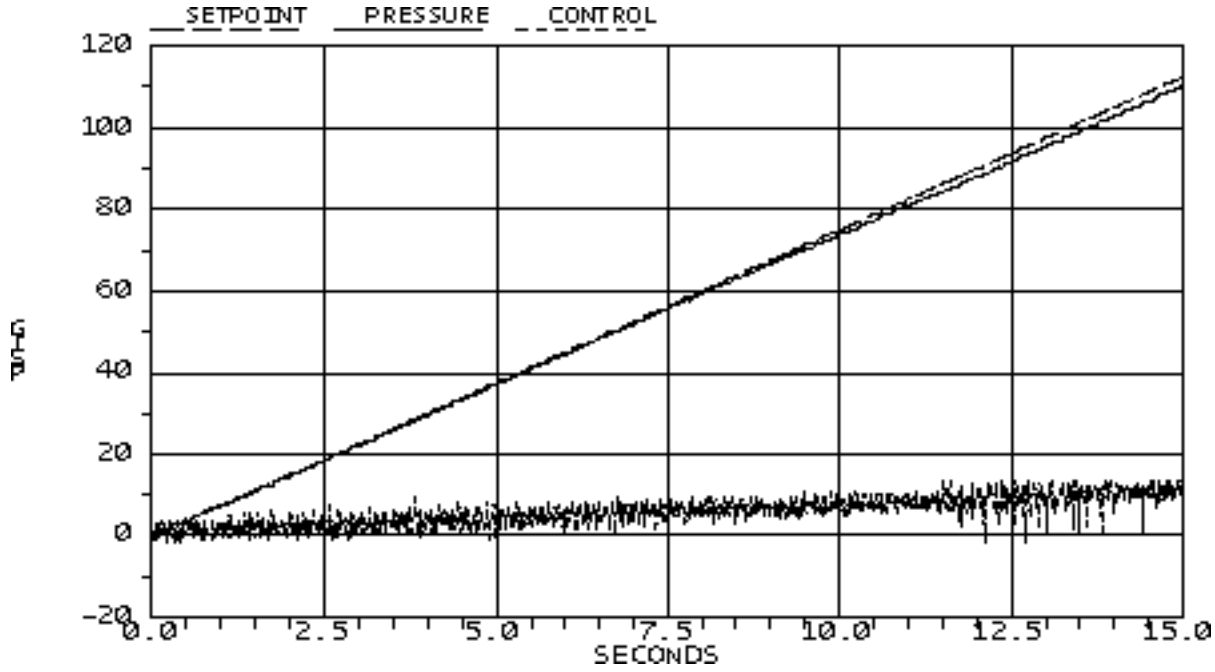


Figure 3.5. Case 7 Simulation Results

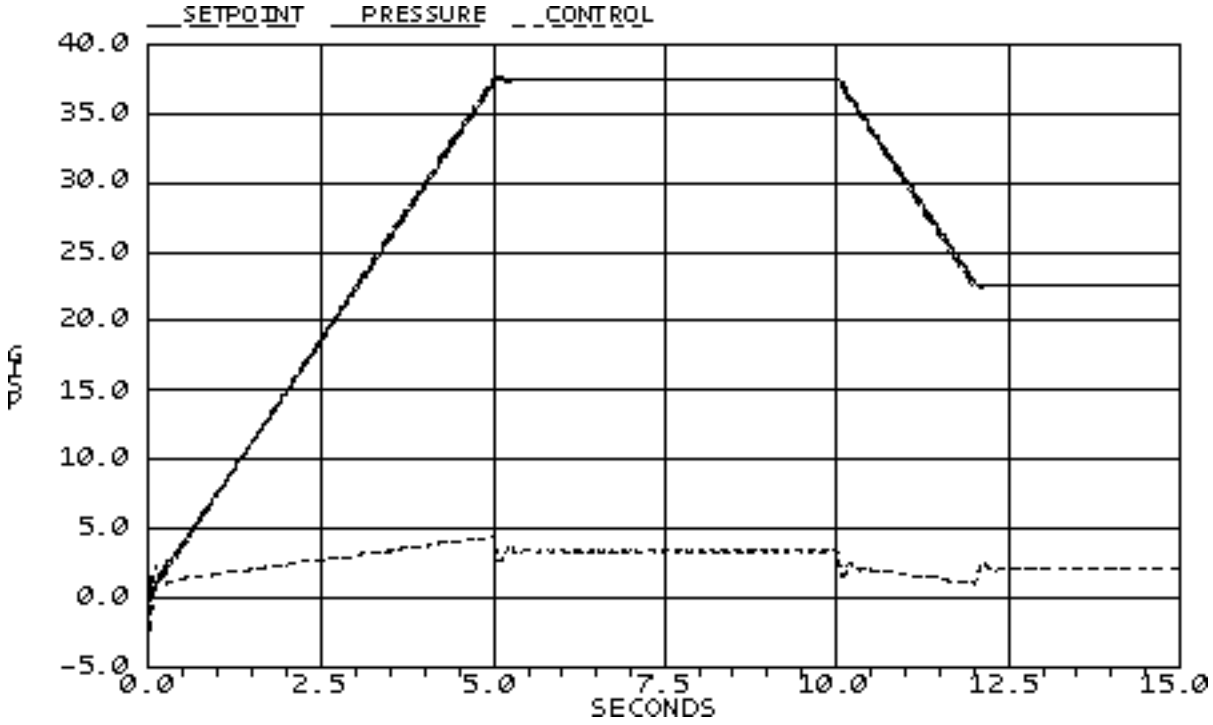


Figure 3.6. Case 8 and 10 Simulation Results

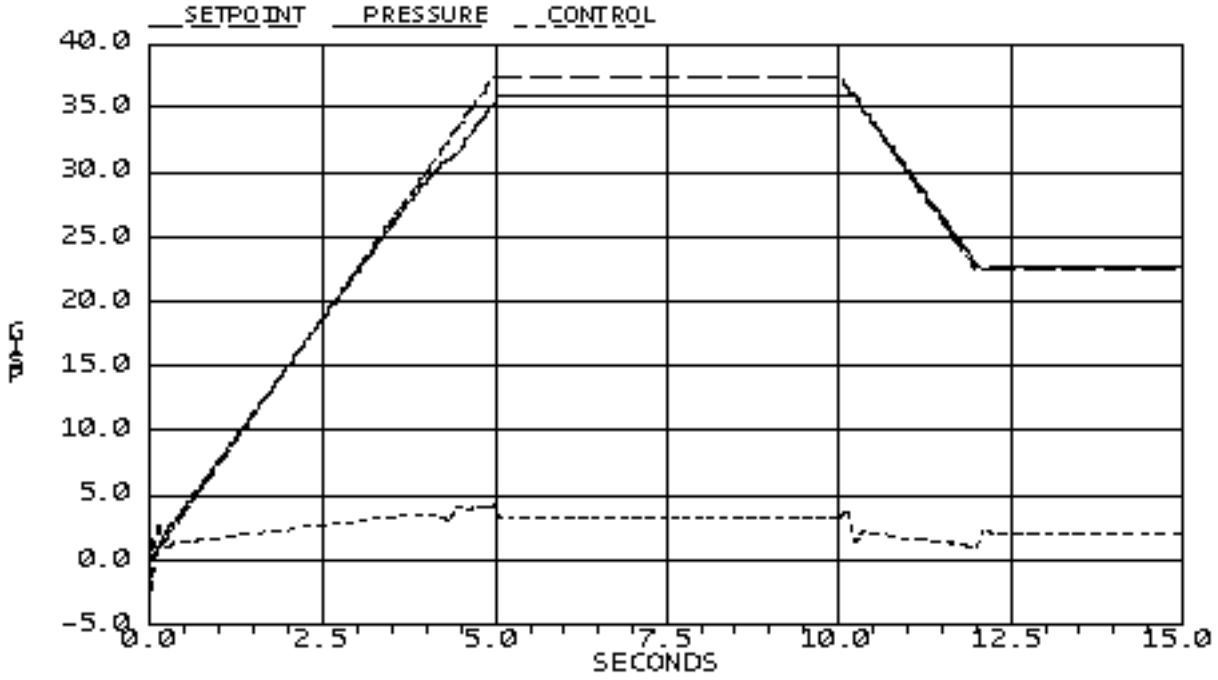


Figure 3.7. Case 9 Simulation Results

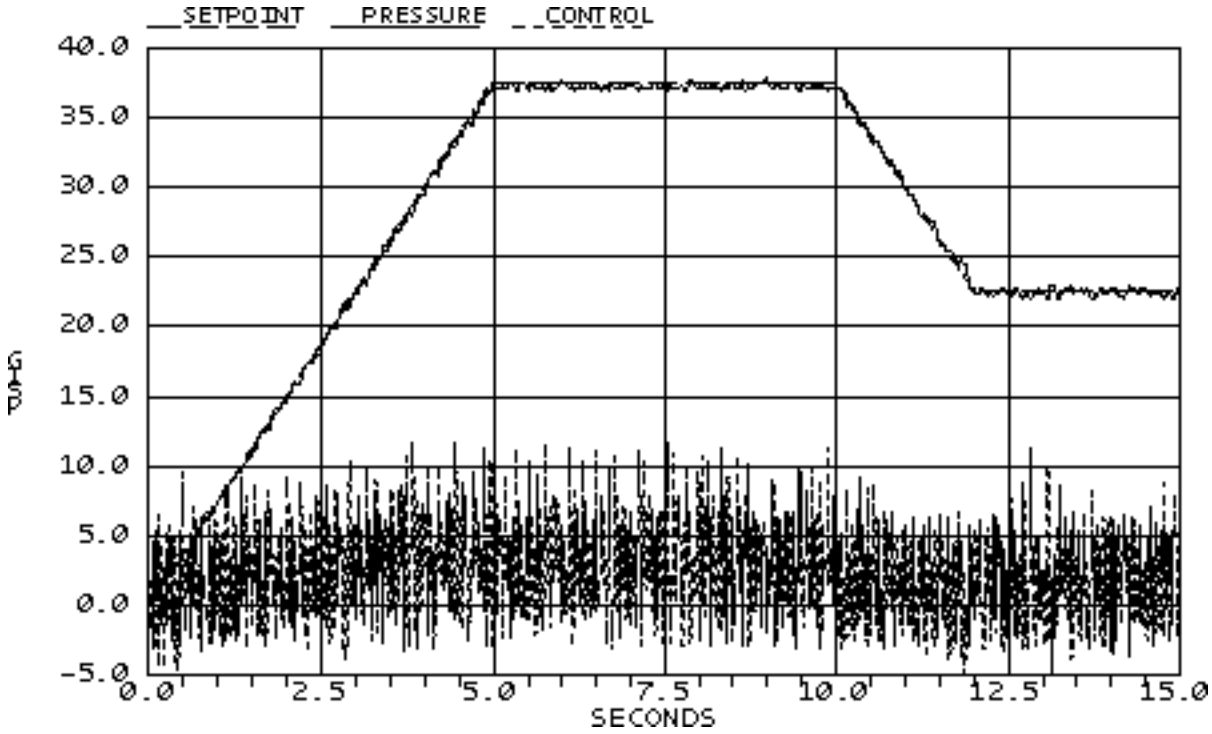


Figure 3.8. Case 11 Simulation Results

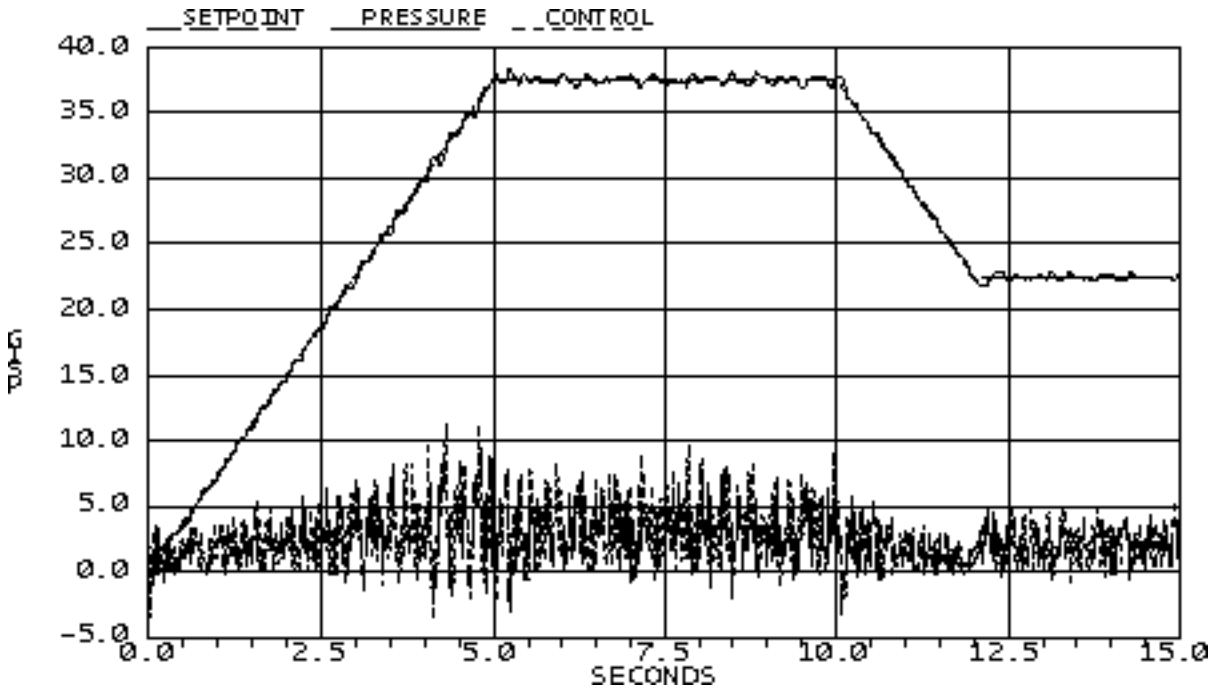


Figure 3.9. Case 12 and 14 Simulation Results

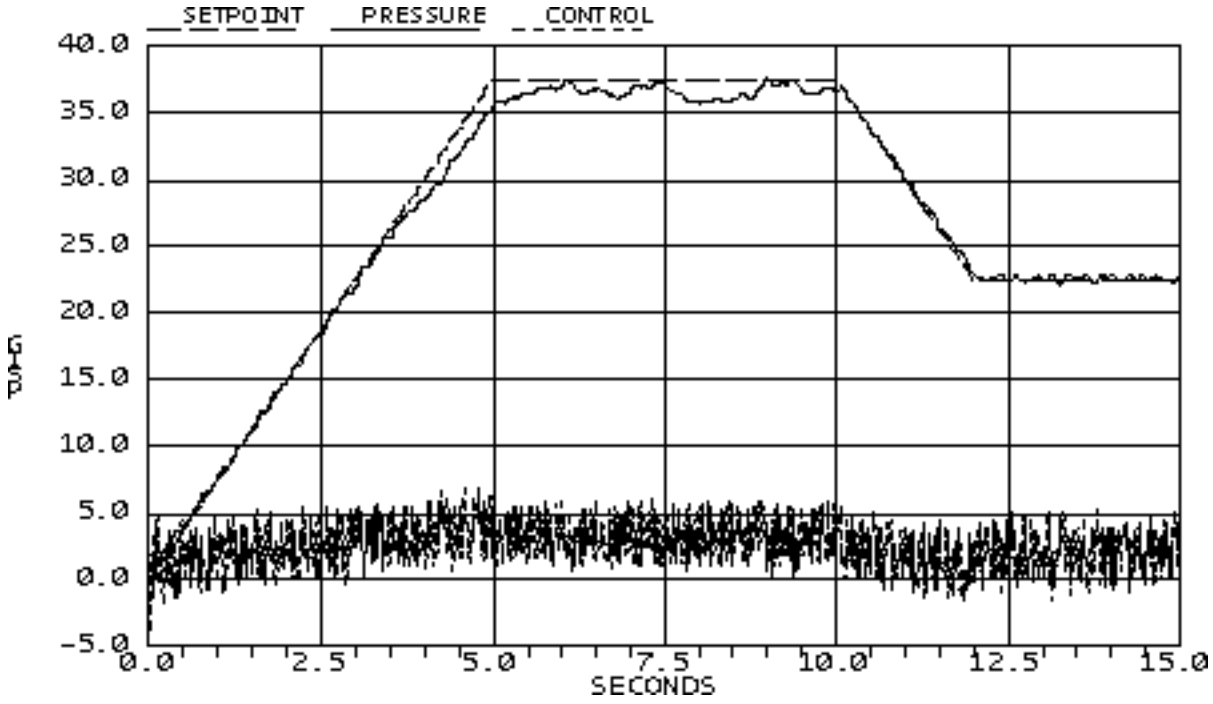


Figure 3.10. Case 13 Simulation Results

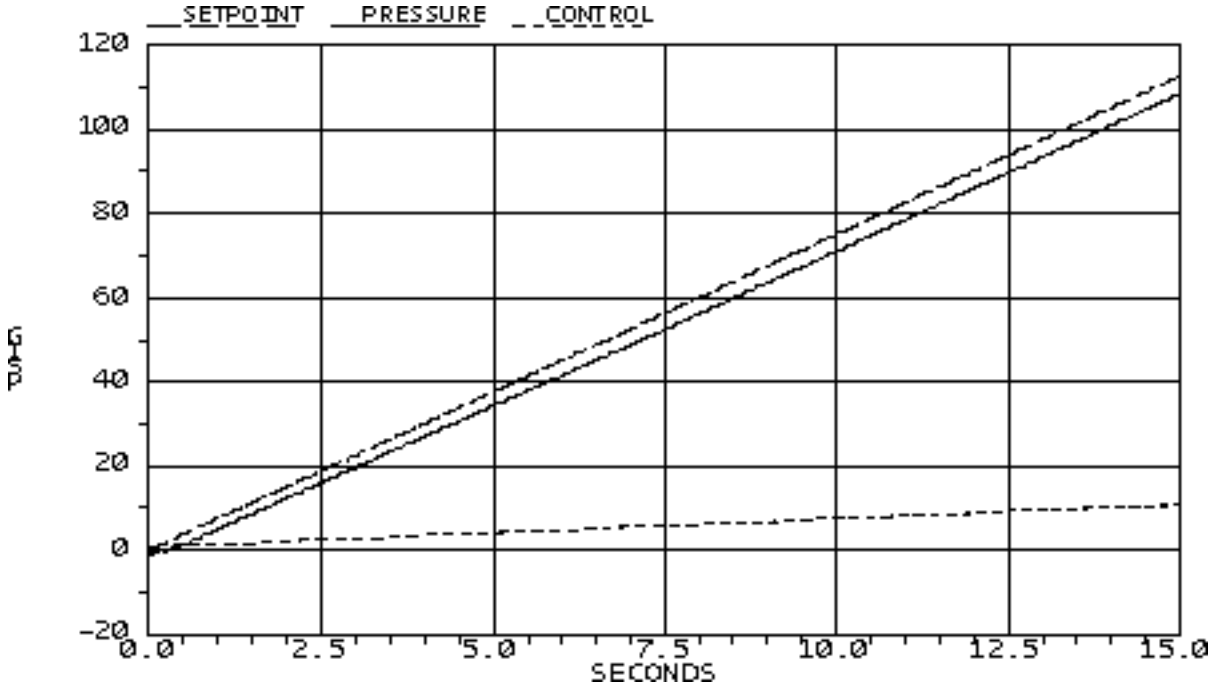


Figure 3.11. Case 15 Simulation Results

DISCUSSION AND CONCLUSIONS

From Table 1 and the presented simulation results, the back-propagation trained and the HOMNA neural system are proven to track varying command setpoints within the bounds of the training i/o histogram. Without incorporation of the integration scheme of [1], the HOMNA

system still proved its tracking ability, though with varying levels of offsets. The proportional-integral-derivative (PID) system results exhibited no offsets due to the inherent integral scheme of the PID controller. Based on the simulation results, it is concluded that the integration scheme of [1] was simpler to employ with equally satisfying results (i.e., a smoother effect). Both back-prop trained, HOMNA and PID systems proved their ability to accommodate for the varying levels of random noise injection.

Though not shown in the table, the original i/o histogram was of cardinality 300+. Larger i/o histogram sets were attempted with no significant difference in performance for select cases. With more effort or other techniques, it is believed that the difference could be corrected.

In this project it was discovered that stripping the first few exemplar vectors from the i/o histogram (or the established training set) made a significant difference in the performance. For some cases, without stripping the first few inherent exemplar state vectors resulted in erroneous results ranging from wide dispersion (between setpoint and system state) to complete instability. The justification for stripping the first few exemplars stems from the scheme of [1]. That is, for the first few exemplars there is always inherent membership in the training set kernal. For select cases, the effects of stripping the exemplars before or after the Kernals algorithm software routine had no indicative difference. Overall, the choice of a back-prop trained or a HOMNA based neural controller is certainly realizable.

REFERENCES

1. Porter, W. A. and Liu, Wie, "Neural Controllers For Systems With Unknown Dynamics," The Laboratory for Advanced Computer Studies, The University of Alabama in Huntsville, March 1994.

2. Trevino, L. C., "Modeling, "Simulation, and Applied Fuzzy Logic For Inlet Pressure Control For A Space Shuttle Main Engine At Technology Test Bed," *Masters Thesis*, The University of Alabama in Huntsville, June, 1993.

A CLIPS TEMPLATE SYSTEM FOR PROGRAM UNDERSTANDING

Ronald B. Finkbine, PhD.
Department of Computer Science
Southeastern Oklahoma State University
Durant, OK 74701
finkbine@babbage.sosu.edu

ABSTRACT

Program Understanding is a subfield of software re-engineering and attempts to recognize the run-time behavior of source code. To this point, the success in this area has been limited to very small code segments. An expert system, HLAR (High-Level Algorithm Recognizer), has been written in CLIPS and recognizes three sorting algorithms, Selection Sort, Quicksort and Heapsort. This paper describes the HLAR system in general and, in depth, the CLIPS templates used for program representation and understanding.

INTRODUCTION

Software re-engineering is a field of Computer Science that has developed inconsistently since the beginning of computer programming. Certain aspects of what is now software re-engineering have been known by many names: maintenance, conversion, rehosting, reprogramming, code beautifying, restructuring, and rewriting. Regardless of the name, these efforts have been concerned with porting system functionality and/or increasing system conformity with programming standards in an efficient and timely manner. The practice of software re-engineering has been constrained by the supposition that algorithms written in outdated languages cannot be re-engineered into robust applications with or without automation.

It is believed by this author that existing software can be analyzed, data structures and algorithms recognized, and programs optimized at the source code level with expert systems technology using some form of intermediate representation. This intermediate form will provide a foundation for common tool development allowing intelligent recognition and manipulation. This paper describes a portion of the HLAR (High-Level Algorithm Recognition) system [1].

The Levels of *understanding* in the software re-engineering and Computer Science fields is displayed in Figure 1 [2]. The lowest of these, *text*, is realized in the simple file operations; opening, reading, writing and closing. The next level is *token* understanding and occurs in compilers within their scanner subsystem. Understanding at the next level, *statement*, and higher generally does not occur in most compilers, which tend to break statements into portions and correctly translate each portion, and, therefore, the entire statement. One exception is the *semantic* level which some compilers perform when searching for syntactically correct but logically incorrect segments such as *while (3 < 4) do S*.

[7] Program
[6] Plan
[5] Semantic
[4] Compound Statement
[3] Statement
[2] Token
[1] Text

Figure 1. Levels of Understanding

Software re-engineering requires that higher-level understanding take place and the act of understanding should be performed on an intermediate form. The representation method used in this research project is the language ALICE, a very small language with a Lisp-like syntax. It is intended that programs in existing high-level languages be translated into ALICE for recognition and manipulation. This language has only five executable statements; *assign, if, loop, call* and *goto*. All syntactic sugar is removed, replaced with parentheses. Figure 2 displays a simple assignment statement. The *goto* statement is used for translating unstructured programs and the goal is to transform all unstructured programs into structured ones.

```
(assign x 0)
```

Figure 2. Assignment Statement

FACT REPRESENTATION

Prior to initiation of the HLAR system, programs in the ALICE intermediate form are translated into a set of CLIPS facts which activate (be input to and fulfill the conditions of) the low-level rules. Facts come in different types called templates (records) which are equivalent to frames and are ``asserted" and placed into the CLIPS facts list. Slots (fields) hold values of specified types (integer, float, string, symbol) and have default values. As a rule fires and a fact, or group of facts, is recognized, a new fact containing the knowledge found will have its slots filled and asserted. Continuation of this process will recognize a larger group of facts and, hopefully, one of the common algorithms.

The *assign* statement from the previous Figure is translated into the equivalent list of facts in Figure 2. This is a much more complicated representation, and not a good one for programmers, but much more suitable for an expert system.

```
(general_node
  (number 1)
  (sibling 0)
  (address "2.10.5")
  (node_type assign_node))
(assign_node
  (number 1)
  (lhs_node 1)
  (rhs_node 1)
  (general_node_parent 1))
(identifier_node
  (number 1)
  (operand 2)
  (name "x"))
(expression_node
  (number 1)
  (operand_1 2))
(integer_literal_node
  (number 1)
  (operand 2)
  (value 1))
```

Figure 3. Facts List

The code of this Figure displays a number of different templates. The *general_node* is used to keep all statements in the proper order within a program and the *node_type* slot describes the type of statement associated with the node. The template *assign_node* with its appropriate *number* slot , its *lhs_slot* slot points to the number one *operand_node*, which is also pointed to by the operand slot in the number one *identifier_node*. The *rhs_node* slot of the assignment points to the number one *expression_node* and its *operand_1* slot points to the number two *operand_node* which is also pointed to by the operand slot in the number one *integer_literal_node*.

This is a more complex representation of a program than its ALICE form, but it is in a form that eases construct recognition. An ALICE program expressed in a series of CLIPS facts is a list and requires no recursion for parsing. Each type of node (i.e. *general*, *assign*, *identifier*, *integer_literal*, *operand*, *expression*) are numbered from one and referenced by that number. Once the ALICE program is converted into a facts list, low-level processing can begin.

TEMPLATE REPRESENTATION

Templates in CLIPS are similar to records or frames in other languages. In the HLAR system, templates are used in a number of different areas including general token, *plan* and knowledge representation. A properly formed template has the general form of the keyword *deftemplate* followed by the name of the template, an optional comment enclosed in double quotes and a number of *field* locations identifying attributes of the template. Each attribute must be of the simple types in the CLIPS system: *integer*, *string*, *real*, *boolean* or *symbol* (equivalent to a Pascal enumerated type) and default values of all attributes can be specified.

Currently there are three types of template recognition; *general* templates are used to represent the statement sequence that constitute the original program., *object* templates are used to represent the clauses, statements and algorithms that are recognized, and *control* templates are used to contain the recognition process.

GENERAL TEMPLATES

The names of all the general templates are listed in Figure 4. These are the templates that are required to represent a program in a list of facts having been translated from its original language which is similar to a compiler derivation tree.

```
general_node
argument_node
assign_node
call_node
define_routine_node
define_variable_node
define_structure_node
evaluation_node
expression_node
if_node
loop_node
parameter_node
program_node
identifier_node
integer_literal_node
struc_ref_node
struc_len_node
```

Figure 4. General Templates

The *general_node* referenced in this Figure is used to organize the order of the statements in the program. Prior to the HLAR system being initialized, a utility program parses the ALICE program depth-first and generates the facts list needed for analysis. The *general_node* template is utilized to represent the order of the statements and contains no pointers to statement nodes. The statement-type nodes identified in the next section contain all pointers necessary to maintain program structure. Each token-type has its own templates for representation within an ALICE program. Included are node types for routine definitions, the various types of compound and simple statements, and the attributes of each of these statements. The various types of simple statement nodes contain pointers back to the *general_node* to keep track of statement order. These statement node templates contain the attributes necessary to syntactically and semantically reproduce the statements as specified in the original language program, prior to translation into ALICE.

In an effort to reduce recognition complexity, which is the intent of this research, specialty templates for each item of interest within a program have been created. An earlier version of this system had different templates for each form, instead of one template with a symbol-type field for specification. This refinement has reduced the number of templates required, thus reducing the amount of HLAR code and the programmer conceptual-difficulty level. The specialty templates are listed in Figure 5.

```
spec_call
spec_parameter
spec_exp
spec_assign
loop_algorithm
spec_eval
minimum_algorithm
swap_algorithm
if_algorithm
sort_algorithm
```

Figure 5. Special Templates

OBJECT TEMPLATES

Expressions are the lowest-common denominator of *program understanding*. They can occur in nearly all statements within an ALICE program. To reduce the complexity of *program understanding*, each expression for which we search is designated as a special expression with a specified symbol-type identifier. Expressions and structure references present a particular problem since they are mutually recursive as expressions can contain structure references and structure references can contain expressions. This problem came to light as the HLAR system became too large to run on the chosen architecture (out of memory) with the number of templates, rules and facts present at one time. Separation of the rules into several passes required that expressions and structure references be detected within the same rule group. Examples *a[i]* and *a[i+1] > a[i]* represent these concepts.

Figure 6 contains the specialty template for expressions. This template contains a number of fields for specific values, but most of interest is the field *type_exp*. Identifiers that represent specific expressions are listed as the allowed symbols. Complexity is reduced in this

representation since multiple versions of the same statement can be represented by the same symbol. An example is two versions of a variable increment statement; *x = x + 1* and *x = 1 + x*.

```
(deftemplate spec_exp
  (field type_exp
    (type SYMBOL) (default exp_none)
    (allowed-symbols
      exp_none exp_0 exp_1 exp_id
      exp_first exp_div_id_2
      exp_plus_id_1 exp_minus_id_1
      exp_plus_id_2
      exp_minus_id_2 exp_mult_id_2
      exp_plus_div_id_2_1
      exp_minus_div_id_2_1
      exp_div_plus_id_id_2
      exp_ge_id_id exp_gt_const_id
      exp_gt_id_id exp_gt_id_const
      exp_gt_id_minus_id_1
      exp_lt_id_minus_id_1
      exp_gt_strucid_strucid
      exp_lt_strucid_strucid
      exp_lt_strucid_struc_plus_id_1
      exp_gt_strucid_id
      exp_ne_id_true
      exp_or_gt_id_min_id_1_ne_id_t
      exp_or_gt_id_id_ne_id_t
      exp_strucfirst))
  (field id_1 (type INTEGER) (default 0))
  (field id_2 (type INTEGER) (default 0))
  (field id_3 (type INTEGER) (default 0))
  (field exp_nr (type INTEGER) (default 0)))
```

Figure 6. Expression Template

Recognition of various forms of the *assign* statement occurs within the variable rules of HLAR. Common statements such as increments and decrements are recognized, as well as very specific statements such as *x = y / 2 + 1*.

Variable analysis is performed from one or more *assign* statements. The intent is to classify variables according to their usage, thus determining knowledge about the program derived from programmer intent and not directly indicated within the encoding of the program. An example would be saving the contents of one specific location of an array into a second variable such as *small = a[0]*.

Next come the multiple or compound statements such as the *if* or *loop* statements. These statements are the first of the two-phase rules to fire a *potential* rule and an *actual* rule. The *potential* rule checks for algorithm form as is defined against by the standard algorithm. The *actual* rule verifies that the proper statement containment and ordering is present. This allows for smaller rules, detection of somewhat *buggy* source code and elimination of *false positive* algorithm recognition.

The *loop* and *if* statements are the first compound statements that are handled within HLAR. Both have an *eval* clause tested for exiting the *loop* or for choosing the boolean path of the *if*. Both of these statements require a field to maintain the limits of the control of the statement. Generally, a *loop* or *if* statement will contain statements of importance within them and the concept of statement containment will be required to be properly accounted for.

Higher-level algorithms, such as a *swap*, *minimization*, or *sort*, require that *subplans* be recognized first. This restriction is due to the size of the Clips rules. The more conditional elements in a rule, the more difficult and unwieldy for the programmer to develop and maintain.

Control templates are listed in Figure 7. These are used to control the recognition cycle within Clips. Control templates are necessary after preliminary recognition of a *plan* by a *potential* rule to allow for detection of any intervening and interfering statements prior to the firing of the corresponding *actual* rule.

```
not_fire
assert_nots_scalar
asserts_not_struc
```

Figure 7. Control Templates

Figure 8 is a hierarchical display of the facts from the previous Figure. It expresses the relationships among the nodes and shows utilization of the integer pointers used in this representation.

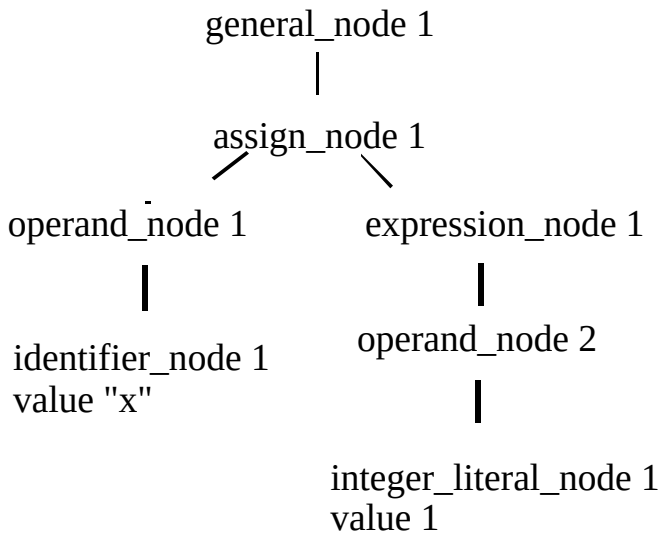


Figure 8.

STATEMENT RECOGNITION

This section will describe the recognition process of a simple variable increment as displayed in Figure 9 as an example of the recognition process. The first rule firing will recognize a *special expression*, an identifier plus the integer constant one. The second rule firing will recognize a variable being assigned a *special expression* and a third rule will recognize that the identifier on the right hand side of the statement and the variable on the left hand side are the same, thus signifying an incrementing assignment statement.

```
(assign x (plus x 1))
```

Figure 9. Variable Increment

These three rule firings process one statement and further rule conditional elements will attempt to group this statement with related statements to recognize multi-statement constructs. An example would have this *assign* statement within a *loop* statement, and both preceded by an *x = 0* initialization statement. This would indicate that the variable *x* is an index of the *loop* statement.

SUMMARY AND FUTURE RESEARCH

This research has lead to the development of a template hierarchy for *program understanding* and a general procedure for developing rules to recognize program *plans*. This method has been performed by the researcher, but will be automated in future versions of this system.

There are currently three algorithm *plans* recognized including the selection sort (SSA), quicksort (QSA) and heapsort (HSA). The SSA requires 50 rule firings, the QSA requires 90 firings, and the HSA, the longest of the algorithms at approximately 50 lines of code and taking over 60 seconds on a Intel 486-class machine requires 150 firings. The complete HLAR recognition system contains 31 templates, 4 functions, and 135 rules.

The research team intends to concentrate on algorithms common to the numerical community. The first version of the HLAR system has been successful at recognizing small algorithms (less than 50 lines of code). Expansion of HLAR into a robust tool requires: rehosting the system into a networking environment for distributing the recognition task among multiple CPUs, automating the generation of recognition rules for improved utility, attaching a database for consistent information and system-wide (multiple program) information, and a graphical user-interface.

AUTHOR INFORMATION

The author has: a B.S. in Computer Science, Wright State University, 1985; an M. S. in Computer Science, Wright State University, 1990; and a Ph.D. in Computer Science, New Mexico Institute of Mining and Technology, 1994. He is currently an Assistant Professor of Computer Science at Southeastern Oklahoma State University.

REFERENCES

1. Finkbine, R. B. “High-Level Algorithm Recognition,” Ph.D. Dissertation, New Mexico Institute of Mining and Technology, 1994.

2. Harandi, M. T. and Ning, J. Q.“Pat: A Knowledge-based Program Analysis Tool,” In 1988 IEEE Conference of Software Maintenance, IEEE CSP, 1988.

GRAPHICAL SEMANTIC DATA EXTRACTION: A CLIPS APPLICATION

G. Giuffrida, C.Pappalardo and G. Stella
University of California
CSATI - Catania
Los Angeles CA 90024

This paper presents a CLIPS application to perform semantic data extraction from maps. The given input is a very large set of maps, each map reproduces a reduced area of a phone network. Several basic graphical elements (nodes, connecting links, descriptive text) have been used for drawing such maps. Standards have been defined over the graphical elements (shape, size, pen thickness, etc.) in order to make a more uniform graphical representation among the set of maps. At drawing time no semantic meta-data have been associated to the maps; all the objects have been drawn only for the sake of graphical representation and printing.

The new generation of powerful GIS systems have so created the stimulous and the need of a richer data description. Metadata are required in order to get out the most from such graphical representations. Sophisticated graphical query/asnwering mechanism will be made available to the user once the enriched set of data will migrate to a GIS system. Purpose of this application is to create the required metaknowledge starting from the flat graphical representation of the input maps. Examples of metadata of interest are: association between links and text describing technical characteristics of such links; connectivity graph, classification of nodes.

The discussed application is strongly based on graphical reasoning; CLIPS set of built-in functions was extended in order to implement computational geometry algorithms. A graphical version of CLIPS, namely G-CLIPS, has been created to properly support implementation of our application. Interesting general purpose graphical features are then available in G-CLIPS.

USING EXPERT SYSTEMS TO ANALYZE ATE DATA

Jim Harrington
Honeywell Military Avionics
MS 673-4
13350 US Highway 19 N.
Clearwater FL, 34624
harrington@space.honeywell.com

ABSTRACT

The proliferation of Automatic Test Equipment (ATE) is resulting in the generation of large amounts of component data. Some of this component data is not be accurate due to the presence of noise. Analyzing this data requires the use of new techniques. This paper describes the process of developing an expert system to analyze ATE data and provides an example rule in the CLIPS language for analyzing trip thresholds for high gain/high speed comparators.

INTRODUCTION

We are seeing a proliferation of “Simple” Automatic Test Equipment (ATE) based on personal computers. Large quantities of test data is being generated by these test stations. Some of this data will not accurately represent the true characteristics of the equipment being tested, particularly when the power supply in the personal computer is being used to power the ATE circuitry. This paper discusses a methodology for developing an expert system to examine the data files generated by the ATE. This expert system can be used to produce a data file containing the “most probable” data values with the effect of power supply noise removed. These “most probable” data values are based on specific statistical processes and special heuristics selected by the rule base.

THE NEED FOR A NEW APPROACH OF DATA ANALYSIS

Power supply noise can become a significant source of error during testing of high speed/high accuracy Analog to Digital (A/D) and Digital to Analog (D/A) converters (10-bits or greater) or high speed/high gain comparators. This power supply noise can cause:

- erratic data values from an A/D converter,
- wandering analog outputs (which can translate to harmonic distortion) from a D/A converter, and
- false triggers from a comparator.

A 10-bit A/D converter, optimized to measure voltages in the 0 to 5 V range, has a voltage resolution of 4.88 mV (i.e., the least significant bit--LSB--represents a voltage step of 4.88 mV). The 5 volt power bus on a personal computer (i.e., an IBM or third party PC) can have noise spikes in excess of 50 milivolts (mV). These noise spikes can, therefore, represent an error of greater than ten times the value of the LSB.

Even though the noise spikes on the PC power bus are considered high frequency (above a hundred Megahertz), high speed A/D converters and high speed comparators can capture enough energy from them to affect their operation.

The power supply noise is both conducted and radiated throughout the PC enclosure. Simple power line filters do not prevent noise from entering the ATE circuitry and affecting the tests being performed. Much of the noise is cyclic in nature, such as that resulting from Dynamic Random Access Memory (DRAM) refresh and I/O polling. Other noise can be tied to specific events such as disk access.

The best method of improving the accuracy of the ATE is proper hardware design. Using power line filters does provide some noise isolation. Metal enclosures can also reduce susceptibility to radiated emissions. Noise canceling designs will also help. Self powered, free standing ATE which is optically coupled to the PC is probably the cleanest solution (from a power supply noise perspective). However, there are cases when an existing design must be used, or when cost and/or portability factors dictate that the ATE be housed within the PC. In these situations, an expert system data analysts can be used to enhance the accuracy of the test data.

The Traditional Software Approach

The traditional software approach to remove noise from a data sample is to take several readings and average the values. This approach may be acceptable if the system accuracy requirement is not overly stringent. A procedural language such as FORTRAN would be a good selection for implementing a system that just calculates averages. If A/D or D/A linearity is being tested, the averaging approach may not be accurate enough. The problem with this particular approach is that all of the data values that have been "obviously" affected by noise (as defined by "an expert") are pulling the average away from the true value.

The Expert System Approach

An expert system could be devised to cull out the "noisy" data values before the average is taken, resulting in a more accurate average. The problem is to develop an expert system that can be used to identify when noise might be effecting a data value.

THE PROCESS

The first step in developing an expert system to remove noisy data from the calculations is to define how the noise affects the circuits. For example, a voltage ramp is frequently used to test the linearity of an A/D converter and the trip points of a comparator. How does noise on that voltage ramp affect the performance of the circuitry being tested?
As a voltage source ramps down from 5 volts to Ø volts, noise can cause high speed A/D converters to output a data set with missing and/or repeated data values. Figure 1 shows the possible effect of random noise on the output of an A/D converter; the data line in this graph represents the source voltage that has been corrupted by noise and then quantized to represent the output of an A/D converter. Even though the general trend may be linear, the actual data is not.

Noise on a Ø to 5 volt ramp can cause a high speed comparator to change states too early or too late. Another common effect of noise on a comparator is a "bouncy" output (turning on and off in quick succession before settling to either the "on" or "off" state). Figure 2 shows the effect of noise on the output of the comparator.

The next step is to identify how the noise appears on the output of the equipment being tested. As seen in Figure 1, noise can cause runs of numbers with the same output from the A/D converter (e.g., three values in succession that the A/D converter interprets as 4002 mV). We also see missing data codes (as in the run of 3919, 3919, 3933, 3933--repeating the data values 3919 and 3933 but missing the data values 3924 and 3929 mV). Figure 2 shows an unexpected "on" and "off" cycle from the comparator before the voltage ramp reached the true trip point of 3970 mV.

An additional cycle to the off state occurs at the end of the grapg as seen in the comparator output going to 0 V, it should have stayed in the high (+5 V) state.

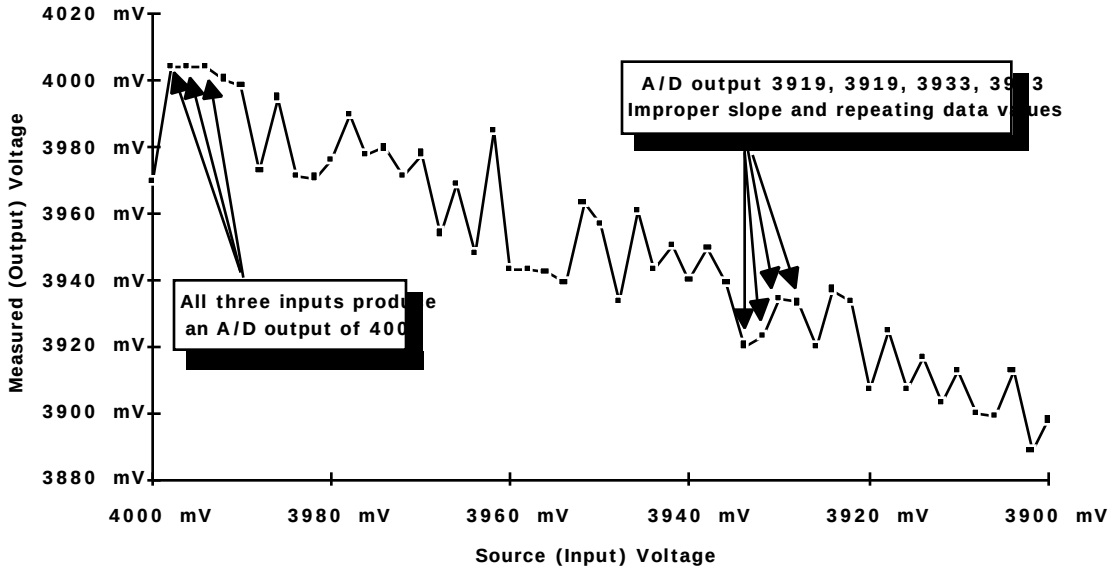


Figure 1. A/D Output Errors Caused by a Noisy Voltage Source

For brevity, the rest of this paper focuses on the comparator example; the approaches discussed for the comparator are directly applicable to the A/D converter as well.

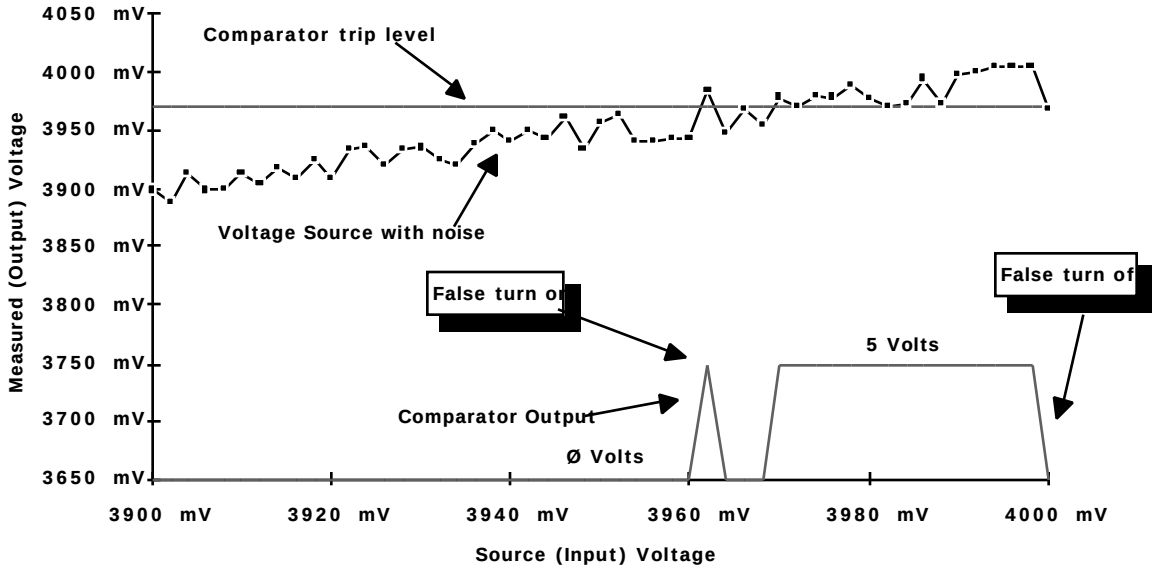


Figure 2. Comparator Output With a Noisy Source Signal

In most test set-ups, there will be no synchronization between any cyclic noise in the PC and the data acquisition process because most computer systems fill a buffer before dumping the data to the disk drive (this is ture unless the code generated for data acquisition goes through the specific process of “read data” “forced write to the disk” “read data” “forced write to the disk” etc.). This means that the cyclic noise will affect the data at random times during the data taking process. The non-cyclic random noise will, of course, appear at random times. Therefore, if we repeat the test multiple times, the noise should manifest itself at different places in the test sequence each time.

The noise shown in Figures 1 and 2 is random, with peaks as high a 50 mV. These plots were constructed to demonstrate the effect of noise on the circuitry. Under normal conditions, with the test voltage generated in a PC, the actual source voltage would look like the data shown in Figure 3. The “Source Voltage ramp” in Figure 3 shows noise spikes as high as 65 mV at regular intervals. The area between the noise spikes has some low—level asynchronous noise. The results of the combination of low level random noise and high level cyclic noise on a comparator is shown at the bottom of the graph (the voltage trip level is still set at 3970 mV).

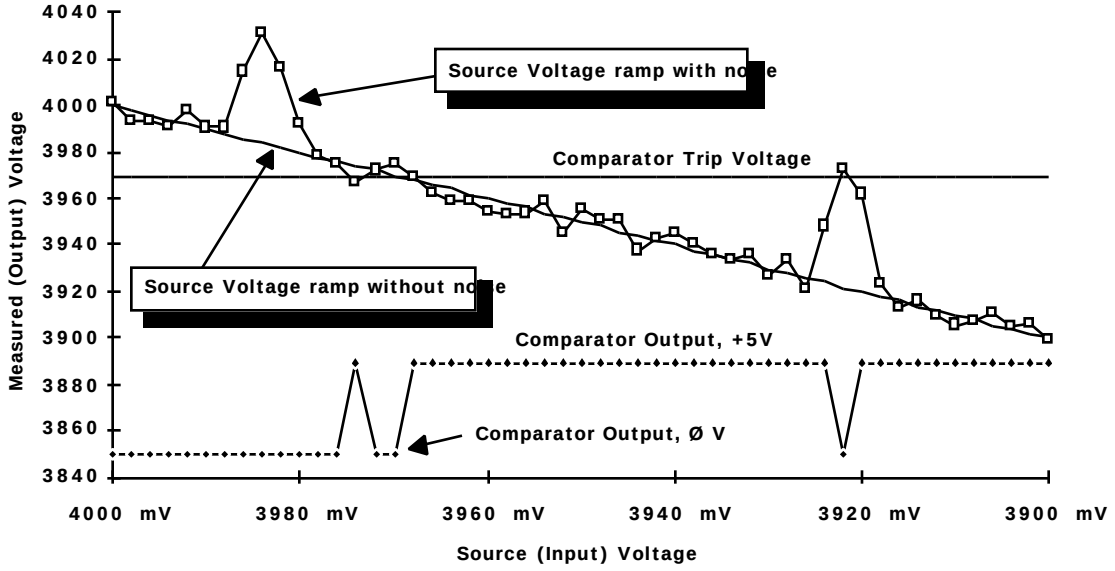


Figure 3. Noise Waveform During Test

Table 1 lists the voltages at which the comparator output shifts between Ø V and 5 V. The table also shows the voltages of the initial “bounce” on and off (“Off to On #1” and “On to Off #1”), and the inadvertent bounce off and back on (“On to Off #2” and “Off to On #3”). A basic set of rules starts to emerge when this noise spectrum is combined with knowledge of the effect of noise on the data taking process.

DEVELOPING THE RULE SET

Three sets of data were accumulated for each threshold analysis. These data sets were separated in time by only seconds as the ATE reset itself; this meant that the basic environmental factors were fairly consistent (an important factor when using any statistical process on data). The following rules are shown in order of precedence, from most important to least important.

Salience levels were used to ensure the most important rule would fire before a rule of lesser importance.

Once the optimum data value is selected by a rule, the data is written to an output file. The old data is then removed from the fact list to prevent a rule of lower precedence from firing and changing the data value again.

Comparator ID	Off to On # 1	On to Off # 1	Off to On # 2	On to Off # 2	Off to On # 3
001	3974	3972	3968	3922	3920
•					
•					
•					

Table 1. Comparator Input Threshold Voltage Test

Rule 1

The first rule developed was one of common sense. If the comparator does not show any bounces in any of the three data sets, and the data values are identical, that value should be used.

Rule 2

The second rule is similar. If two of the three data sets contain no bounces and the comparator output changes from Ø V to 5 V at the same input ramp voltage, the data value found in those two data sets is used. This rule takes effect (is instanciated) regardless of the number of bounces in the third data set (i.e., the third data set could contain a single trip level or 3 trip levels--as in Figure 3--or 100 trip levels). In mathematical terms this is called selecting the mode of the data (i.e., the most commonly repeated value in a data set [1]).

The second rule handles the possible situation of the noise spike occurring just prior to the time the true source voltage exceeds the trip level. The noise spike would cause the comparator to trip early. The philosophy behind the rule is that the high level noise spikes seen in Figure 3 can cause the comparator to switch early in one data set, but the probability of the noise spike affecting two data sets the same way is extremely low.

This rule will also fire if none of the three data sets have bounces, and the values are the same. If both rule 1 and rule 2 fire, the trip level will be recorded in the output file twice. Repeated data in the output file can cause other data analysis programs to have problems. This is the reason the old data (the data used to infer the optimum trip level) is removed by each rule that fires; so that the rule with the highest precedence will remove the trip level data, preventing a secondary rule from also firing.

Rule 3

The next rule handles the case where no bounces are detected, but the trip levels in the three data sets are all different. In this situation, it is difficult to determine which data set contains good data and which might contains noisy data. We decided to average the two closest together values. This averaging represents the effect of finding a trip level that is between two steps in the voltage ramp (e.g., the addition of low-level noise causing the comparator to trip at one level one time and another level the second time). If all three data values are equally dispersed, the average of

all three values is taken. The mathematics used to implement this rule in effect takes the median of the data when all three data sets were used (an easy way of selecting the average of three equally dispersed numbers).

In hindsight, another method of approaching the same rule is to track the interval of the 65 mV noise spike. By tracking the last occurrence of comparator output bounces, the period of the large cyclic noise spike can be determined (this approach requires an additional pass through the data to determine the period of the cyclic noise). The period of the cyclic noise can be used as an additional constraint, and the data set(s) that are closest to the middle of their periods can be averaged.

At this point, we have covered all of the easy rules. The following rules rely more on heuristics developed by the project research team (i.e., the domain experts).

Rule 4

The fourth rule covers the case when two of the data sets contain no bounces, but the data values are different, and the third file contains bouncy data. In this situation, we decided to average the data from the two non-bouncy data sets. An additional precaution was added to this rule in that if either of the two values is more than ±5% from the average value, the operator is notified of the comparator being tested and the two trip voltage levels.

Rule 5

If only one of the data sets doesn't contain bouncy data, the data value of that one data set is used. This rule covers the case when "the experts" feel that two of the data sets are affected by some high level noise spikes. It is "felt" that the one file non-bouncy data set contains the most accurate trip level on the most frequent basis.

Rule 6

The final case is when all three data sets contain bounces. This means that all three data sets have been affected by noise. In Figure 3, the noise spikes are only one reading increment wide. Both the on-off cycle and the off-on cycle are in the inverted state for one data reading. The difference between the first two data values in Table 1 is 2 mV; this is also true for the last two data values. The difference between the second and third data values is 4 mV, and the difference between the third and fourth data values is 46 mV. The comparator settles to the "On" state at 3968 mV. The rule that forms out of this analysis is to select for the trip level (when the comparator turns "On") the voltage that has the greatest difference between it and the adjacent "Off" voltage (e.g., in Figure 3, the trip level would be selected as 3968 mV, the point which starts the 46 mV span before the next "Off" spike).

The average of all 5 data values is 3951.20 mV; an error of 16.8 mV compared to the 3968 mV trip level. The average of the 3 values which cause an "Off to On" transition of the comparator is 3954 mV; an error of 14 mV. The increase in accuracy is easily seen in this example. But, we are not done yet. What if all of the transitions of the comparator occur at equal voltage intervals?

There is no clear cut heuristic when all of the "Off to On" and "On to Off" data values are evenly spaced. The system we developed defaulted to the first "Off to On" transition. Depending on the system implementation, the middle "Off to On" transition may be the best selection.
The coding of this rule is a little tricky. The code in Figure 4 shows how this rule could be implemented in CLIPS.

```
(defrule Compromise
   "This rule finds the most probable data value out of a set of bounces"
   (Evaluate_Data)  ; flag fact that allows the compromise rule to function
   ?Orig_Data <- (?set&set_1|set_2|set3 $?data)   ;get the data
   (test (>= (length $?data) 2))         ;test for bounces
   =>
   (retract ?Orig_Data)  ;Get rid of the data set that is to be replaced
   (bind ?elements (length $?data))
   (if (evenp ?elements)
      then
      (assert (?set 0))  ;Bit ended in original state--no data
      else
      (bind ?delta 0)  ;Set up the variable to find the largest difference
      (bind ?y 2)  ;Pointer to the first even number (position) data value
      (while (<= ?y ?elements)
         (bind ?x (- ?y 1))  ;Pointer to the position preceding ?y
         (bind ?z (+ ?y 1))  ;Pointer to the position following ?y
         (bind ?test_val (abs (- (nth ?x $?data) (nth ?y $?data))))
         (if (> ?test_val ?delta)
            then
            (bind ?pointer ?x)
            (bind ?delta ?test_val))
         (bind ?test_val (abs (- (nth ?y $?data) (nth ?z $?data))))
         (if (> ?test_val ?delta)
            then
            (bind ?pointer ?z)
            (bind ?delta ?test_val))
         (bind ?y (+ ?y 2)))  ;Increment ?y to the next even position
      ; put the new data on the fact list
      (assert (?set =(nth ?pointer $?data)))))
```

Figure 4. CLIPS Code Implementing Rule Number 6

The actual searching of the data set is performed in the “while” statement found on the right hand side of the rule (the “then” part of the rule, found after the “=>” symbols).

Rule 6 is be performed on all three data sets separately. Remember that this rule is invoked only when bounces are seen in all three data sets. This point helps to alleviate some of the worry surrounding the case where all of the trip levels are equally dispersed. Once all three data sets are evaluated and a single voltage level is selected, the three new values are put on the fact list to be operated on by the first 5 rules. Only rules number 1 through 3 actually apply since none of the data generated by rule 6 would contain bounces.

CONCLUSIONS

This process demonstrates a new approach to analyzing data taken by ATE. Increases in accuracy of 14 to 17 mV is demonstrated in rule 6. The other rules select the heuristic or statistical procedure used to determine the best data value from the multiple data sets provided. This paper illustrates the method of developing a set of rules, implementable in CLIPS, for defining the presence of noise in a data set and for removing the noisy data from the calculations.

ACKNOWLEDGMENTS

I would like to thank Richard Wiker and Roy Walker for their diligent work in helping to define the heuristics involved in the data analysis.

REFERENCES

1. Spiegel, Murray R., “Chapter 3: The Mean, Median, Mode and Other Measures of Central Tendency,” *Schaum’s Outline Series: Theory and Problems of Statistics*, McGraw-Hill Book Company, New York, 1961, pp. 47 - 48.

REAL-TIME REMOTE SCIENTIFIC MODEL VALIDATION

Richard Frainier and Nicolas Groleau
Recom Technologies, Inc.

NASA Ames Research Center, m/s 269-2
Moffett Field, CA 94035-1000, USA

ABSTRACT

This paper describes flight results from the use of a CLIPS-based validation facility to compare analyzed data from a space life sciences (SLS) experiment to an investigator's pre-flight model. The comparison, performed in real-time, either confirms or refutes the model and its predictions. This result then becomes the basis for continuing or modifying the investigator's experiment protocol. Typically, neither the astronaut crew in Spacelab nor the ground-based investigator team are able to react to their experiment data in real time. This facility, part of a larger science advisor system called Principal-Investigator-in-a-Box, was flown on the Space Shuttle in October, 1993. The software system aided the conduct of a human vestibular physiology experiment and was able to outperform humans in the tasks of data integrity assurance, data analysis, and scientific model validation. Of twelve pre-flight hypotheses associated with investigator's model, seven were confirmed and five were rejected or compromised.

INTRODUCTION

This paper examines results from using a CLIPS-based scientific model validation facility to confirm, or refute, a set of hypotheses associated with a Shuttle-based life-science experiment. The model validation facility was part of a larger software system called "PI-in-a-Box" (Frainier et al., 1993) that was used by astronauts during the October, 1993 Spacelab Life Sciences 2 (SLS-2) mission. The model validation facility (called Interesting Data Filter in the PI-in-a-Box system) compares the output of the scientific data analysis routines with the investigator's pre-flight expectations in real-time.

The model validation facility compares analyzed data from the experiment with the investigator's model to determine its fit with pre-flight hypotheses and predictions. The fit can be either statistical or heuristic. Deviations are reported as "interesting". These deviations are defined as "needing confirmation", even if not part of the original fixed protocol. If confirmed, then at least a portion of the theoretic model requires revision. Further experiments are needed to pinpoint the deviation. This idea is at the heart of the iterative process of "theory suggesting experiment suggesting theory".

THE ROTATING DOME EXPERIMENT

The PI-in-a-Box software system was associated with a flight investigation called the Rotating Dome Experiment (RDE). This was an investigation into the effects of human adaptation to the micro-gravity condition that exists in Earth-orbiting spacecraft. A sensation, called "angular vection", was induced in a set of human subjects by having them view a rotating field of small, brightly-colored dots. After a few seconds, the subject perceives that s/he is rotating instead of the constellation of dots. This perception of self-rotation generally persists throughout the time that the dots are rotating, though occasionally the subject realizes that it is in fact the dots that are rotating. This sudden cessation of the sensation of vection is termed a "dropout". With the RDE, the field of dots rotates in a set direction (clockwise/counter-clockwise) with a set speed for 20 seconds. There is a 10-second pause, and then the rotation resumes (though with a new direction and/or angular speed). There are six such 20-second trials for each experiment run.

There are three experiment conditions for RDE subjects. In the first, called “free-float”, the subject grips a biteboard with his/her teeth in front of the rotating dome (and is otherwise floating freely). In the second, called “tether”, the subject is loosely tethered to the front of the rotating dome without the biteboard. In the third, called “bungee”, the subject is attached to the “floor” of the laboratory by a set of bungee cords, again teeth gripping a biteboard.

There are eight main parameters measured with respect to angular vection during the RDE. Four of these parameters are “subjective”, meaning that the subject consciously reports them by manipulating a joystick/potentiometer: These are the time interval from the start of dome rotation to the onset of the sensation of vection (measured in seconds), the average rate of perceived vection (expressed as a percent of the maximum), the maximum rate of perceived vection (also expressed as a percent of the maximum), and the number of times during a 20-second trial that the sensation suddenly ceases (the dropout count, an integer). The remaining four parameters are “objective”, meaning that the subject’s involuntary movements are recorded. These are the first and second head movements associated with the torque strain gage mounted on the biteboard and the same head movements associated with subject neck muscle activity detectors (electromyograms). These eight parameters were measured for each 20-second trial of a run.

In the flight system, twelve distinct hypotheses were identified. These were all associated with the joystick-generated subjective parameters. They are:

1. There should be some sensation of angular vection.
2. The average time for the onset of the sensation of vection for the six trials of a run should be greater than 2 seconds*.
3. The average time for the onset of the sensation of vection for the six trials of a run should be less than 10 seconds.
4. Early in a mission, before adaptation to micro-gravity is significantly underway, the average of the six trials’ maximum sensation of vection should be less than 90%.
5. Late in a mission, after adaptation to micro-gravity is complete, the average of the six trials’ maximum sensation of vection should be more than 80%.
6. Tactile cues decrease the sensation of vection, therefore, the average of the six trials’ maximum sensation of vection for a free-float run should be more than that of a bungee run.
7. The average of the six trials’ average sensation of vection should be more than 30%.
8. The average of the six trials’ average sensation of vection should be less than 80%.
9. Tactile cues decrease the sensation of vection, therefore, the average of the six trials’ average sensation of vection for a free-float run should be more than that of a bungee run.
10. There should be at least one dropout during a bungee run.
11. There should not be an average of more than two dropouts per trial during a free-float run.
12. The average of the six trials’ dropout count for a free-float run should be less than that of a bungee run.

MODEL VALIDATION

Real-time, quick-look data acquisition and analysis routines extract significant parameters from the experiment that are used by the model validation facility (see Figure 1). With the RDE, predictions were formed on the basis of data from two sources. The first source was previously-collected flight data. (The RDE was flown on three earlier missions: SL-1, D-1, and SLS-1.) The

*Strictly speaking, the numeric value of this hypothesis (and most other hypotheses) was subject to adjustment on a subject-by-subject basis as a result of pre-flight "baseline data" measurements. This was due to the significant variability between individual human subjects.

second source was from SLS-2 crew responses recorded on earth before the flight during baseline data collection sessions. These predictions were used to define thresholds that, if violated, indicated significant deviations from the investigator's model. Many space life-sciences investigations (including the RDE) are exploratory in nature, and the investigator team expected significant deviations for perhaps 20% of the experiment runs. When detected, these deviations were made available for display to the astronaut-operator. It is then that the reactive scientist briefly reflects on the situation and try to exploit the information to increase the overall science return of the experiment. This would most likely result in a change to the experiment protocol.

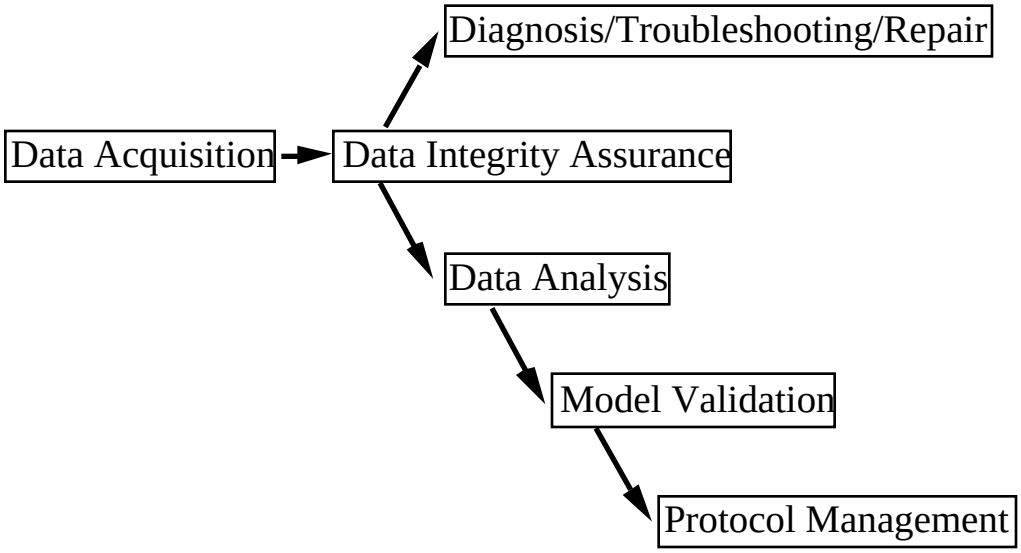


Figure 1. Flow of control.

For the PI-in-a-Box system, the model validation facility was named the Interesting Data Filter (IDF). The IDF was a set of CLIPS rules and facts that compared current experiment results with the investigator's preflight expectations. There were approximately two dozen rules† and 40 facts that comprised the pre-flight hypotheses.

FLIGHT RESULTS

Results of the flight use with respect to the 12 hypotheses are listed in Table 1:

- the first column identifies the hypothesis number from the list of hypotheses presented earlier.
- the second column is the binomial probability of observing the given outcome assuming 95% of the run results agree with the model.
- the third column is our conclusion with respect to the hypothesis given the overall SLS-2 flight evidence. The hypothesis is rejected when the probability of observing the flight results given the hypothesis is < 0.001; it is compromised when the probability of observing the flight results given the hypothesis is < 0.01; it is suspect‡ when the probability of observing the flight results given the hypothesis is < 0.05; and it is accepted otherwise.

†See Appendix for listing of CLIPS rules.
‡This case does not occur for this data set.

- the fourth column summarizes the mission results. This is expressed as a ratio where the denominator represents the number of experiment runs producing data that bears on the hypothesis and the numerator represents the subset of those experiment runs whose data supports the hypothesis. The entry "n/a" denotes that the hypothesis was not applicable to that flight day.

- the last three columns present a more detailed view of the results from each of the three flight days (fd) when the system was in use.

These results indicate that seven of 12 pre-flight hypotheses were accepted. Five hypotheses were either rejected or compromised, indicating a need to modify the existing model with respect to the pattern of human adaptation to weightlessness over time, with respect to the importance of dropouts as an indication of adaptation, and with respect to the influence of tactile cues.

hyp #	probability	result	mission total	FD2	FD8	FD11
1	1.0000	accepted	38/38	12/12	14/14	12/12
2	0.2642	accepted	18/20	6/6	8/8	4/6
3	1.0000	accepted	38/38	12/12	14/14	12/12
4	1.0000	accepted	12/12	12/12	n/a	n/a
5	0.0000	rejected	0/12	n/a	n/a	0/12
6	0.0040	compromised	17/22	3/4	9/10	5/8
7	0.8576	accepted	37/38	11/12	14/14	12/12
8	1.0000	accepted	38/38	12/12	14/14	12/12
9	0.0040	compromised	17/22	4/4	8/10	5/8
10	0.0000	rejected	5/13	1/4	3/5	1/4
11	1.0000	accepted	13/13	4/4	5/5	4/4
12	0.0009	rejected	14/22	4/4	5/10	5/8

Table 1. Flight data confirmation of vection-related hypotheses.

CONCLUSION

A scientific model validation facility has been devised for space science advisor systems that appears to be a useful framework for confirming or refuting pre-flight hypotheses. This facility is a key step to achieving truly reactive space-based laboratory science.

ACKNOWLEDGMENTS

The authors would like to thank other members of the PI-in-a-Box team, past and present, for their efforts, especially Lyman Hazelton, Silvano Colombano, Michael Compton, Irv Statler, Peter Szolovits, Larry Young, Jeff Shapiro, Guido Haymann-Haber, and Chih-Chao Lam. Thanks as well to NASA, and especially Peter Friedland, for management support. This work was funded by the NASA Office of Advanced Concepts and Technology AI program.

REFERENCES

1. Frainier Richard, Groleau Nicolas, Hazelton Lyman, Colombano Silvano, Compton Michael, Statler Irv, Szolovits Peter, and Young Larry, "PI-in-a-Box: a knowledge-based system for space science experimentation", *AI Magazine*, Menlo Park, CA, vol. 15, no. 1, Spring, 1994, pp. 39-51.

APPENDIX: MODEL VALIDATION FLIGHT RULE-SET

```
;; ;these rules follow CLIPS v4.3 syntax

;;start up IDF
(defrule idf-startup
   (idf)
   (disk-drive ?disk)
   (interface-directory ?dir)
   =>
   (bind ?predictions-file (str_cat ?disk ?dir “BDC-predictions”))
   (bind ?input-file (str_cat ?disk ?dir “idf-input”))
   (load-facts ?predictions-file)
   (load-facts ?input-file)
   (assert (idf-result not-interesting)) ; the default result
   (open (str-cat ?disk ?dir “idf-stats”)
         idf-stats “w”)) ; no append w/o file size limit checking!


;; formula: SD_square = (1/n)(sum_of squares - (square of sum)/n)
(defrule compute_statistics_for_six_trials
   (declare (salience 10))
   (?parameter trial_data ?t1 ?t2 ?t3 ?t4 ?t5 ?t6)
   =>
   (bind ?sum (+ (+ (+ (+ (+ ?t1 ?t2) ?t3) ?t4) ?t5) ?t6))
   (bind ?sum_of_squares (+ (** ?t1 2) (** ?t2 2) (** ?t3 2)
                            (** ?t4 2) (** ?t5 2) (** ?t6 2)))
   (bind ?mean (/ ?sum 6))
   (bind ?SD_square (/ (- ?sum_of_squares (/ (** ?sum 2) 6)) 6))
   (bind ?SD (sqrt ?SD_square))
   (assert (?parameter sum ?sum))
   (assert (?parameter sum of squares ?sum_of_squares))
   (assert (?parameter experiment_result ?mean))
   (assert (?parameter standard deviation ?SD)))


;;; Parameter-specific rules to detect interestingness


;; ONSET_OF_VECTION

;; Onset of vection is interesting if it’s non-existent
;; (that is, less than 0.03 seconds)
(defrule no-vection-detected
   (declare (salience 5))
   (Onset_Of_Vection experiment_result ?x&:(< ?x 0.03))
   =>
   (assert (no-vection-detected)))

(defrule no-vection-detected--interesting
   (declare (salience 5))
   (no-vection-detected)
   (Maximum_Vection_Intensity experiment_result ?x&:(< ?x 10))
   =>
   (assert (Onset_Of_Vection conclusion potentially_interesting
            “No vection was detected.”)))

;; Onset of vection is interesting if it’s consistently < threshold
```

```
;; (but >= 0.03)
(defrule onset_of_vection_less_than_2
   (Onset_Of_Vection experiment_result ?mean_found)
   (subject ?subj)
   (BDC-datum ?subj quick_onset ?threshold)
   (test (and (> ?mean_found 0.03) (< ?mean_found ?threshold)))
   =>
   (bind ?msg (str_cat "Mean onset of vection is less than "
                       ?threshold " seconds"))
   (assert (Onset_Of_Vection conclusion potentially_interesting ?msg)))


;; Onset of vection in flight is interesting if it's
;; consistently > threshold
(defrule onset_of_vection_greater_than_10
   (environment flight)
   (subject ?subj)
   (BDC-datum ?subj slow_onset ?threshold)
   (Onset_Of_Vection experiment_result
                    ?mean_found&:(> ?mean_found ?threshold))
   =>
   (bind ?msg (str_cat "Mean onset of vection is greater than "
                       ?threshold " seconds"))
   (assert (Onset_Of_Vection conclusion potentially_interesting ?msg)))


;; MAXIMUM_VECTION_INTENSITY

(defrule early_interesting_maximum_vection
; Early in flight (Day 0 / Day 1), maximum vection is
; interesting if it's consistently > threshold.
   (environment flight)
   (day 0|1)
   (subject ?subj)
   (BDC-datum ?subj early_hi_max ?threshold)
   (Maximum_Vection_Intensity experiment_result
         ?mean_found&:(> ?mean_found ?threshold))
   (not (no-vection-detected))
   =>
   (bind ?msg (str_cat "Max vection intensity mean is greater than "
                       ?threshold "%"))
   (assert (Maximum_Vection_Intensity conclusion
                                potentially_interesting ?msg)))


(defrule late_interesting_maximum_vection
; Late in the flight, maximum vection is interesting
; if it's consistently < threshold
   (environment flight)
   (day ?day&:(> ?day 7))     ; "Late" is Day 8 or later
   (subject ?subj)
   (BDC-datum ?subj late_lo_max ?threshold)
   (Maximum_Vection_Intensity experiment_result
           ?mean_found&:(< ?mean_found ?threshold))
   (not (no-vection-detected))
   =>
   (bind ?msg (str_cat "Max vection intensity mean is less than "
         ?threshold "%"))
```

```
   (assert (Maximum_Vection_Intensity conclusion
                                potentially_interesting ?msg)))

;; Max vection is interesting if tactile > free-float
(defrule free-below-bungee--interesting--maximum-vection
   (body_position free-flt)
   (Maximum_Vection_Intensity experiment_result ?ff)
   (Maximum_Vection_Intensity running_mean
         ?val&:(> ?val ?ff)) ; bungee mean
   =>
   (assert (Maximum_Vection_Intensity conclusion potentially_interesting
      "Subj's max. vection < bungee cond. max. vection")))

(defrule bungee-above-free--interesting--maximum-vection
   (body_position bungee)
   (Maximum_Vection_Intensity experiment_result ?b)
   (Maximum_Vection_Intensity running_mean
            ?val&:(> ?b ?val)) ; free-float mean
   =>
   (assert (Maximum_Vection_Intensity conclusion potentially_interesting
      "Subj's max. vection > free-float cond. max. vection")))


;; AVERAGE_VECTION_INTENSITY

(defrule low_average_vection_intensity
; Average vection intensity is interesting if it's consistently < threshold
   (environment flight)
   (subject ?subj)
   (BDC-datum ?subj lo_average ?threshold)
   (Average_Vection_Intensity experiment_result
         ?mean_found&:(< ?mean_found ?threshold))
   (not (no-vection-detected))
   =>
   (bind ?msg (str_cat "Avg. vection intensity mean is less than "
                     ?threshold "%"))
   (assert (Average_Vection_Intensity conclusion
                   potentially_interesting ?msg)))

(defrule high_average_vection_intensity
; Average vection intensity is interesting if it's consistently > threshold
   (environment flight)
   (subject ?subj)
   (BDC-datum ?subj hi_average ?threshold)
   (Average_Vection_Intensity experiment_result
            ?mean_found&:(> ?mean_found ?threshold))
   (not (no-vection-detected))
=>
   (bind ?msg (str_cat "Avg. vection intensity mean is greater than "
                     ?threshold "%"))
   (assert (Average_Vection_Intensity conclusion potentially_interesting
                                  ?msg)))

;; Average vection is interesting if tactile > free-float
(defrule free-below-bungee--interesting--average-vection
   (body_position free-flt)
   (Average_Vection_Intensity experiment_result ?ff)
   (Average_Vection_Intensity running_mean
```

```
         ?val&:(> ?val ?ff)) ; bungee mean
   =>
   (assert (Average_Vection_Intensity conclusion potentially_interesting
      "Subj's ave. vection < bungee cond. ave. vection")))

(defrule bungee-above-free--interesting--average-vection
   (body_position bungee)
   (Average_Vection_Intensity experiment_result ?b)
   (Average_Vection_Intensity running_mean
          ?val&:(> ?b ?val)) ; free-float mean
   =>
   (assert (Average_Vection_Intensity conclusion potentially_interesting
      "Subj's ave. vection > free-float cond. ave. vection")))


;; DROPOUTS

;; Number of dropouts is interesting if it's
;; consistently 0 under tactile conditions
(defrule interesting_dropouts_tactile
   (environment flight)
   (body_position bungee)
   (Dropouts experiment_result 0)
   (not (no-vection-detected))
   =>
   (assert (Dropouts conclusion potentially_interesting
      "There were no dropouts with bungees attached")))

;; Number of dropouts is interesting if it's
;; consistently >2 under free-float conditions
(defrule interesting_dropouts_free
   (environment flight)
   (body_position free-flt)
   (Dropouts experiment_result ?mean_found&:(> ?mean_found 2))
   =>
   (assert (Dropouts conclusion potentially_interesting
      "Mean number of free-float dropouts is greater than 2")))

;; Number of dropouts is interesting if tactile consistently < free-float
(defrule free-above-bungee--interesting--dropouts
   (body_position free-flt)
   (Dropouts experiment_result ?ff)
   (Dropouts running_mean ?val&:(> ?ff ?val)) ; bungee mean
   =>
   (assert (Dropouts conclusion potentially_interesting
      "Subj's dropout count > bungee cond. dropout count")))

(defrule bungee-below-free--interesting--dropouts
   (body_position bungee)
   (Dropouts experiment_result ?b)
   (Dropouts running_mean ?val&:(< ?b ?val)) ; free-float mean
   =>
   (assert (Dropouts conclusion potentially_interesting
      "Subj's dropout count < free-float cond. dropout count")))


;;; OUTPUT STATS
```

```
(defrule output_idf_stats
   (?parameter experiment_result ?mean)
   (?parameter standard deviation ?SD)
   (body_position ?cond)
   (subject ?subj)
   =>
   (fprintout idf-stats “Subject: “ ?subj “ Cond: “ ?cond
              “ “ ?parameter “ mean: “ ?mean “ SD: “ ?SD crlf))

;; Output Interestingness info to “Session History”
;; file for Session Manager
(defrule record-interestingness--start
   (declare (salience -100))
   (?parameter conclusion ?interesting ?source)
   ?f <- (idf-result not-interesting)
   (disk-drive ?disk-drive)
   (interface-directory ?interface-dir)
=>
   (retract ?f)
   (assert (idf-result interesting))
   (open (str-cat ?disk-drive ?interface-dir “history-session”)
                  history-session “a”)
   (assert (record-interestingness)))

; potentially_interesting  => medium
; certainly_interesting => high
(defrule record-interestingness
   (record-interestingness)
   ?int <- (?parameter conclusion ?interesting ?source)
   (subject ?subj)
   (body_position ?cond)
   (current-step ?step)
   (this-session ?session)
=>
   (retract ?int)
   (if   (eq ?interesting potentially_interesting)
   then  (bind ?level medium)
   else  (bind ?level high))
   (fprintout history-session “(int-hist class interesting session “
          ?session “ step “ ?step “ subj “
       ?subj “ cond “ ?cond “ source “ ?source “ level “
          ?level “)” crlf))

(defrule record-interestingness--end
   (record-interestingness)
   (not (?parameter conclusion ?interesting ?source))
=>
   (close)
   (assert (ctrl--stop idf)) ; inhibit rules-control “abnormal” message
   (printout “hyperclips” “interesting”)) ; return to HyperCard

(defrule no-interesting-results
   (declare (salience -200))
   (idf-result not-interesting)
=>
   (close)
   (assert (ctrl--stop idf))
   (printout “hyperclips” “as-expected”)) ; return to HyperCard
```

ADDING INTELLIGENT SERVICES TO AN OBJECT ORIENTED SYSTEM

Bret R. Robideaux and Dr. Theodore A. Metzler

LB&M Associates, Inc.
211 SW A Ave.
Lawton, OK 73505-4051

(405) 355-1471

robidb@lbm.com metzlert@lbm.com

ABSTRACT

As today's software becomes increasingly complex, the need grows for intelligence of one sort or another to become part of the application- often an intelligence that does not readily fit the paradigm of one's software development.

There are many methods of developing software, but at this time, the most promising is the Object Oriented (OO) method. This method involves an analysis to abstract the problem into separate 'Objects' that are unique in the data that describe them and the behavior that they exhibit, and eventually to convert this analysis into computer code using a programming language that was designed (or retrofitted) for OO implementation.

This paper discusses the creation of three different applications that are analyzed, designed, and programmed using the Shlaer/Mellor method of OO development and C++ as the programming language. All three, however, require the use of an expert system to provide an intelligence that C++ (or any other 'traditional' language) is not directly suited to supply. The flexibility of CLIPS permitted us to make modifications to it that allow seamless integration with any of our applications that require an expert system.

We illustrate this integration with the following applications:

1. An After Action Review (AAR) station that assists a reviewer in watching asimulated tank battle and developing an AAR to critique the performance of the participants in the battle.

2. An embedded training system and over-the-shoulder coach for howitzer crewmen.

3. A system to identify various chemical compounds from their infrared absorption spectra.

Keywords - Object Oriented, CLIPS

INTRODUCTION

The goal of the project was to develop a company methodology of software development. The requirement was to take Object Oriented Analysis (done using the Shlaer-Mellor method) and efficiently convert it to C++ code. The result was a flexible system that supports reusability, portability and extensibility. The heart of this system is a software engine that provides the functionality the Shlaer/Mellor Method [2, 3] allows the analyst to assume is available. This includes message passing (inter-process communication), state machines and timers. CLIPS provides an excellent example of one facet of the engine. Its portability is well-known, and its extensibility allowed us to embed the appropriate portions of our engine into it. We could then

modify its behavior to make it pretend it was a natural object or domain of objects. This allowed us to add expert system services to any piece of software developed using our method (and on any platform to which we have ported the engine).

OBJECT ORIENTED METHODS

The principal concept in an OO Method is the object. Many methods include the concept of logically combining groups of objects. The logic behind determining which objects belong to which grouping is as varied as the names used to identify the concept. For example, Booch [4] uses categories and Rumbaugh [5] uses aggregates. Shlaer/Mellor uses domains [3]. Objects are grouped into Domains by their functional relationships; that is, their behavior accomplishes complementary tasks.

Figure 1 shows that the operating system is its own domain. It is possible the operating has very little to do with being OO, but it is useful to model it as part of the system. Another significant domain to note is the Software Architecture Domain. This is the software engine that provides the assumptions made in analysis. The arrows denote the dependence of the domain at the beginning of the arrow upon the domain at the end of the arrow. If the model is built properly, a change in any domain will only affect the domains immediately above it. Notice there are no arrows from the main application (Railroad Operation Domain) to the Operating System Domain. This means a change in operating systems (platforms) should not require any changes in the main application. However, changes will be required in the domains of Trend Recording, User Interface, Software Architecture and Network. The extent of those changes is dependent on the nature of the changes in the operating system. But this does stop the cascade effect of a minor change in the operating system from forcing a complete overhaul of the entire system.

While CLIPS could be considered an object, the definition of domains makes it seem more appropriate to call it a domain.

THE SOFTWARE ENGINE

The portion of the software engine that is pertinent to CLIPS is the message passing. The event structure that is passed is designed to model closely the Shlaer/Mellor event. We have implemented five (5) kinds of events:

[Figure Deleted]

Figure 1. Domain Chart for Automated Railroad Management System
Copied from page 141 (Figure 7.4.1) of [3]

1. Standard Events: events passed to an instance of an object to transition the state model

2. Class Events: events dealing with more than one instance of an object

3. Remote Procedure Calls (RPCs): An object may provide functions that other objects can call. One object makes an RPC to another passing some parameters in, having some calculations performed and receiving a response.

4. Accesses: An object may not have direct access to data belonging to another object. The data other objects need must be requested via an accessor.

5. Return Events: the event that carries the result of an RPC or Access event

The event most commonly received by CLIPS is the Class Event. CLIPS is not a true object. Therefore it has no true instances, and Standard Events are not sent to it. CLIPS is more likely to make an RPC than to provide one. It is possible that an object may need to know about a certain fact in the fact base, so receiving an Access is possible. Should CLIPS use an RPC or Access, it will receive a Return Event. RPCs and Accesses are considered priority events so the Return Event could be handled as part of the function that makes the call.

The events themselves have all of the attributes of a Shlaer/Mellor event, plus a few to facilitate functionality. The C++ prototype for the function Send_Mesg is:

```
void Send_Mesg (char* Destination,
                EventTypes Type,
                unsigned long EventNumber,
                char* Data,
                unsigned int DataLength = 0,
                void* InstanceID = 0,
                unsigned long Priority = 0);
```

The engine is currently written on only two different platforms: a Sun 4 and an Amiga. CLIPS, with its modifications, was ported to the Amiga --along with some applications software that was written on the Sun-- and required only a recompile to work. This level of portability means that only the engine needs to be rewritten to move any application from one platform to another. Once the engine is ported, any applications written using the engine (including those with CLIPS embedded in them) need only be recompiled on the new platform and the port is complete. Companies are already building and selling GUI builders that generate multi-platform code so the GUI doesn't need to be rewritten.

The Unix version of the Inter-Process Communication (IPC) part of the engine was written using NASA's Simple Sockets Library. Since all versions of the engine, regardless of the platform, must behave the same way, we have the ability to bring a section of the software down in the middle of a run, modify that section and bring it back up without the rest of the system missing a beat. In the worst case, the system will not crash unrecoverably.

MODIFICATIONS TO CLIPS

Since the software engine is written in C++, at least some part of the CLIPS source code must be converted to C++. The most obvious choice is main.c, but this means the UserFunctions code must be moved elsewhere since they must remain written in C. I moved them to sysdep.c.

The input loop has been moved to main.c because all input will come in over the IPC as valid CLIPS instructions. The stdin input in CommandLoop has been removed, and the function is no longer a loop. The base function added to CLIPS is:

```
(RETURN event_number  destination  data)
```

This function takes its three parameters and adds them to a linked list of structures (see Figure 2). When CommandLoop has completed its pass and returns control to main.c, it checks the linked list and performs the IPC requests the CLIPS code made (see Fig. 3). RETURN handles the easiest of the possible event types that can be sent out: Class Events. To handle Standard Events, some way of managing the instance handles of the destination(s) needs to be implemented. To handle Accesses and RPCs some method of handling the Return events needs to be implemented. The easiest way is to write a manager program. Use *RETURN* to send messages to the Manager. The event number and possibly part of the data can be used to instruct the Manager on what needs to be done.

```
RETURN ()
  {
    extract parameters from call

    allocate a new element for the list

    fill in the structure

    add structure to the list

  }
```

Figure 2. Pseudocode for function RETURN

```
main ()
  {
   init CLIPS

   begin loop

     Get Message

     set command string

     call CommandLoop

     check output list

     if there are elements in the structure
       make the appropriate send messages

  end loop
}
```

Figure 3. Pseudocode for main ()

Sending instructions and information into CLIPS requires a translator to take Shlaer/Mellor events and convert them to CLIPS valid instructions. In Shlaer/Mellor terms this construct is called a Bridge.

With these changes CLIPS can pretend it is a natural part of our system.

EXAMPLES

This system has been successfully used with three different projects. An After Action Review Station for reviewing a simulated tank battle, an Embedded Training System to aid howitzer crewmen in the performance of their job and a chemical classifier based on a chemical's infrared absorption spectra.

The After Action Review (AAR) station is called the Automated Training Analysis Feedback System or ATAFS. See Figure 4 for its domain chart. Its job is to watch a simulated tank battle on a network and aid the reviewer in developing an AAR in a timely manner. It does so by feeding information about the locations and activities of the vehicles and other pertinent data about the exercise to the expert system. The expert system watches the situation and marks certain events in the exercise as potentially important. ATAFS will then use this information to generate automatically a skeleton AAR and provide tools for the reviewer to customize the AAR to highlight the events that really were important.

[Figure Deleted]

Figure 4. ATAFS Domain Chart

In an early version of the system, ATAFS was given the location of a line-of-departure, a phase-line and an objective. During the simulated exercise of four vehicles moving to the objective, CLIPS was fed the current location of each vehicle. CLIPS updated each vehicle's position in its fact base and compared the new location with the previous location in relation to each of the control markings. When it found a vehicle on the opposite side of a control mark than it

previously was, it *RETURN*ed an appropriate event to ATAFS. Upon being notified that a potentially interesting event had just occurred, ATAFS is left to do what ever needs to be done. CLIPS continues to watch for other events it is programmed to deem interesting.

Embedded Training (ET) is a coaching and training system for howitzer crewmen. See Figure 5 for its domain chart. It provides everything from full courseware to simple reminders/checklists. It also provides a scenario facility that emulates real world situations and includes an expert system that watches the trainee attempt to perform the required tasks. During the scenario the expert system monitors the trainee's activities and compares them to the activities that the scenario requires.

[Figure Deleted]

Figure 5. ET Domain Chart

It adjusts the user's score according to his actions and uses his current score to predict whether the user will need help on a particular task. All of the user's actions on the terminal are passed along to CLIPS, which interprets them and determines what the he is currently doing and attempting to do. Having determined what the user is doing, CLIPS uses its *RETURN* in conjunction with a manager program to access data contained in various objects. This data is used by CLIPS to determine what level of help, if any, to issue the user. The decision is then *RETURN*ed to ET which supplies the specified help.

The chemical analysis system [6] designed by Dr. Metzler and chemist Dr. Gore takes the infrared (IR) spectrum of a chemical compound and attempts to identify it using a trained neural net. In the course of their study, they determined the neural net performed significantly better when some sort of pre-processor was able to narrow down the identification of the compound. CLIPS was one of the methods selected. The IR spectrum was broken down into 52 bins. The value of each of these bins was abstracted into values of strong, medium and weak. Initially three rules were built: one to identify ethers, one for esters and one for aldehydes (Figure 6).

```
(defrule ester
   (bin 15 is strong)
   (or (bin 24 is strong)
       (or (bin 25 is strong)
           (bin 26 is strong))
       (and (or (bin 22 is strong)
                (bin 23 is strong)
                (bin 24 is strong))
            (or (bin 25 is strong)
                (bin 26 is strong)
                (bin 27 is strong)))
        (and (bin 28 is medium)
            (or (bin 22 is strong)
                (bin 23 is strong)
                (bin 24 is strong))))
=>
  (RETURN 1 "IR_Classifier" "compound is an ester"))
```

Figure 6. Example of a Chemical Rule

The results of this pre-processing were *RETURN*ed to the neural net which then was able to choose a module specific to the chosen functional group. Later, the output from another pre-processor (a nearest neighbor classifier) was also input into CLIPS and rules were added to resolve differences between CLIPS' initial choice and the nearest-neighbor's choice.

CONCLUSIONS

CLIPS proved to be a very useful tool when used in this way. In ET and ATAFS, the Expert System Domain could easily be expanded to use a hybrid system similar to that of the Chemical Analysis problem, or use multiple copies of CLIPS operating independently or coordinating their efforts using some kind of Blackboard. In many applications, intelligence is not the main concern. User interface concerns in both ET and ATAFS were more important than the intelligence. ET has the added burden of operating in conjuction with a black box system (software written by a partner company). ATAFS' biggest concern was capturing every packet off of the network where the exercise was occurring and storing them in an easily accessible manner. The flexibility of C++ is better able to handle these tasks than CLIPS, but is not nearly as well suited for representing an expert coaching and grading a trainee howitzer crewman or reviewing a tank battle. While recognizing individual chemical compounds would be tedious task for CLIPS (and the programmer), recognizing the functional group the compound belongs to and passing the information along to a trained neural net is almost trivial.

Expert systems like CLIPS have both strengths and weaknesses, as do virtually all other methods of developing software. Object Oriented is becoming the most popular way to develop software, but it still has its shortfalls. Neural nets are gaining in popularity as the way to do Artificial Intelligence, especially in the media, but they too have their limits. By mixing different methods

and technologies, modifying and using existing software to interact with each other, software can be pushed to solving greater and greater problems.

REFERENCES

1. Gary Riley, et al, *CLIPS Reference Manual*, Version 5.1 of CLIPS, Volume III Utilities and Interfaces Guide, Lyndon B. Johnson Space Center (September 10, 1991).

2. Sally Shlaer and Stephen Mellor, *Object-Oriented Systems Analysis, Modeling the World in Data*. (P T R Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988).

3. Sally Shlaer and Stephen Mellor, *Object Lifecycles, Modeling the World in States*. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992).

4. Grady Booch, *Object Oriented Design with Applications*. (The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991).

5. James Rumbaugh, et al, *Object-Oriented Modeling and Design*. (Prentice-Hall, Inc., Englewood Cliffs, 1991).

6. Theodore Metzler and Ronald Gore, "Application of Hybrid AI to Identification of Chemical Compounds." *Proceedings of TECOM Artificial Intelligence Technology Symposium,* September 13-16, 1994, Aberdeen Maryland

CLIPS, APPLEEVENTS, AND APPLESCRIPT: INTEGRATING CLIPS WITH COMMERCIAL SOFTWARE

Michael M. Compton
compton@ptolemy.arc.nasa.gov
(415) 604-6776

Shawn R. Wolfe
shawn@ptolemy.arc.nasa.gov
(415) 604-4760

AI Research Branch / Recom Technologies, Inc.
NASA Ames Research Center
Moffett Field, CA 94035-1000

ABSTRACT

Many of today's intelligent systems are comprised of several software modules, perhaps written in different tools and languages, that together help solve the users' problem. These systems often employ a knowledge-based component that is not accessed directly by the user, but instead operates "in the background" offering assistance to the user as necessary. In these types of modular systems, an efficient, flexible, and easy-to-use mechanism for sharing data between programs is crucial. To help permit transparent integration of CLIPS with other Macintosh applications, the AI Research Branch at NASA Ames Research Center has extended CLIPS to allow it to communicate transparently with other applications through two popular data-sharing mechanisms provided by the Macintosh operating system; Apple Events (a "high-level" event mechanism for program-to-program communication), and AppleScript, a recently-released scripting language for the Macintosh. This capability permits other applications (running on either the same or a remote machine) to send a command to CLIPS, which then responds as if the command were typed into the CLIPS dialog window. Any result returned by the command is then automatically returned to the program that sent it. Likewise, CLIPS can send several types of Apple Events directly to other local or remote applications. This CLIPS system has been successfully integrated with a variety of commercial applications, including data collection programs, electronic forms packages, DBMSs, and email programs. These mechanisms can permit transparent user access to the knowledge base from within a commercial application, and allow a single copy of the knowledge base to service multiple users in a networked environment.

INTRODUCTION

Over the past few years there has been a noticeable change in the way that "intelligent applications" have been developed and fielded. In the early and mid-1980s, these systems were typically large, monolithic systems, implemented in LISP or PROLOG, in which the inference engine and knowledge base were at the core of the system. In these "KB-centric" systems, other parts of the system (such as the user interface, file system routines, etc.) were added on once the inferential capabilities of the system were developed. These systems were often deployed as multi-megabyte "SYSOUTs" on special-purpose hardware that was difficult if not impossible to integrate into existing user environments.

Today, intelligent systems are more commonly being deployed as a collection of interacting software modules. In these systems, the knowledge base and inference engine serve as a subordinate and often "faceless" component that interacts with the user indirectly, if at all.

Several such systems have been deployed recently by the Artificial Intelligence Research Branch at NASA's Ames Research Center:

The Superfluid Helium On-Orbit Transfer system (or SHOOT) was a modular expert system that monitored and helped control a Space-Shuttle-based helium transfer experiment which flew onboard the Endeavor orbiter during mission STS-57 in June, 1993. In that system, a CLIPS-based knowledge system running on a PC laptop on the Shuttle's Aft Flight Deck was embedded in a custom-developed C program that helped crew members (and ground controllers) perform transfer of cryogenic liquids in microgravity.

The Astronaut Science Advisor (also known as "PI-in-a-Box", or simply "[PI]") was another modular system that flew onboard the Space Shuttle in 1993 ([3]). [PI] ran on a Macintosh PowerBook 170 laptop computer and helped crew members perform an experiment in vestibular physiology in Spacelab. This system was comprised of a CLIPS-based knowledge system that interacted with HyperCard (a commercial user interface tool from Claris Corporation and Apple Computer) and LabVIEW (a commercial data acquisition tool from National Instruments) to monitor data being collected by the crew and offer advice on how to refine the experiment protocol based on collected data and, when necessary, troubleshoot the experiment apparatus. In that system, CLIPS ran as a separate application and communicated with HyperCard via specially-written external commands (XCMDs) that were linked into the HyperCard application.

The Prototype Electronic Purchase Request (PEPR) system ([1] and [2]) is yet another modular system that motivated the enhancements being described in this paper. The PEPR system uses a CLIPS-based knowledge system to validate and generate electronic routing slips for a variety of electronic forms used frequently at NASA Ames. A very important part of the PEPR system's capabilities is that it is able to work "seamlessly" with several commercial applications.

Interestingly, these applications represent a progression of integration techniques in which the knowledge-based component interacts with custom-developed programs. This progression starts with tightly-coupled, "linked" integration of components (in SHOOT), to integration via custom-built extensions to a specific commercial product (in [PI]), and finally to a general-purpose integration mechanism that can be used with a variety of commercial products (in PEPR).

REQUIREMENTS FOR THE PEPR SYSTEM

A brief description of the PEPR system will help explain how we determined our approach to the problem of integrating CLIPS with other commercial applications.

Our primary goal in the development of the PEPR system was to demonstrate that knowledge-based techniques can improve the usability of automated workflow systems by helping validate and route electronic forms. In order to do this, we needed to develop a prototype system that was both usable and provided value in the context of a real-world problem. This meant that in addition to developing the knowledge base and tuning the inference engine, we also needed to provide the users with other tools. These included software to fill out electronic forms and send them between users, as well as mechanisms that could assure recipients of these forms that the senders were really who they appeared to be, and that the data hadn't been altered, either accidentally or maliciously, in transit. We also wanted to provide users with a way by which to find out the status of previously-sent forms, in terms of who had seen them and who hadn't.

Of course, we didn't want to have to develop all of these other tools ourselves. Thankfully, commercial solutions were emerging to help meet these additional needs, and we tried to use these commercial tools wherever possible. Our challenge became, then, how to best integrate the CLIPS-based knowledge system component with these other commercial tools.

OUR FIRST TRY AT COMPONENT INTEGRATION: KEYBOARD MACROS

Our initial integration effort was aimed at developing a "seamless" interface between the knowledge system and the electronic forms package (which of course was the primary user interface). Our first try was to use a popular keyboard macro package to simulate the keystrokes and mouse clicks that were necessary to transfer the data from the forms program to CLIPS. The keyboard macro would, with a single keystroke, export the data on the form to a disk file, switch to the CLIPS application, and then "type" the `(reset)` and `(run)` commands (and the requisite carriage returns) into the CLIPS dialog window. This would then cause CLIPS to read in and parse the data from the exported file.

This worked fairly well, and was reasonably easy to implement (that is, it did not require modifying any software). However, the solution had several serious drawbacks. First, it required that the user have the keyboard macro software (a commercial product) running on his or her machine. Also, the macro simply manipulated the programs' user interface, and was therefore very distracting for the user because of the flurry of pull down menus and dialog boxes that would automatically appear and disappear when the macro was run. Most importantly, however, was that the keyboard macro approach required each user to have their own copy of CLIPS and the knowledge base running locally on their machine. This, of course, would have presented numerous configuration and maintenance problems; the "loadup" files would have to be edited to run properly on each user's machine, and fixing bugs would have involved distributing new versions of the software to all users (and making sure they used it). Another drawback was that we found that the keyboard macros would often "break" midway through their execution and not complete the operation. This would, of course, have also been very frustrating for users. As a result, this early system, although somewhat useful for demonstrations, was never put into production.

OUR SECOND TRY: APPLE EVENTS

Our next attempt at integrating the electronic forms package with CLIPS was motivated by a demonstration we saw that showed that it was possible to integrate the forms package we had selected with a commercial DBMS. In this demo, a user was able to bring up an electronic order form, and enter information such as, say, a vendor number. When the user tabbed out of the Vendor Number field, the forms software automatically looked up the corresponding information in a "vendor table" in a commercial DB running on a remote machine. The vendor's name and address automatically appeared in other fields on the form.

Of course, this sort of data sharing is quite common in many data processing environments. However, we found this capability exciting because it involved two Macintosh applications (including our forms package of choice), from different vendors, cooperating across an AppleTalk network, with virtually no assistance from the user. This was exactly the sort of "seamless" behavior we were seeking.

This functionality was provided by what are known as "Apple Events", a "high-level event mechanism" that is built into "System 7", the current major release of the Macintosh operating system. We were encouraged by the fact that the forms package we had selected supported the use of Apple Events to communicate with external programs. We thought it might be possible to implement our desired inter-program interface by making CLIPS mimic the DBMS application with respect to the sending and receiving of Apple Events. However, the particular set of Apple Events supported by that version of the forms software was limited to interaction that particular DBMS product. This was problematic for several reasons. First, the set of Apple Events being used were tailored to interaction with this particular data base product, and it wasn't clear how to map the various DB constructs they supported into CLIPS. Second, it would have required a

considerable amount of programming effort to develop what would have been the interface between a specific forms product and a specific data base product. We wanted to minimize our dependence on a particular product, in case something better came along in the way of a forms package or a DBMS.

So, we ended up not implementing the DB-specific Apple Event mechanism. However, we were still convinced that the Apple Event mechanism was one of the keys to our integration goals, and *did* implement several more general Apple Event handlers, which are described below.

WHAT ARE APPLE EVENTS?

The following provides a brief description of the Apple Event mechanism. As mentioned above, Apple Events are a means of sharing data between to applications, the support for which is included in System 7. An Apple Event has four main parts. The first two components are a 4-character alphanumeric "event class" and a 4-character "event ID". The third component identifies the "target application" to which the event is to be sent. The actual data to be sent is a collection of keyword/data element pairs, and can range from a simple string to a complex, nested data structure. (In addition to these, there are several other parts of the event record which specify whether a reply is expected, a reply time-out value, and whether the target application can interact with a user.) From a programmer's perspective, these events are created, sent, received, and processed using a collection of so-called "ToolBox" calls that are available through system subroutines.

Although Apple Events are a "general purpose" mechanism for sharing data between programs, programs that exchange data must follow a pre-defined protocol that describes the format of the events. These protocols and record formats are described in the *Apple Event Registry*, published periodically by Apple Computer.

```
Event Class: MISC
   Event ID: DOSC
Target App: <ProcessID>
Event Data: "----"
            "(load \"MYKB.CLP\")"
```

Figure 1. An Example Apple Event

Figure 1 shows an example of a "do script" event that might be sent to CLIPS. The event class is "MISC", and the event ID is "dosc". The 4-dash "parameter keyword" identifies the string as a "direct object" parameter, and is used by the receiving program to extract the data in the event record. The protocol associated with the "do script" event in the Apple Event Registry calls for a single string that contains the "script" to be executed by the target application. In this example, it's a CLIPS command for loading a knowledge base file. This example shows the simplest data format. More complex Apple Events may require numerous (and even embedded) keyword/data pairs.

The Apple Event mechanism is very powerful. However, we wanted a more flexible solution to our needs of integrating CLIPS with the commercial forms application. It would have required considerable effort to implement a Apple-Event based programmatic interface between CLIPS and the other applications with which we needed to share data. That is, we didn't want to have to code specific Apple Event handling routines for every other application we were using. This would have made it very difficult to "swap in" different applications should we have needed to. What we needed was a way by which we could use the power of Apple Events to communicate between CLIPS and other programs, but make it easier to code the actual events for a given program.

OUR THIRD TRY: APPLESCRIPT

Just as we were trying to figure out the best way to link CLIPS with the other programs we wanted to use, we learned about a new scripting technology for the Mac called AppleScript. This new scripting language provides most of the benefits of Apple Events (and is in fact based on the Apple Event mechanism) such as control of and interaction with Macintosh applications running either locally or remotely. In addition, it offers some other benefits that pure Apple Event programming does not. For example, using AppleScript, one is able to control and share data with an ever-growing array of commercial applications, without having to understand the details of the application's Apple Event protocol. AppleScript comes with a reasonably simple Script Editor that can be used to compose, check, run, and save scripts. In addition to providing all the constructs that one might expect in any programming language (variables, control, I/O primitives) it is also extendible and can even be embedded in many applications.

The thing that made AppleScript particularly appealing for our use was that it utilized the Apple Event handlers that we had already added to CLIPS. All that was necessary to permit the "scriptability" we desired was the addition of a new "Apple Event Terminology Extension" resource to our already-modified CLIPS application. This AETE resource simply provided the AppleScript Editor (and other applications) with a "dictionary" of commands that CLIPS could understand, and the underlying Apple Events that the AppleScript should generate.

Another very appealing aspect of integrating programs with AppleScript is more and more commercial software products are supporting AppleScript by becoming scriptable. That of course makes it much easier to take advantage of new software products as they come along. For example, we recently upgraded the forms tracking data base used by the PEPR system. We were able to replace a "flat-file" data base package with a more-powerful relational DBMS product with only minor modifications to the AppleScript code linking the DB to the other applications. This would have been far more difficult (if even possible) had we relied solely on integration with AppleEvents.

The main disadvantage to using AppleScript rather than Apple Events is that AppleScript is somewhat slower than sending Apple Events directly. However, the increased flexibility and power of AppleScript more than compensates for the comparative lack of speed.

```
tell application "CLIPS" of machine "My Mac"
      of zone "Engineering"
   do script "(reset)"
   do script "(run)"
   set myResult to evaluate "(send [wing-1] get-weight)"
   display dialog "Wing weight is " & myResult
end tell
```

Figure 2. An example AppleScript program

Figure 2 shows an example script that could be used to control CLIPS remotely. There are several things to note in this example. First, the commands are passed to CLIPS and executed as though they had been entered into the Dialog Window. The example shows both the "do script" command (which does not return a result) and the "evaluate" command (which does). The example also shows a "display dialog" command which is built in to AppleScript and displays the result in a modal dialog box. Of particular interest is that the CLIPS application is running on another Macintosh, which is even in another AppleTalk zone.

SPECIFIC CLIPS EXTENSIONS

The following paragraphs describe the actual CLIPS extensions that have been implemented to support the functionality described above. Note that some of these extensions were actually implemented by Robert Dominy, formerly of NASA Goddard Space Flight Center.

Receiving Apple Events

It's now possible to send two types of Apple Events to CLIPS. Each takes a string that is interpreted by CLIPS as though it were a command typed into the Dialog Window. The format of these Apple Events is dictated by the *Apple Event Registry*, and they are also supported by a variety of other applications. Note that CLIPS doesn't currently return any syntax or execution errors to the program that sent the Apple Events, so it is the sender's responsibility to ensure that the commands sent to CLIPS are syntactically correct.

The "do script" Event

The "do script" event (event class = MISC, event ID=DOSC) passes its data as a string which CLIPS interprets as if it were a command that were typed into the Dialog Window. It returns no value to the sending program.

The "evaluate" Event

The "evaluate" event (event class = MISC, event ID=EVAL) is very similar to the do script event, and also passes its data as a string which CLIPS interprets as if it were a command that were typed into the Dialog Window. However, it does return a value to the sending program. This value is always a string, and can be up to 1024 bytes in length.

Sending Apple Events from CLIPS

The two Apple Events described above can also be sent by CLIPS from within a knowledge base. Of course, the application to which the events are sent must support the events or an error will occur. However, as mentioned above, the "do script" and "evaluate" events are very common and supported by many Mac applications.

SendAEScript Command

The SendAEScript command sends a "do script" event and can appear in a CLIPS function or in the right-hand-side of a rule. The syntax of the SendAEScript command is as follows:

```
(SendAEScript <target app> <command>)
```

In the above prototype, `<target app>` is an "application specification" and `<command>` is a valid command understandable by the target application. An application specification can have one of

three forms; a simple application name, a combination application name, machine name and AppleTalk Zone name, or a process identifier (as returned by `PPCBrowser`, described below). The `SendAEScript` command returns zero if the command is successfully sent to the remote application, and a variety of error codes if it was not. Note that a return code of zero does *not* guarantee that the command was successfully executed by the remote application; only that it was sent successfully .

The following examples show each of the application specification types.

```
CLIPS>(SendAEScript "HyperCard" "put \"hello\" into msg")
0
CLIPS>
```

The above example sends a “do script” Apple Event to HyperCard running on the local machine, and causes it to put “hello” into the HyperCard message box.

```
CLIPS>
(SendAEScript "HyperCard" "John's Mac" "R&D" "put \"hello\" into msg")
0
CLIPS>
```

The above example sends a similar “do script” Apple Event to HyperCard running on a computer called “John’s Mac” in an AppleTalk zone named “R&D”. Note that it is necessary to “escape” the quote characters surrounding the string “hello” to avoid them being interpreted by the CLIPS reader.

SendAEEval Command

The SendAEEval command is very similar to the SendAEScript command, differing only in that it returns the value that results from the target application evaluating the command.

```
(SendAEEval <target app> <command>)
```

The following examples show CLIPS sending a simple command to HyperCard running on the local machine:

```
CLIPS> (SendAEEval "HyperCard" "word 2 of \"my dog has fleas\" ")
"dog"
CLIPS>
```

Note that the result returned by SendAEEval is always a string, e.g.:

```
CLIPS> (SendAEEval "HyperCard" "3 + 6")
"9"
CLIPS>
```

The `SendAEEval` command does not currently support commands that require the target application to interact with its user. For example, one could not use `SendAEEval` to send an "ask" command to HyperCard.

PPCBrowser Command

The `PPCBrowser` function permits the CLIPS user to select an AppleEvent-aware program that is currently running locally or on a remote Mac. This command brings up a dialog box from which the user can click on various AppleTalk zones, machine names and “high-level event aware”

applications. It returns a pointer to a “process ID” which can be bound to a CLIPS variable and used in the previously-described “send” commands.

```
CLIPS>(defglobal ?*myapp* = (PPCBrowser))
CLIPS> ?*myapp*
<Pointer: 00FF85E8>
```

The above example doesn’t show the user’s interaction with the dialog box.

GetAEAddress Command

The `GetAEAddress` function is similar to `PPCBrowser` in that it returns a pointer to a high-level aware application that can then be bound to a variable that’s used to specify the target of one of the “SendAE” commands described earlier. Rather than presenting a dialog box to the user, however, it instead takes a “target app” parameter similar to that described above.

```
(GetAEAddress <target app>)
```

The following example shows the GetAEAddress function being used to specify the target of a `SendAEEval` function call.

```
CLIPS> (defglobal ?*myapp* = (GetAEAddress "HyperCard" "Jack’s Mac" "R&D"))
CLIPS> (SendAEEval ?*myapp* "8 + 9")
"17"
CLIPS>
```

TimeStamp Command

Another extension we've made is unrelated to inter-program communication. We have added a `TimeStamp` command to CLIPS. It returns the current system date and time as a string:

```
CLIPS>(TimeStamp)
"Wed Sep 7 12:34:56 1994"
CLIPS>
```

POSSIBLE FUTURE EXTENSIONS

In addition to the CLIPS extensions described above, we are also looking into the possibility of implementing several other enhancements. First, we want to generalize the current Apple Event sending mechanisms to permit the CLIPS programmer to specify the event class and event ID of the events to be sent. This is a relatively straightforward extension if we limit the event data to a string passed as the “direct object” parameter. It would be somewhat harder to allow the CLIPS programmer to specify more complex data structures, because we would have to design and implement a mechanism that allows the CLIPS programmer to construct these more complex combinations of keywords, parameters, and attributes. We will probably implement these extensions in stages.

Another extension we’re considering is to make CLIPS “attachable”. This would permit the CLIPS programmer to include pieces of AppleScript code in the knowledge base itself. This would significantly enhance the power of CLIPS, as it would make it possible to compose, compile, and execute AppleScript programs from within the CLIPS environment, and save these programs as part of a CLIPS knowledge base.

ACKNOWLEDGMENTS

Some the extensions described in this paper were designed and implemented by Robert Dominy, formerly of NASA's Goddard Space Flight Facility in Greenbelt, Maryland.

Also, Jeff Shapiro, formerly of Ames, ported many of the enhancements described herein to CLIPS version 6.0.

REFERENCES

1. Compton, M., Wolfe, S. 1993 Intelligent Validation and Routing of Electronic Forms in a Distributed Workflow Environment.. Proceedings of the Tenth IEEE Conference on AI and Applications.

2. Compton, M., Wolfe, S. 1994 AI and Workflow Automation: The Prototype Electronic Purchase Request System. Proceedings of the Third Conference on CLIPS.

3. Frainier, R., Groleau, N., Hazelton, L., Colombano, S., Compton, M., Statler, I., Szolovits, P., and Young, L., 1994 PI-in-a-Box, A knowledge-based system for space science experimentation, AI Magazine, Volume 15, No. 1, Spring 1994, pp. 39-51.

TARGET'S ROLE IN KNOWLEDGE ACQUISITION, ENGINEERING, AND VALIDATION

Keith R. Levi and Ahto Jarve
Department of Computer Science
Maharishi International University
1000 North Fourth Street, FB1044
Fairfield, Iowa 52557

We have developed an expert system using the CLIPS expert system tool and the TARGET task analysis tool. The application domain was a help desk for a commercial software product. The expert system presently encompasses about 200 rules and covers the most critical and common issues encountered by the technical support personnel for the help desk. The expert system is a rule-based data-driven reasoning system for diagnosis, testing, and remediation of customer problems. The system is presently undergoing validation testing by the help-desk personnel.

The focus of this paper is on our experiences in using TARGET to assist in the processes of knowledge acquisition, knowledge engineering, and knowledge verification. TARGET greatly facilitated each of these processes. In contrast to the traditional knowledge acquisition process in which one knowledge engineer interviews the domain expert and another takes notes on the interaction, we found that only a single knowledge engineer was needed when using TARGET. The domain expert observed the knowledge engineer record tasks in TARGET as the interview proceeded. This facilitated communication between the knowledge engineer and the expert because it was clear to each just what knowledge was being communicated. It also provided an explicit and easily modifiable graphical representation of the knowledge in the system as the session proceeded.

AN EXPERT SYSTEM FOR CONFIGURING A NETWORK FOR A MILSTAR TERMINAL

Melissa J. Mahoney
Dr. Elizabeth J. Wilson

Raytheon Company
1001 Boston Post Road
Marlboro, MA 01572

melissa@tif352.ed.ray.com
bwilson@sud2.ed.ray.com

ABSTRACT

This paper describes a rule-based expert system which assists the user in configuring a network for Air Force terminals using the Milstar satellite system. The network configuration expert system approach uses CLIPS. The complexity of network configuration is discussed, and the methods used to model it are described.

INTRODUCTION

Milstar Complexity

Milstar is a flexible, robust satellite communications system which provides world-wide communication capability to a variety of users for a wide range of tactical and strategic applications. Milstar is interoperable requiring simultaneous access by Air Force, Navy and Army terminals. The transmit/receive capability of these terminals ranges from small airborne units with limited transmission power to stationary sites with large fixed antennas.

The wide range of applications of the Milstar system includes services capable of supporting status reporting, critical mission commands, point-to-point calls, and networks. The same system that provides routine weather reports must also allow top-level mission coordination among all services. This diversity results in traffic which is unpredictable and variable. The system must accommodate both voice and data services which can be transferred at different rates simultaneously. In order to protect itself from jamming and interference, the Milstar waveform includes a number of signal processing features, such as frequency hopping, symbol hop redundancy, and interleaving, some of which are among the network parameters defined at service setup.

The configuration of each terminal is accomplished through the loading of adaptation data. This complex data set provides all the terminal specific and mission specific parameters to allow the terminal to access the satellite under the mission conditions with the necessary resources to establish and maintain communication. Network definitions are a subset of this adaptation data.

Defining a network for a terminal is a challenging task requiring a large body of knowledge, much of it heuristic. The systems engineers who have been on the Milstar program for many years have amassed this knowledge; many of them have been reassigned to other programs. Since the recent launch of the first Milstar satellite, testing activity has been very high and expertise in this area is not readily available. When testing ends and Milstar becomes fully operational, the terminals will be operated by personnel whose Milstar knowledge is considerably less than that of the systems engineers. To solve the problem of having a user with limited knowledge take advantage of the many network configuration capabilities of the Milstar

system, one of the systems engineers proposed developing an expert system to encapsulate the vast amount of knowledge required to successfully design network definitions accurately and reliably.

Network Complexity

A Milstar network consists of 82 parameters which are interrelated. Some relationships are well-understood, but many are obscure. The few existing domain experts spend a significant amount of time constructing network definitions starting from a high-level mission description usually provided by a commanding authority. Not only should the network allow communication, but it should do so in a manner which makes optimal use of satellite resources. The expert, then, is faced with an optimization problem: to define a network whose parameters are consistent with each other while minimizing the use of payload resources.

The complexity of network configuration is compounded by the number and locations of the terminals which will participate in the network. Additional terminals add more constraints to the problem space. The resource requirements of different terminals may vary, and their locations may be dynamic, as in the case of airborne terminals.

The complexity further increases when configuring a terminal to participate in several networks simultaneously. Certain resources cannot be shared among a terminal's active networks.

Knowledge Acquisition

Several system engineers were available as knowledge sources. Initially, they were asked to review each of the network parameters and to describe obvious relationships between them. Remarks about rules, exceptions, and values were recorded and coded into prototype rules. These rules were reviewed with the experts and modified as needed to confirm that the information had been accurately translated.

In addition to interviewing the domain experts, scores of documents were read. Tables and charts were constructed to focus on the relationships between only several parameters at a time. Gradually, a hierarchy of knowledge was formed.

The knowledge base was partitioned into functional areas (e.g., hardware parameters, mission-level parameters, terminal-specific parameters, and service-specific parameters). A knowledge engineer was assigned to each one. When it was deemed that most of the knowledge had been acquired in one of these areas, the entire knowledge engineering team of four would check the functional area's rules with respect to each network type (of which there are 11). This cross-checking resulted in more rules which captured the exceptions and special cases associated with the more general rules belonging to the functional area.

In-house operations and maintenance training courses provided the knowledge engineers with an opportunity to learn Milstar as an Air Force terminal operator. This provided the hands-on training that taught lessons not documented in specifications.

THE EXPERT SYSTEM

Scope

The scope of the task was to develop an expert system to assist the user in configuring a network definition for a single terminal. This scope represents the necessary first step in later building an extended system which will address the issues of multiple terminals and multiple networks.

System Componenets

The expert system consists of three major components: a graphical user interface (GUI), a rule and fact base, and an interface between those two components. The GUI performed the more trivial consistency checks between the input parameters. For example, if a voice network is being configured, then the only valid data rate for the network is 2400 bps. The GUI ghosts out all other choices. The GUI also checks the syntax and ranges of user-entered data.

The interface code controls the flow of the expert system. It reads in the user-supplied input items, translates them into facts, loads the appropriate rules; when the rules have finished firing, the interface code passes their results to the GUI. This process continues as the user moves between the screens.

Rules and facts were used to check the more complicated relationships between the parameters. Rules were not used to perform trivial tasks (e.g., syntax checking) nor inappropriate tasks (e.g., controlling the flow).

Software Tools

CLIPS was chosen for this project because of its forward-chaining inference engine, and its ability to run as both a standalone and an embedded application. Other benefits were its object-oriented design techniques and its level of customer support.

The rules and facts were developed, tested and refined on a Sun workstation under UNIX. They were ported to an IBM 486/66, the target environment. A point-and-click graphical user interface was developed for the PC using Visual Basic. Borland C++ was used to write the interface code between the GUI and the CLIPS modules.

Currently, the expert system contains about 100 rules and 1000 facts.

Objects

CLIPS Object-Oriented Language (COOL) was used where appropriate.

In the problem domain, terminals are represented as objects having attributes such as a terminal id, an antenna size, a latitude/longitude location, and a port usage table. The port usage table is made up of 34 ports. A port is an object which has attributes such as data rate and i/o hardware type. Terminals are objects to prepare for future expansion to the analysis of a network definition for multiple terminals.

CLIPS offers a query system to match instances and perform actions on a set of instances. For example, CLIPS can quickly find the terminal requiring the most robust downlink modulation by keying off each terminal's antenna size and satellite downlink antenna type. Satisfying the needs of this terminal would be a constraint to ensure reliable communication among a population of terminals.

Templates

Valid parameter relationships are defined using templates and asserted as facts during system initialization. For example, only a subset of uplink time slots are valid with each combination of data rate, uplink modulation, and uplink mode. Figure 1a shows a table relating these parameters. Figure 1b shows a template modeled after this table. Figure 1c shows a deffacts structure which will assert the facts contained in the table by using the template.

Primary U/L Channels

U/L Modulation		Data Rate (bps)					
		2400	1200	600	300	150	75
HHR FSK	2	1,2,3,4	--	--	--	--	--
	4	5,6	1,2,3,4	--	--	--	--
	8	7	5,6	1,2,3,4	--	--	--
	16	--	7	5,6	1,2,3,4	--	--
	32	--	--	7	5,6	1,2,3,4	--
	64	--	--	--	7	5,6	1,2,3,4
	128	--	--	--	--	7	5,6
	256	--	--	--	--	--	7
LHR FSK	10	--	--	--	7	5,6	1,2,3,4
	20	--	--	--	--	7	5,6
	40	--	--	--	--	--	7

Secondary U/L Channels

U/L Modulation		Data Rate (bps)		
		300	150	75
HHR FSK	2	1,2,3,4	--	--
	4	5,6	1,2,3,4	--
	8	7	5,6	1,2,3,4
	16	--	7	5,6
	32	--	--	7

Figure 1a. Table Relating Uplink Time Slot, Uplink Mode, Uplink Modulation and Data Rate

```
; valid-dr-ts

; this template sets up associations of uplink mode,

; uplink modulation, and data rate/timeslot pairs

(deftemplate valid-dr-ts)
  (field u1-mode
    (type SYMBOL)
    (default ?NONE))
  (field u1-modulation
    (type SYMBOL)
    (default ?NONE))
  (multifield dr-ts
    (type ?VARIABLE)
   (default ?NONE))
```

Figure 1b. Template Corresponding to Table in Figure 1a.

[See Appendix]

Figure 1c. Deffacts Representation of Table in Figure 1a

Rules

Several rules exist for most of the 82 parameters. The left-hand side of each rule specifies a certain combination of patterns which contain some bounded values (i.e., network parameters that already have values assigned). The right-hand side of each rule performs three actions when it fires. First, it assigns a suggested parameter value. Second, it creates a multifield containing other valid, but less optimal, parameter values. And, third, it creates an explanation associated with the parameter. The GUI highlights the suggested value, but allows the user to change it to one of the other valid choices. Figure 2 shows a rule which fires when the expert system does not find any valid uplink time slots*.

[See Appendix]

Figure 2. Rule Using the Facts Shown in Figure 1c

Explanations

The explanation associated with each parameter reflects the left-hand side of the rule which ultimately assigned the parameter's value. An explanation template contains the text, and a flag to indicate whether a positive rule fired (i.e., a valid parameter value was suggested), or a negative rule fired (i.e., the expert system could find no valid value for the parameter based upon preconditions). In the latter case, the GUI immediately notifies the user that the expert system is unable to proceed, and it states a recommendation to fix the problem. Normally the user is instructed to back up to a certain parameter whose current value is imposing the unsatisfiable constraint. The explanation facility is invoked by pointing the mouse on the parameter and clicking the right button.

Output

After the user has proceeded through the screens and values have been assigned to all network parameters, s/he may choose various output media and formats. In production, the completed network will be converted to hexadecimal format and downloaded into the terminal's database which is then distributed to the field.

Future Extensions

As stated earlier, a terminal class was established to allow for future expansion to the analysis of a network with respect to multiple terminals. Taking this idea one step further would involve creating a network class, and making a network definition an instance of it. Multiple networks could then be considered simultaneously when writing rules to suggest values for certain parameters. For example, a terminal cannot participate in two networks which use the same uplink time slot. Advancing the expert system to this level of consistency checking would improve its utility significantly. CLIPS promotes a modular approach to design which allows the developer to plan for future extensions like these.

* The problem of refiring needed to be addressed. Refiring occurred because when a rule modifies a fact, the fact is implicitly retracted and then reasserted. Since the fact is “new,”the rule fires again immediately. This problem was solved by including a check in the antecedent whether the valid choices multifield is currently the same as what the rule will set it to be. If the fields are identical, then the rule will not refire.

CONCLUSIONS

An expert system has proven to be a valuable tool in the configuration of Milstar terminal network parameter sets. The generation of these parameter sets is complex and primarily heuristic lending itself to an expert system approach. CLIPS and COOL has provided an effective and efficient development environment to capture the knowledge associated with the application domain and translate these relationships into a modular set of rules and facts. The decision to use rules vs. facts, CLIPS vs. C++, rules vs. GUI was made after careful consideration of the overall implementation strategy and the most efficient means to provide the expertise.

The complexity of the problem domain, variety of sources of expertise (design engineers, operational support, and program documentation), and broad and varying scope of the applications has made this project a significant knowledge acquisition challenge. The knowledge engineering team approach has been successful in translating the vast and subtle nuances of the implementation strategies and in developing a synergetic composite of the available expertise.

The combined use of rules, facts, objects, C++ code, and a graphical interface has provided a rich platform to capture the Milstar knowledge and transform this knowledge into a modular tool staged for expansion and improvement.

REFERENCES

1. Basel, J., D'Atri, J. and Reed, R., "Air Force Agile Beam Management," Raytheon Company, Equipment Division, Marlborough, Massachusetts, June 15, 1989.

2. Giarratano, J. and Riley, G., *Expert Systems: Principles and Programming*, ed. 2, PWS Publishing Company, Boston, Massachusetts, 1994.

3. Vachula, G., "Milstar System Data Management Concept," GMV:92:05, Raytheon Company, Equipment Division, Communication Systems Laboratory, Marlborough, Massachusetts, February 7, 1992.

APPENDIX

Figure 1b. Template Corresponding to Table in Figure 1a

```
; this deffacts defines valid data rate and timeslot pairs
; for each combination of uplink mode and uplink modulation

(deffacts set-valid-dr-ts
   (valid-dr-ts
      (u1-mode PRIMARY)
      (u1-modulation HHRFSK2)
      (dr-ts 2400 bps with time slots 1-4))
   (valid-dr-ts
      (u1-mode PRIMARY)
      (u1-modulation HHRFSK4)
      (dr-ts 2400 bps with time slots 5-6 1200 bps with time slots 1-4))
   (valid-dr-ts
      (u1-mode PRIMARY)
      (u1-modulation HHRFSK8)
      (dr-ts 2400 bps with time slot 7 1200 bps with time slots 5-6))
   (valid-dr-ts
      (u1-mode PRIMARY)
```

```
   (u1-modulation HHRFSK16)
   (dr-ts 1200 bps with time slot 7
    600 bps with time slots 5-6 150 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation HHRFSK32)
   (dr-ts 600 bps with time slot 7
    300 bps with time slots 5-6 150 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation HHRFSK64)
   (dr-ts 300 bps with time slot 7
    150 bps with time slots 5-6 75 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation HHRFSK128)
   (dr-ts 150 bps with time slot 7 75 bps with time slots 5-6))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation HHRFSK256)
   (dr-ts 75 bps with time slot 7))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation LHRFSK10)
   (dr-ts 300 bps with time slot 7
    150 bps with time slots 5-6 75 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation LHRFSK20)
   (dr-ts 150 bps with time slot 7 75 bps with time slots 5-6))
(valid-dr-ts
   (u1-mode PRIMARY)
   (u1-modulation LHRFSK40)
   (dr-ts 75 bps with time slot 7))
(valid-dr-ts
   (u1-mode SECONDARY)
   (u1-modulation HHRFSK2)
   (dr-ts 300 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode SECONDARY)
   (u1-modulation HHRFSK4)
   (dr-ts 300 bps with time slots 5-6
(valid-dr-ts
   (u1-mode SECONDARY)
   (u1-modulation HHRFSK8)
   (dr-ts 300 bps with time slot 7
    150 bps with time slots 5-6 75 bps with time slots 1-4))
(valid-dr-ts
   (u1-mode SECONDARY)
   (u1-modulation HHRFSK16)
   (dr-ts 150 bps with time slot 7 75 bps with time slots 5-6))
(valid-dr-ts
   (u1-mode SECONDARY)
   (u1-modulation HHRFSK32)
   (dr-ts 75 bps with time slot 7)))
```

Figure 2. Rule Using the Facts Shown in Figure 1c

```
(defrule 12bults-009
   "this rule fires when the terminal's required u/1 modulation is
    more robust than the implied u/1 modulation using any u/1 time slot."

   (outnet  (name data-rate)
            (value ?dr-value))         ; get value of data-rate
   (outnet  (name u1-mode)
            (value ?u1-mode))          ; get value of u1-mode
   (outnet  (name u1-antenna-type)
            (value ?u1-ant-type))      ; get value of u1-ant-type
   (outnet  (name net-type)
            (value ?net-type))         ; get value of net-type
   ?fact <- (outnet  (name u1-time-slot)           ; get address
                     (choices $?oldchoices))       ; of time slot
   (test (neq $?oldchoices =(mv-append nil)))      ; don't refire
                                                   ; (* see note)
   (terminal ?term-id)                ; get the terminal id
   (u1modmap   (ant-size =(get-ant-size  ?term-id)
               (u1-ant ?ul-ant-type)
               (u1-modulation ?ul-mod-terminal))
                     ; get the minimum u1 mod this terminal
                     ; can use given its antenna size
                     ; and the u1 antenna type
   (outnet  (name u1-modulation)
            (value ?u1-mod-type))           ; get the u/1 modulation type
   (valid-u1-quad (data-rate ?dr-value)     ; get all valid modulations
                  (u1-mode ?u1-mode)        ; for this data rate
                  (u1-mod-type ?u1-mod-type)
                  (u1-mods $?all-valid-mods))
   (most-to-least u1 ?u1-mode $?allmods)    ; get all modulations
   (test (more-robustp ?u1-mod-terminal
                       (first $?all-valid-mods) $?allmods))
                     ; if the terminal's required robustness is
                     ; more robust than the most robust implied
                     ; modulation
   (valid-dr-ts   (u1-mode ?u1-mode)
                  (u1-modulation ?u1-mod-terminal)
                  (dr-ts $?all-dr-ts)) ; get all pairs of valid data rates
                                       ; and timeslots for this modulation
   ?explain <- (explanation
                  (param-name ul-time-slot)
                                       ; get previous explanation
                  (text ?oldtext))     ; associated with this parameter
   =>

   (modify ?fact  (suggest nil)                    ; no suggested value
                  (choices =(mv-append nil)))      ; no valid choices
   (if (neq ?net-type MCE)                         ; tailor the explanation
      then                                         ; to the network type
      (bind ?newtext
         (str-implode
            (mv-append
```

```
               (str-explode
"This terminal cannot reliably support this service's data rate using the
 current u/1 antenna type.  Using any time slot with this data rate would
 imply an insufficient u/1 modulation for this terminal. Either use a
 stronger u /1 antenna type, or lower the data rate.  To satisfy this
 terminal's u/1 modulation requirements, the valid data rate/time slot
 pairs are:"
               ) ; end explode
               $?all-dr-ts
            )  ;end append
         )  ;end implode
      )  ;end bind
      else
      (bind ?newtext
         (str-implode
            (mv-append
               (str-explode
"This terminal cannot reliably support this service's data rate using the
 current u/1 antenna type.  Using any time slot with this data rate would
 imply an insufficient u/1 modulation for this terminal.  Either use a
 stronger u/1 antenna type, or lower the data rate.  To satisfy this
 terminal's u/1 modulation requirements, the valid data rate/time slot
 pairs are:"
               ) ; end explode
               $?all-dr-ts
               (str-explode
".  Since this is an MCE network, time slot 4 should be used.")
            )  ;end append
         )  ;end implode
      )  ;end bind
   )  ;end if
   (explain ?explain  ?old text  ?newtext YES))
```

EXPERT SYSTEM TECHNOLOGIES FOR SPACE SHUTTLE DECISION SUPPORT: TWO CASE STUDIES

Christopher J. Ortiz
Workstations Branch (PT3)
Johnson Space Center
Houston, TX 77058
chris.ortiz@jsc.nasa.gov

David A. Hasan
LinCom Corporation
1020 Bay Area Blvd., Suite 200
Houston, TX 77058
hasan@gothamcity.jsc.nasa.gov

ABSTRACT

This paper addresses the issue of integrating the C Language Integrated Production System (CLIPS) into distributed data acquisition environments. In particular, it presents preliminary results of some ongoing software development projects aimed at exploiting CLIPS technology in the new Mission Control Center (MCC) being built at NASA Johnson Space Center. One interesting aspect of the control center is its distributed architecture; it consists of networked workstations which acquire and share data through the NASA/JSC-developed Information Sharing Protocol (ISP). This paper outlines some approaches taken to integrate CLIPS and ISP in order to permit the development of intelligent data analysis applications which can be used in the MCC.

Three approaches to CLIPS/ISP integration are discussed. The initial approach involves clearly separating CLIPS from ISP using *user-defined functions* for gathering and sending data to and from a local storage buffer. Memory and performance drawbacks of this design are summarized. The second approach involves taking full advantage of CLIPS and the CLIPS Object-Oriented Language (COOL) by using *objects* to directly transmit data and state changes from ISP to COOL. Any changes within the object slots eliminate the need for both a data structure and external function call thus taking advantage of the object matching capabilities within CLIPS 6.0. The final approach is to treat CLIPS and ISP as *peer toolkits*. Neither is embedded in the other, rather the application interweaves calls to each directly in the application source code.

INTRODUCTION

A new control center is being built at the NASA Johnson Space Center. The consolidated Mission Control Center (MCC) will eventually replace the existing control center which has been the location of manned spaceflight flight control operations since the Gemini program in the 1960s. This paper presents some preliminary results of projects aimed at incorporating knowledge-based applications into the MCC. In particular, it discusses different approaches being taken to integrate CLIPS with the MCC Information Sharing Protocol (ISP) system service.

MCC AND ISP

The new control center architecture differs significantly from the current one. The most prominent difference is its departure from the mainframe-based design of the existing control center. The new MCC architecture is aggressively distributed. It consists of UNIX workstations (primarily DEC Alpha/OSF1 machines at present) connected by a set of networks running

TCP/IP protocols. Exploitation of commercially available products which will be relatively easy to replace and upgrade has been a prime motivating factor in this change. Particular attention has been paid to the use of standard hardware, and commercial off-the-shelf (COTS) software is being used wherever possible.

With a mainframe computer at the center of all flight support computation, the process of presenting telemetry and computational results to flight controllers was really a matter of "pushing" the relevant data out of the mainframe onto the appropriate flight control terminals. In recent years, the task of acquiring telemetry on the system of MCC upgrade workstations has been a matter of requesting the telemetry data from the mainframe. The central computer has served as the one true broker for telemetry data and computations.

In the distributed MCC design, the notion of a central telemetry and computation broker has disappeared. In fact, terminals have disappeared. The flight control consoles consist of UNIX workstations which have access to telemetry streams on the network but must share data instead of relying on the mainframe for common computations.

Software applications running on the MCC workstations will obtain telemetry data from a system service called the Information Sharing Protocol (ISP). ISP is a client/server service. Distributed clients may request ISP support from a set of "peer" ISP servers. The servers are responsible for extracting data from the network. Client applications needing those data initiate a session with the servers (possibly from different machines), using the ISP client application program interface (API). These clients *subscribe* to data from the ISP servers, which deliver the data asynchronously to the clients as the data change. Parenthetically, the ISP API provides *data source independence*. Thus, ISP client applications may be driven by test data for validation and verification, playback data for training or live telemetry for flight support.

It is the intent of the new MCC architecture that hardware (workstations, routers, network cabling, etc...) will take a backseat to the "software platform". ISP is a prominent element of this platform, since it is the data sharing component of the system services in addition to providing a telemetry acquisition service. Indeed, the architecture presupposes that many of the functions previously handled in the mainframe program (e.g., display management, limit sensing, fault summary messages, "comm fault" detection) will now be carved up into smaller, easier to maintain application programs.

ISP supports the integration of these applications by allowing clients to receive data that have been *published* by other clients. These shared data will in many instances be computations which were previously handled internally to the mainframe, e.g., spacecraft trajectories and attitude data, but with greater frequency, the shared data are expected to be "higher order information" derived from analyses of telemetry data.

Some of the data analysis necessary for the generation of this higher order information will be derived from rule-based pattern matching of telemetry. One example of this is the Bus Loss Smart System which is used by EGIL flight controllers to identify power bus failures. Another example discussed later in this paper is the Operational Instrumentation Monitor (oimon) which assesses the health of electronic components involved in the transmission of Shuttle sensor data down to Earth.

Rule-based applications can capture the often heuristic procedures used by flight controllers to perform their duties. A number of such applications are currently under development by flight control groups where the knowledge-based approach contributes to getting the job done "better, faster and cheaper". CLIPS is being used in a number of these. In the discussion that follows, methods for combining the telemetry acquisition and data sharing functionality of ISP with the pattern matching capabilities of CLIPS are discussed.

AN EMBEDDED APPROACH

CLIPS is a data-driven language because it uses data in the form of facts to activate and fire if-then rules which result in a change of execution state for the computer. CLIPS was designed to be embedded in other applications thus allowing them the ability to perform complex tasks without the use of complex programming on the part of the rule developer. Since ISP is a transport protocol for both receiving and transmitting information, it only makes sense to find a way to integrate the two toolkits to provide CLIPS developers with a reliable transport mechanism for data.

In our work using ISP and CLIPS we have taken two approaches in integrating the two toolkits. The first was to directly embed ISP within CLIPS and provide a simple set of user-defined functions which allow the CLIPS developer to communicate via ISP. The second approach was to use both toolkits as peers allowing the developer to have complete control over the integration.

The first method of embedding ISP into CLIPS grew out of three separate attempts to balance data, speed and ease of use concerns for the application developer. The first attempt at integration (Figure 1) consisted of creating a separate *data layer* with which ISP and CLIPS could communicate. This layer provides a storage location for all the change only data that ISP receives as well as hiding the low level ISP implementation. This approach provided several challenges in how ISP and CLIPS would communicate with the data layer.

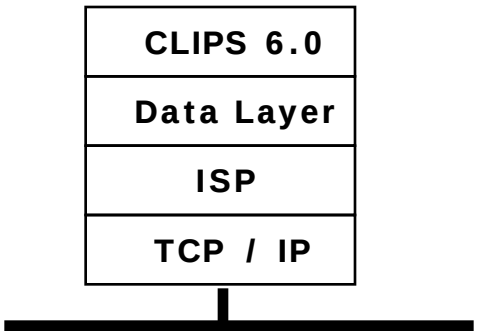


Figure 1. The first attempt at integration

A simple set of commands was added to CLIPS to aid in the communication with ISP.

```
(connect-server [CLIPS_TRUE / CLIPS_FALSE])
(subscribe-symbol [symbol_name] [CLIPS_TRUE / CLIPS_FALSE])
(enable-symbols [CLIPS_TRUE / CLIPS_FALSE])
```

The *connect-server* command establishes or disconnects a link with the ISP server and registers the application as requesting ISP data. Likewise, *subscribe-symbol*, informs the ISP server of which data elements are requested or no longer needed by the application. The *enable-symbols* command begins and ends the flow of data to the application.

Other API calls were added to CLIPS to enable an application to publish data to the ISP server for use in other CLIPS or ISP only applications.

```
(publish-symbol [symbol-name] [EXCLUSIVE])
(publish [VALUE/LIMIT/STATUS/Message] [symbol-name] [data] [time])
(unpublish-symbol [symbol-name] )
```

The *publish-symbol* command informs the ISP server that the CLIPS application wants permission to send data under a given symbol-name. The optional EXCLUSIVE parameter informs the server that the requesting application should be the only application allowed to change the value.

The final set of APIs allows ISP to communicate with CLIPS in the form of facts.

```
(want-isp-event [CYCLE/LIMIT] [CLIPS_TRUE/CLIPS_FALSE])
```

The *want-isp-event* command will activate or deactivate cycle or limit facts to be published in the fact list.

This first attempt provided a solid foundation for communication between the two tool kits. However, the use of a separate data layer proved to be cumbersome to implement. The data layer needed to have a dynamic array and a quick table look-up mechanism as well as a set of functions to provide CLIPS with the ability to check values. Each time a rule needed a data value, an external function would have to be called. This proved to be a costly option in the amount of time needed to call the external function. A better method was clearly needed.

Our second attempt at integration ISP and CLIPS consisted of the elimination of the data layer (Figure 2). All of the ISP data would be fed to CLIPS via facts. This method had several advantages. First, it reused the ISP functions developed earlier. Second, it allowed CLIPS to store all ISP data as facts. Finally, CLIPS application developers could do direct pattern matching on the facts to retrieve the data.

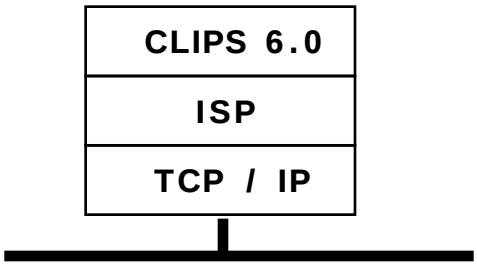


Figure 2. Removal of the data layer

The use of facts provided a few smaller challenges in the update and management of the fact list. There needed to be a way to find an old fact that had not been used and retract it when new data arrived. This could be done at the rule level or in the integration level. At the rule level, the application programmer would be required to create new rules to seek out and remove obsolete facts. At the integration level, time would be spent looking for matching facts with similar IDs to be removed.

The final approach at integration of ISP with CLIPS involved the use of the CLIPS Object-Oriented Language (COOL). Each data packet would be described as an instance of a class called MSID. The MSID class would provide a data storage mechanism for storing the data name, value, status, time tag, and server acceptance information.

```
(defclass MSID
   (is-a USER)
   (role concrete)
   (pattern-match reactive)
   (slot value ( create-accessor read-write) (default 0.0))
   (slot status ( create-accessor read-write)(default 0.0))
```

```
   (slot time (create-accessor read-write)(default 0.0))
   (slot accepted (create-accessor read-write)
      (default CLIPS_TRUE)))
```

A second advantage of using the object implementation consists of inheriting constructors and destructors. When an instance of the MSID class is created, a constructor is activated and the symbol is automatically subscribed. On the other hand, when an symbol is no longer needed a destructor is activated and the symbol is automatically unsubscribed. Constructors and destructors free the application programmer from worrying about calling the appropriate ISP APIs for creating and deleting symbols.

```
(defmessage-handler MSID init after()
   (subscribe-symbol (instance-name ?self) CLIPS_TRUE))

(defmessage-handler MSID delete before()
   (enable-symbols CLIPS_FALSE)
   (subscribe-symbol (instance-name ?self) CLIPS_FALSE)
   (enable-symbols CLIPS_TRUE))

(definstances MSIDS
   (S02K6405Y of MSID)
   (S02K6205Y of MSID)
   (S02K6026Y of MSID)
   •••
   (S02K6078Y of MSID))
```

Creating and subscribing symbols are automatically handled by COOL. One of the only ISP implementation details that the application programmer needs to be concerned is to enable the symbols and to schedule which ISP events are to be handled.

```
(defrule connect
   ?fact<- (initial-fact)
   =>
   (retract ?fact)
   (enable-symbols CLIPS_TRUE)
   (want-isp-event LIMIT CLIPS_FALSE)
   (want-isp-event CYCLE CLIPS_TRUE))
```

Another clear advantage of using objects, like facts, is the ability to do direct pattern matching. As the ISP data changes, a low level routine updates the value in the affected slot. CLIPS could then activate any rule which needed data from the changed slot and work with this information on.

```
(defrule ValueChanges
   ?msid <- (object (is-a MSID ) (value ?value))
=>
   (update-interface
      (instance-name-to-symbol (instance-name ?msid)) ?value))
```

After working with the integration of CLIPS and ISP there is an advantage that CLIPS/COOL bring to bear on the ease of use for linking CLIPS with external ‘real time’ data. One such application that used this integrated technology was a prototype to display switch positions from on-board systems to Space Shuttle ground controllers. The prototype was up and running within three days. The display technology had already been developed as part of a training tool to help astronauts learn procedures for the Space Habitation Module. The display technology was then reused and combined with CLIPS and ISP to monitor telemetry and react whenever subscribed

data were detected. CLIPS was needed deduce single switch settings based on the downlinked telemetry data. For example, several parameters may contain measurements of pressure across a line. If most of the pressure sensors begin to fall low, then a pressure switch might have been turned off.

AN OPEN TOOLKIT APPROACH

So far, the discussion has focused on integration of CLIPS and ISP by embedding ISP into CLIPS. This has the advantage of hiding the details of the ISP client API from developers of CLIPS applications; however, it is easy to imagine applications which are no more "CLIPS applications" per se than they are "ISP clients". CLIPS and ISP provide distinct services, so it is not immediately obvious which ought to be embedded in the other, or whether embedding is even necessary.

The third approach we used to integrating CLIPS and ISP was implemented into the oimon application, discussed below. Instead of embedding ISP inside CLIPS, the two APIs are treated as "peers" within the application. Consider the following definitions:

- **open system**: a system which makes its services available through a callable API,
- **open toolkit**: an open system which permits the caller to always remain in control of execution.

The primary distinction between these two is that open systems may require that the caller turn over complete control of the application (e.g., by calling a `MainLoop()` function).

Open toolkits permit the caller to exploit the systems' functionality while maintaining control of the application; it real-time applications, this can be critical. Examples of open toolkits include the X-toolkit, the ISIS distributed communications system, and Tcl/Tk. Figure 3 depicts a number of the CLIPS and ISP API functions. Although both have functions which will take control of the application (i.e., `Run(-1)` and `ItMainLoop()`), the use of these functions is not required; lower level primitives are available to fine-tune execution of the two systems (i.e., `Run(1)` and `ItNextEvent()`, `ItDispatchEvent()`). By the definitions above, both CLIPS and ISP are open toolkits. As a result, they may both be embedded in a single application which chooses how and when to dispatch to each.

```
•  ISP:
   –     ItInitialize()
   –     ItPublish(), ItSubscribe()
   –     ItConnectServer(), ItDisconnectServer()
   –     ItNextEvent(), ItDispatchEvent()
•  CLIPS:
   –     InitializeCLIPS()
   –     AssertString()
   –     Reset()
   –     Run()
```

Figure 3. Elements of the CLIPS and ISP application program interfaces

OIMON

The Operational Instrumentation Monitor (oimon) is being developed as a MCC application run at Instrumentation/Integrated Communications Officer (INCO) console workstations. The program is useful to other non-INCO flight controllers, since it publishes (via ISP) the status of

certain electronic components which may affect the validity of data in the telemetry stream. The paragraphs which follow outline oimon and discuss how ISP and CLIPS have been integrated into it as peer open toolkits.

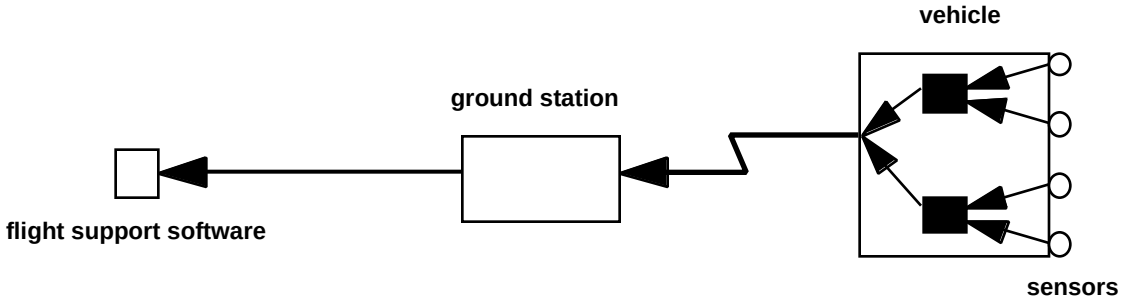


Figure 4. Shuttle sensor “channelization”

Figure 4 presents a summary of the data flow originating at sensors on board the Shuttle (e.g., temperatures, pressures, voltages, current) and ending up on a flight controller's display or in some computation used by a flight control application. The significant aspect of this figure is that there are a number of black boxes (multiplexer/demultiplexers—MDMs and discrete signal conditioners—DSCs) which sit in the data path from sensor to flight controller. A number of these black boxes (the so-called operational instrumentation (OI) MDMs and DSCs) are the responsibility of the INCO flight control discipline. There are other black boxes managed by other disciplines, for example the “flight critical” MDMs. The oimon application is concerned with the OI black boxes.

Failure of an OI MDM or DSC can corrupt the telemetry data for a number of on-board sensors. As a result, most flight controllers and many flight control applications are interested in the status of MDMs and DSCs. Instead of requiring that each consumer of telemetry data individually implement the logic necessary to assess MDM/DSC status, the INCO discipline will run oimon as an ISP client which publishes the status of the OI MDMs and DSCs. Any consumer of data affected by a particular OI black box may subscribe to its status through ISP and thus effectively defer OI MDM/DSC to the INCO discipline. This deferral of responsibility to the appropriate discipline is one of the benefits of a software architecture which promotes data sharing between different applications on different workstations.

The oimon application is a C-language program which makes calls to the ISP API to obtain its input data, the CLIPS API to execute the pattern matching necessary to infer the MDM/DSC statuses, and the ISP API to publish its conclusions. There are no explicit parameters in the downlist which unambiguously indicate OI black box status, and this is the reason CLIPS is needed. The major elements of oimon of relevance here are

- a set of CLIPS rules which implement the pattern matching,
- callback functions invoked whenever a telemetry event occurs,
- assertions of CLIPS facts from within the callback functions, and
- a main loop which coordinates CLIPS rule firing and ISP event dispatching.

A template of oimon is shown in Figure 5.

```
isp_event_callback(){
    ...
    sprintf( fact, ...);
    AssertString(fact);
    ...
}

main(){
    ...
    ItAddCallback( ...isp_events... );
    while(True){
           while( moreISP() ){
                  ItNextEvent( ... );
                  ItDispatchEvent(...);
           }

           while( moreCLIPS() ){
                  Run(1);
           }
    }
}
```

Figure 5. oimon code template

The CLIPS rule base is constructed so as to implement a number of tests currently used by INCO flight controllers to manually assess MDM/DSC status and to enable/disable some tests based on the results of others. In particular, these tests are (1) the MDM wrap test, (2) MDM and DSC built-in test equipment tests, (3) power bus assessment, and (4) a heuristic test developed by INCO flight controllers to deduce OI DSC status based on a handful of telemetry parameters which are connected to the DSCs through each of the DSC cards and channels.

The oimon application is still under development. However, preliminary experience with it suggests that the integration of CLIPS and ISP as peer toolkits called from a main application is not only feasible but easily implemented. Preliminary experience with this approach to CLIPS/ISP integration has revealed one advantage of it over embedding ISP in CLIPS. Under certain circumstances, the invocation of CLIPS functions can invoke the CLIPS “periodic action”. When CLIPS is embedded in ISP, this can cause ISP events to “interrupt” the execution of the consequent of an ISP rule. In the peer toolkit approach, ISP functionality is not invoked using the CLIPS periodic action and thus this behavior does not exist. Subsequent testing of oimon will focus on tuning the event-dispatching/rule-firing balance to ensure that oimon is neither starved of telemetry nor prevented from reasoning due to high data rates.

SUMMARY

This paper has outlined three approaches we took to integrating the CLIPS inference engine and the ISP client API into single applications. A summary of a “data layer” approach was given, but this approach was not actually implemented. A similar method was also described in which ISP API calls are embedded in CLIPS, and ISP event processing is handled as an ISP “periodic action”. The CLIPS syntax for this approach was presented. A quick prototype was developed based on this second approach, and the prototype demonstrates the soundness of the technique. In particular, it permitted very rapid development of the application. Unlike the first two approaches, the third approach we discussed did not embed ISP in CLIPS. Rather the CLIPS and ISP APIs are invoked as “peer toolkits” in the C-based oimon application. This application is currently being tested against STS-68 flight data, but additional development is expected. Preliminary results from oimon suggest that the peer toolkit approach is also sound. The possibility of ISP events interrupting the firing of ISP rules is eliminated in the oimon approach,

since ISP is invoked directly from the application instead of being called as a CLIPS periodic action.

REFERENCES

1. Paul J. Asente and Ralph R. Swick, *X Window System Toolkit*, Digital Press, 1990.

2. Joseph C. Giarratano and Gary Riley, *Expert Systems: Principles and Programming*, PWS Pub. Co.

3. Joseph C. Giarratano, *CLIPS User's Guide*, NASA JSC-25013

4. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

5. G. Riley, *CLIPS: An Expert System Building Tool*, Proceedings of the Technology 2001 Conference, San Jose, CA, December 1991.

6. *The ISIS Distributed Toolkit Version 3.0 User Reference Manual*, Isis Distributed Systems, 1992.

THE METEOROLOGICAL MONITORING SYSTEM FOR THE KENNEDY SPACE CENTER/CAPE CANAVERAL AIR STATION

Allan V. Dianic
Ensco, Inc.
445 Pineda Ct
Melbourne, FL 32940

The Kennedy Space Center (KSC) and Cape Canaveral Air Station (CCAS) are involved in many weather-sensitive operations. Manned and unmanned vehicle launches, which occur several times each year, are obvious examples of operations whose success and safety are dependent upon favorable meteorological conditions. Other operations involving NASA, Air Force and contractor personnel—including daily operations to maintain facilities, refurbish launch structures, prepare vehicles for launch and handle hazardous materials—are less publicized but are no less weather-sensitive. The Meteorological Monitoring System (MMS) is a computer network which acquires, processes, disseminates and monitors near real-time and forecast meteorological information to assist operational personnel and weather forecasters with the task of minimizing the risk to personnel, materials and the surrounding population.

The MMS acquires data from sensors located in and around the KSC/CCAS area, and subjects them to a quality-control check by an embedded CLIPS implementation. The data are then disseminated to the network of MMS Monitoring and Display Stations (MDS) which, using a second MMS implementation of CLIPS, provides the end user with a tool to monitor weather and generate warnings and alerts when weather conditions violate published criteria. The system maintains a database of operations andassociated weather criteria for the user to select. Once activated, the meteorological constraints for an operation are transformed into a series of CLIPS rules which, along with a current stream of near real-time and forecast data, are used to trigger alarms notifying the user of a potentially hazardous weather condition. Several user-defined functions have been added to CLIPS, giving it the ability to access MMS resources and alarms directly.

This paper describes the design and implementation of MMS/CLIPS-Quality Control and MMS/CLIPS-Monitoring, as well as the rule builder, the structure of rules, and the performance of the shell in a near real-time environment. Future CLIPS development issues will also be discussed.

USING PVM TO HOST CLIPS IN DISTRIBUTED ENVIRONMENTS

Leonard Myers
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA 93402
lmyers@calpoly.edu

Kym Pohl
California Polytechnic State University
San Luis Obispo, CA 93402
kpohl@calpoly.edu

ABSTRACT

It is relatively easy to enhance CLIPS to support multiple expert systems running in a distributed environment with heterogeneous machines. The task is minimized by using the PVM (Parallel Virtual Machine) code from Oak Ridge Labs to provide the distributed utility. PVM is a library of C and FORTRAN subprograms that supports distributive computing on many different UNIX platforms. A PVM deamon is easily installed on each CPU that enters the virtual machine environment. Any user with *rsh* or *rexec* access to a machine can use the one PVM deamon to obtain a generous set of distributed facilities. The ready availability of both CLIPS and PVM makes the combination of software particularly attractive for budget conscious experimentation of heterogeneous distributive computing with multiple CLIPS executables. This paper presents a design that is sufficient to provide essential message passing functions in CLIPS and enable the full range of PVM facilities.

INTRODUCTION

Distributed computing systems can provide advantages over single CPU systems in several distinct ways. They may be implemented to provide one or any combination of the following: parallelism, fault tolerance, heterogeneity, and cost effectiveness. Our interest in distributive systems is certainly for the cost effectiveness of the parallelism they provide. The speedup and economy possible with multiple inexpensive CPUs executing simultaneously make possible the applications in which we are interested, without a supercomputer host. The economy of distributed computing may be necessary for the realization of our applications, but we are even more interested in distributed systems to provide a simplicity of process scheduling and rapid interaction of low granularity domain specific actions. Our approach to implementing certain large and complex computer systems is focused on the notion of quickly providing reasoned response to user actions.

We accomplish the response by simultaneously examining the implications of an user action from many points of view. Most often these points of view are generated by individual CLIPS[1] expert systems. However, there is not necessarily a consistency in these viewpoints. Also, a domain inference may change over time, even a small period of time, as more information becomes available. Our systems have the experts immediately receive and respond to low level user action representations. The action representations encode the lowest level activities that are meaningful to the user, such as the drawing of a line in a CAD environment. Thus, if there are a dozen, or more, expert domains involved in an application, it is necessary to transmit user actions to each expert and combine the responses into a collective result that can be presented to the user. Of course, the intention is to emulate, in a relatively gross manner, the asynchronous parallel cooperative manner in which a biological brain works.

Over the course of eight years, the CAL POLY CAD Research Center and CDM Technologies, Incorporated have developed several large applications and many small experiments based essentially on the same ideas.[2, 3, 4] In the beginning we wrote our own socket code to communicate between the processes. Unfortunately, with each new brand of CPU introduced into a system it was necessary to modify the communication code. There simply are not sufficient standards in this area to provide portability of code, except by writing functions that are machine specific. Moreover, our main interest is not the support of what is essentially maintenance of the inter-process communication code. But for many years it seemed necessary to support our own communication code because the other distributed systems we investigated were simply too inefficient for our applications, even though they were generally much more robust. We then found that PVM[5] code was very compatible with our requirements.

PVM

PVM is a free package of programs and functions that support configuration, initiation, monitoring, communication, and termination of distributed UNIX processes among heterogeneous CPUs. Version 3.3 makes UDP sockets available for communication between processes on the same CPU, which is about twice as fast as the TCP sockets generally necessary for communication between CPUs. Also, shared memory communication is possible on SPARC, SGI, DEC, and IBM workstations, and several multiprocessor machines.

Any user can install the basic PVM daemon on a CPU and make it available to all other users. It does not require any root privileges. Users can execute their own console program, *pvm3*, that provides basic commands to dynamically configure the set CPUs that will make up the 'virtual machine'. The basic daemon, *pvmd*, will establish unique dynamic socket communications for each user's virtual machine. Portability is obtained through the use of multiple functions that are written specifically for each CPU that is currently supported. An environment variable is used to identify the CPU type of each daemon and hardware specific directory names, such as *SUN4*, are used to provide automatic recognition of the proper version of a PVM function. Since all of the common UNIX platforms are currently supported, the code is very portable.

The fundamental requirements for a PVM distributed system involving CLIPS processes consist of starting a CLIPS executable on each selected CPU and calling PVM communication facilities from CLIPS rulesets. The first of these tasks is very easy. PVM supports a 'spawn' command from the console program as well as a 'pvm_spawn' function that can be called from a user application program. Both allow the user to let PVM select the CPU on which each program will be executed or permit the user the specify a particular CPU or CPU architecture to host a program. Over sixty basic functions are supported for application programming in C, C++, or FORTRAN, but only the most basic will be mentioned here and only in their C interface forms.

PVM COMMUNICATION BASICS

Each PVM task is identified by its 'task id'(tid), an integer assigned by the local PVM daemon. The tid is uniquely generated within the virtual machine when either the process is spawned, or the pvm_mytid function is first called. Communication of messages between two processes in a PVM system generally consists of four actions:

1. clear the message buffer and establish a new one - **pvm_initsend**
2. pack typed information into the message buffer - **pvm_pkint,** **pvm_pkfloat,** etc.
3. send the message to a PVM task or group of tasks - **pvm_send**, **pvm_mcast,** etc.

4. receive the message, either blocking or not - **pvm_recv**, **pvm_nrecv**, etc.

Each message is required to be assigned an integer, 'msgtype', which is intended to identify its format. For instance, task1 may send a message that consists of ten integers to task2. In the 'pvm_send' used to transmit the message, the last argument is used to identify the type of message. The programmer may decide that such a message is to be of type 4, for example. Then, task2 can execute 'pvm_recv', specifying which task and message type it would like to receive. A -1 value can be used as a wildcard for either parameter in order to accept messages from any task and/or of any type. If task2 indicates that it wishes to accept only a message of type 4 to satisfy this call, it then knows when the 'pvm_recv' blocking function succeeds that it has a message of ten integers in a new input buffer.

As a more complete example, the following code shows a host task that will spawn a client task; then it will send the client a message consisting of an integer and a floating point value; and then it will wait for a reply from the client. The client waits for the message from the host, generates a string message in reply, and halts. When the client message is received, the host prints the message and halts.

Host

```
#include "pvm3.h"
main() {

int hostid, tids[1];/* pvm encodes pid within a taskid (tid) */
int ivalue1;      /* first value to be sent                  */
float fvalue2;    /* second value to be sent                 */
char message[25];/* string to be returned from client       */

hostid = pvm_mytid;          /* enroll in pvm */

pvm_spawn("Client",     /* name of process to start up  */
          (char**)0,    /* args to be passed to process */
          0,            /* options - 0 to use any CPU   */
          "",           /* host name when not option 0  */
          1,            /* number of copies to spawn    */
          tids);        /* tids of processes started    */

pvm_initsend(PvmDataRaw); /* get buffer                  */
                          /* no encoding of data if same type CPU */
pvm_pkint(23, 1, 1);      /* pack 1 int into the current buffer   */
pvm_pkfloat(45.678, 1, 1); /* pack 1 float into the current buffer */
pvm_send(tid[0], 2);      /* send buffer as user chosen type 2    */

pvm_recv(-1, 1);          /* wait for message from anyone (-1 is a */
                          /* wildcard) that is user typed as 1     */
pvm_upkstr(message);      /* unpack string from buffer into message*/

printf("I got the message: %s\n", message);

pvm_exit();
}
```

Client

```
#include "pvm3.h"
main() {

int hostid, clientid;
clientid = pvm_mytid;

pvm_recv(-1, 2);      /* wait for type 2  */
/* we ignore the contents in this example */

pvm_initsend(PvmDataRaw);
pvm_pkstr("Hi Host!");   /* pack string */
hostid = pvm_parent();   /* get host tid */
pvm_send(hostid, 1);     /* send string */
                         /* as type 1   */

pvm_exit();
}
```

Figure 1. A PVM Example

CLIPS IMPLEMENTATION CONSIDERATIONS

The basic need is to assert a fact or template from a rule in one CLIPS process into the factlist of another CLIPS process, which may be executing on a different processor. We might want a function that could use a PVM buffer to transmit a CLIPS ‘person’ template as in Figure 2:

```
(BMPutTmplt ?BufRef (person (name "Len") (hair "brown") (type “grumpy”)) )
```

Figure 2. Sample Send Template Call

The first problem this presents is that the form of this function call is illegal. It is possible to use a syntax in which the fact that is to be communicated is presented as a sequence of arguments, rather than a parenthesized list. But this syntax does not preserve the appearance of the local assert, which is pleasing to do. The solution is to add some CLIPS source code to provide a parser for this syntax.

Second, consideration must be given as to when the PVM code that receives the messages should execute. It is desirable that receiving functions can be called directly from the RHS of a CLIPS rule. It is also desirable in most of our applications that message templates are transparently asserted and deleted from the receiver’s factlist without any CLIPS rules having to fire. In order to accommodate the latter, our CLIPS shell checks for messages after the execution of every CLIPS rule, and blocks for messages in the case that the CLIPS agenda becomes empty.

Third, it is useful to be able to queue message facts to be asserted and have them locally inserted into the receiver’s factlist during the same CLIPS execution cycle. There are occasions when a number of rules might fire on a subset of the message facts that are sent. In most cases the best decision as to what rule should fire can be made if all associated message facts are received during one cycle, and all such rules are activated at the same time. Salience can usually be used to select the best of such rules and the execution of the best rule can then deactivate the other alternatives. In order to implement this third consideration, the message facts are not sent for external assertion until a special command is given.

OVERVIEW OF CMS

The CLIPS/PVM interface or CLIPS Message System (CMS), provides efficient, cost effective communication of simple and templated facts among a dynamic set of potentially distributed and heterogeneous clients. These clients may be C, C^{++}, or CLIPS processes. Clients may communicate either single fact objects or collections of facts as a single message. This communication takes place without requiring either the sender or receiver to be aware of the physical location or implementation language of the other. CMS will transmit explicit generic simple fact forms by dynamically building C representations of any combination of basic CLIPS atoms, such as INTEGERs, FLOATs, STRINGs, and MULTIFIELDs. CMS will also communicate facts or templates referenced by a fact address variable.

In addition to the dynamic generic process described above, the communication of templated facts is supported in a more static and execution time efficient manner. Specific routines capable of manipulating client templates in a direct fashion are written for each template that is unique in the number and type of its fields The deftemplate identifies what message form these routines will expect. This is a distinct luxury that eliminates the time involved in dynamically determining the number and types of fields for a generic fact that is to be communicated. The disadvantage is that the routines must be predefined to match the templates to be communicated. However, the

applications in which the authors are involved have predetermined vocabularies for the data objects that are to be communicated. In much the same way that deftemplates provide efficiency through the use of predetermined objects, these template-form specific communication routines provide object communication in the most efficient, cost-effective fashion. The following section describes the overall architecture of a system which supports the previously described functionality.

ARCHITECTURE

The underlying support environment consists of a set of utility managers called the System Manager, Memory Manager, and Error Manager. All three managers provide various functionality utilized by the communication system to perform efficient memory usage and error notification. The core of the communication system consists of five inter-related components as shown in Figure 3. These components are the Session Manager (SM), Buffer Manager (BM), Communication Object Class Library, Communication Object META Class, and CLIPS Interface Module. Collectively, these components provide the communication of objects among a dynamic set of heterogeneous clients.

[Figure Deleted]

Figure 3. Communication System Architecture

SESSION MANAGER

The Session Manager (SM) allows clients to enter and exit application sessions. Within a specific session, clients can enter and exit any number of client groups. This functionality is particularly useful in sending messages to all members of a group. The SM can also invoke PVM to spawn additional application components utilizing a variety of configuration schemes. The facilities supported range from the specification of CPUs to host the spawned applications to an automatic selection of CPUs to use as hosts and balance the load across the virtual machine.

BUFFER MANAGER

The buffer manager provides for the creation and manipulation of any number of fact buffers. Dynamic in nature, these buffers can be used to communicate several fact objects as a single atomic unit. That is, receiving CLIPS clients will accept and process the buffer's contents within a single execution cycle. Clients are free to 'put' or 'get' facts into and out of buffers throughout the life of the buffer. Information as to the contents of a buffer can be obtained by querying the buffer directly.

COMMUNICATION OBJECT CLASS LIBRARY

The Communication Object Class Library contains methods which are specific to a particular data object. (These are the routines referred to in the Overview of CMS section above.) These methods include a set of constructors and destructors along with methods to 'pack' and 'unpack' an object component-by-component. This library includes additional methods to translate an object to and from the CLIPS environment and the C Object representation used with PVM. It is this class library which may require additions if new template forms are to be communicated.

COMMUNICATION OBJECT META CLASS

The motivation of the Communication Object META Class is twofold. The META Class combines common object class functionality into a single collection of META Class methods.

That is, the highest level communication functions are exist as META Class methods, which freely accept any object defined in the Object Class Library without concern for its type. The same high-level META methods are invoked regardless of the type of objects in a communication buffer. They determine the actual type of object to be processed and then call the appropriate class method. This effectively allows clients to deal with a single set of methods independent of object type.

CLIPS INTERFACE

Essentially acting as a client to the PVM system, this collection of modules extends the functional interface of the communication system described above to CLIPS processes. This interface provides CLIPS clients with the same functionality as their C and C^{++} counterparts. In an effort to preserve the syntactical style of the CLIPS assert command, several parsers were incorporated. Since the standard CLIPS external user functions interface does not support such functionality, some additional source code to the CLIPS distribution code was required. The following section provides a brief description of the functional interface presented to CLIPS clients.

CMS FUNCTIONS

ID_ObjRef **SMConnect()**
Connects the caller to a multi-agent session.

ARGUMENTS :STRING Name
LONG InitIdServerValue
LONG IdServerRange

RETURNS : A reference to the object representation of the caller or EM_ERROR_REF. Note, this object can be passed to other clients as a way to address the caller directly.

int **SMDisconnect()**
Disconnects the caller from a multi-agent session.

ARGUMENTS : void

RETURNS :EM_SUCCESS or EM_ERROR

int **SMEnterGrp()**
Enrolls the caller into the group <Group>.

ARGUMENTS : STRING Group

RETURNS : The caller's instance number within <Group> or EM_ERROR. Instance numbers start at 0 and increment upward.

int **SMExitGrp()**
Unenrolls the caller from the group, <Group>.

ARGUMENTS : void

RETURNS : EM_SUCCESS or EM_ERROR

int **SM Spawn()**

Spawns <NumCopies> copies of <Task> in accordance with the values of <How> and <where>.

```
ARGUMENTS   :   SYMBOL      Task
                INTEGER     NumCopies
                INTEGER     How
```

<SM_DEFAULT - Target host is determined by the Session Manager. In this case <Where> should be NULL.
SM_HOST - Target host is determined by <Where>.
SM_ARCH - Target host is determined by the Session Manager. The target host's architecture will be of type <Where>.
SM_DEBUG - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will be run in the PVM debugger.
SM_TRACE - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will generate PVM trace data. >

```
                SYMBOL      Where
```

<Examples: "moby.cadrc.calpoly.edu", or SUN4>

```
                MORE ARGS   ArgV
```

<Any # of args passed to each task upon startup.>

```
RETURNS   : Actual number of spawned tasks or EM_ERROR.
```

int **SPrintIdentity()**

Prints the object referenced by NIdentityRef> to stdout.

```
ARGUMENTS   : ID_ObjRef     IdentityRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

BM_BufRef **BMCreate()**

Creates a "send" buffer which encodes its contents according to <Encoding>.

```
ARGUMENTS   :INTEGER        Encoding
```

<BM_DEFAULT - Data for heterogeneous networks
BM_RAW - No encoding takes place
BM_IN_PLACE - Same as BM_DEFAULT except data is not copied into the buffer until it is sent. >

```
RETURNS     : A reference to buffer or EM_ERROR_REF
```

int **BMDestroy()**

Destroys the buffer referenced by <BufRef>.

```
ARGUMENTS   :   BM_BufRef   BufRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

---

int **BMPutTmplt()**

Places <Tmplt> into <BufRef>.

```
ARGUMENTS   :   BM_BufRef       BufRef
                DEFTEMPLATE     Tmplt
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **BMSend()**

Sends contents of <BufRef> to <IdentityRef>. The contents of <BufRef> is left unaltered.

```
ARGUMENTS   :ID_ObjRef      IdentityRef
             BM_BufRef      BufRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

---

int **BMGrpSend()**

Sends the contents of <BufRef> to each member of <Group>.

```
ARGUMENTS   :   STRING      Group
                BM_BufRef   BufRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **BMQueryReceive()**

Formally processes incoming object groups. If no pending object groups exist, control is IMMEDIATELY returned to the caller.

```
ARGUMENTS   :   void
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **SGrpSendIdentity()**

Sends an identity to each member of a group.

```
ARGUMENTS   :   STRING      Group
                ID_ObjRef   ObjRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **BMReceive()**

Formally processes incoming object groups. If no pending object groups exist, caller is put to sleep until such an event occurs. Received objects will be placed into the caller's fact-list in their appropriate form.

```
ARGUMENTS   : void
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **SSendIdentity()**

Sends an identity to another client.

```
ARGUMENTS   :ID_ObjRef      DstIdentityRef
             ID_ObjRef      ObjRef
RETURNS     : EM_SUCCESS or EM_ERROR
```

int **SGrpSendTmplt()**	int **SSendTmplt()**
Sends a template to each member of a group.	Sends a template to another client.
ARGUMENTS : STRING Group Expression Tmplt	ARGUMENTS :ID_ObjRef DstIdentityRef Expression Tmplt
RETURNS : EM_SUCCESS or EM_ERROR	RETURNS : EM_SUCCESS or EM_ERROR

Figure 4. CMS Functions

A CMS EXAMPLE

The following example illustrates a CLIPS knowledge base intended to communicate **VALUE_FRAME** templates with CMS. The example consists of four CLIPS constructs: one deftemplate and three defrules. The first step of the example is to define a **VALUE_FRAME** template having four slots. The client registers itself in the session via the **CONNECT_ME** rule. This rule also enters the client into a group. The rule then broadcasts the client's identity to all other members of its group. Incoming identifications are processed by the **RECEIVE_IDENTITY** rule. After receiving another client's identification, several **VALUE_FRAME** facts are placed into a buffer and sent back to the client.

Incoming **VALUE_FRAME** facts are processed by the **PROCESS_VALUE_FRAME** rule. Finally, when there are no more rules on the agenda, the client goes into a blocking receive state via the execution of the **RECEIVE** rule. CLIPS clients receive facts in two distinct manners. In the first case, the communication system is queried at the end of each execution cycle for pending messages. The second method by which a client can receive messages is through an explicit call to BMReceive. The functionality of this call is identical to the implicit method, with the exception that the caller will be put to sleep until a pending message exists. In either case, incoming facts are processed transparently to the client and produce immediate modifications of the fact-list.

```
(deftemplate VALUE_FRAME
    ( slot    Frame    )
    ( slot    Instance )
    ( slot    Slot     )
    ( slot    Value    )

)


(defrule RECEIVE_IDENTITY
    ( BM_DEFAULT ?Mode )
    (    SESSION-MEMBER ?Name
         CAN-BE-REFERENCED-BY ?Identity
    )
=>
    ;Send our new friend some templates
    ( bind ?BufRef ( BMCreate ?Mode ) )

    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame1 )
                           ( Instance    Instance1 )
                           ( Slot    Slot1 )
                           ( Value  1234 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame2 )
                           ( Instance    Instance2 )
                           ( Slot    Slot2 )
                           ( Value  56.789 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame3 )
                           ( Instance    Instance3 )
                           ( Slot    Slot3 )
                           ( Value  "How are you?" )
                         )
    )
    ;Send the buffer to our new friend?
    ( BMSend ?Identity ?BufRef )

    ;Destroy the buffer
    ( BMDestroy ?BufRef )
)
```

```
(defrule CONNECT_ME
    ( initial-fact )
=>
    ( bind ?MyName "Access" )
    ( bind ?MyGrp "AgentGrp" )

    ;Connect to the session
    ( bind ?MyId ( SMConnect ?MyName 200 200) )

    ;Join a group
    ( SMEnterGrp ?MyGrp )

    ;Send my identity to members of my group
    ( SGrpSendIdentity ?MyGrp ?MyId)


    ;Assert the "receive" control fact

    ( assert( RECEIVE ) )
)


(defrule PROCESS_VALUE_FRAMES
    ?TMPLT <- ( VALUE_FRAME
                ( Frame ?Frame )
                ( Instance ?Instance )
                ( Slot ?Slot )
                ( Value ?Value )
              )
=>
    ;Process the template

        ...

    ( retract ?TMPLT )
)

(defrule RECEIVE
    (declare (salience -10000 ))
    ?REC <- ( RECEIVE )
=>
    ( BMReceive )
    ( retract ?REC )
    ( assert( RECEIVE ) )
)
```

Figure 5. A CMS Sample

CONCLUSION

PVM and CLIPS both provide free source code systems that are well maintained by developers and a sizable number of users. Relative few source code changes are necessary to either system in order to build a reliable and robust platform that will support distributed computing in a heterogeneous environment of CPUs operating under UNIX. The CMS system described in this paper provides the CLIPS interface code and some parsing code sufficient to enable efficient use of PVM facilities and communication of CLIPS facts and templates among C, C^{++}, and CLIPS

processes within a PVM virtual machine. Even more efficient communication can be obtained through enhancements to the PVM source code that can provide more efficient allocation of memory and reuse of PVM message buffers in certain applications.

NOTES

Information on PVM is best obtained by anonymous ftp from: netlib2.cs.utk.edu

Shar and tar packages are available from the same source.

The authors are currently using the CMS system in applications that involve multiple CLIPS expert systems in sophisticated interactive user interface settings. It is expected that the basic CMS code will become available in the Spring, 1995. Inquiries via e-mail are preferred.

BIBLIOGRAPHY

1. Riley, Gary, B. Donnell et. al., "CLIPS Reference Manual," JSC-25012, Lyndon B. Johnson Space Center, Houston, Texas, June, 1993.

2. Pohl, Jens, A. Chapman, L.Chirica, R. Howell, and L. Myers, "Implementation Strategies for a Prototype ICADS Working Model," CADRU-02-88, CAD Research Unit, Design Institute, School of Architecture and Design, Cal Poly, San Luis Obispo, California, June, 1988.

3. Myers, Leonard and J. Pohl, "ICADS Expert Design Advisor: An Aid to Reflective Thinking," Knowledge-Based Systems, London, Vol. 5, No. 1, March, 1992, pp. 41-54.

4. Pohl, Jens and L. Myers, "A Distributed Cooperative Model for Architectural Design," Automation In Construction, Amsterdam, 3, 1994, pp. 177-185.

6. Geist, Al, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge. Tennessee, May, 1993.

A PARALLEL STRATEGY FOR IMPLEMENTING REAL-TIME EXPERT SYSTEMS USING CLIPS

Laszlo A. Ilyes & F. Eugenio Villaseca
Cleveland State University
Dept. of Electrical Engineering
1983 East 24th Street
Cleveland, Ohio 44115

John DeLaat
NASA Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135

ABSTRACT

As evidenced by current literature, there appears to be a continued interest in the study of real-time expert systems. It is generally recognized that speed of execution is only one consideration when designing an effective real-time expert system. Some other features one must consider are the expert system's ability to perform temporal reasoning, handle interrupts, prioritize data, contend with data uncertainty, and perform context focusing as dictated by the incoming data to the expert system.

This paper presents a strategy for implementing a real time expert system on the iPSC/860 hypercube parallel computer using CLIPS. The strategy takes into consideration, not only the execution time of the software, but also those features which define a true real-time expert system. The methodology is then demonstrated using a practical implementation of an expert system which performs diagnostics on the Space Shuttle Main Engine (SSME).

This particular implementation uses an eight node hypercube to process ten sensor measurements in order to simultaneously diagnose five different failure modes within the SSME. The main program is written in ANSI C and embeds CLIPS to better facilitate and debug the rule based expert system.

INTRODUCTION

Strictly defined, an expert system is a computer program which imitates the functions of a human expert in a particular field [1]. An expert system may be described as a real-time expert system if it can respond to user inputs within some reasonable span of time during which input data remains valid. A vast body of recently published research clearly indicates an active interest in the area of real-time expert systems [2-12].

Science and engineering objectives for future NASA missions require an increased level of autonomy for both onboard and ground based systems due to the extraordinary quantities of information to be processed as well as the long transmission delays inherent to space missions. [13]. An expert system for REusable Rocket Engine Diagnostics Systems (REREDS) has been investigated by NASA Lewis Research Center [14, 15, 16]. Sequential implementations of the expert system have been found to be too slow to analyze data for practical implementation. As implemented sequentially, REREDS already exhibits a certain degree of inherent parallelism. Ten sensor measurements are used to diagnose the presence of five different failures which may manifest themselves in the working SSME. Each module of code which diagnoses one failure is referred to as a failure detector. While some of the sensor measurements are shared between

failure detectors, the computations within these detectors are completely independent of one other.

One apparent way to partition the problem of detecting failures in the SSME, is to assign each failure detector to its own node on the hypercube system. Because the failure detectors may be processed simultaneously, a speedup in the execution is expected. But while execution time is a critical parameter in any real-time expert system, it is not the only ingredient required in order to guarantee its success. A recent report characterized the features required of expert systems to operate in real-time. In addition to the requirement of fast execution, the real-time expert system should also possess the ability to perform context focusing, interrupt handling, temporal reasoning, uncertainty handling, and truth maintenance. Furthermore, the computational time required by the system should be predictable and the expert system should potentially be able to communicate with other expert systems [17]. These aspects are considered in the design presented in this paper.

METHODS

The rules for diagnosing failures in the SSME were elicited from NASA engineers and translated into an off-line implementation of a REREDS expert system [18]. While some of the failures can be diagnosed using only sensor measurements, other failures require both data measurements and the results obtained from condition monitors. The condition monitors measure both angular velocity and acceleration on various bearings of the High Pressure Oxidizer Turbo-Pump (HPOTP) shaft and determine the magnitudes of various torsional modes in the HPOTP shaft [19]. Due to the lack of availability of high frequency bearing data and additional hardware requirements for implementing real-time condition monitors, this expert system considered only those failure detectors which required sensor measurements alone.

The five failure detectors which rely solely on sensor measurements for diagnosis are listed in Table 1 along with a description of the failure, the required sensor measurements, and their respective, relative failure states. Notice that failure detectors designated F11 and F15 cannot be differentiated from one another and are thus combined into one single failure mode.

For each sensor measurement listed, the expert system knowledge base is programmed with a set of nominal values and deviation values (designated in our work by σ). One of the roles of the expert system is to match incoming sensor measurements with the nominal and deviation values which correspond to the specific power level of the SSME at any given time. Any sensor measurement may deviate from the nominal value by $\pm \sigma$ without being considered high or low relative to nominal. Beyond the σ deviation, the sensor measurement is rated with a value which is linearly dependent upon the amount of deviation. This value is referred to as a vote and is used by a failure detector to determine a confidence level that the failure mode is present. This voting curve is illustrated in Figure 1.

Once a vote has been assigned to every sensor measurement, each failure detector averages the votes for all of its corresponding sensor measurements. The final result will be a number between -1.00 and +1.00. This result is converted to a percent and is referred to as the corresponding confidence level of that failure mode. The underlying motivation for this approach is to add inherent uncertainty handling to the expert system.

Failure Detector	Description	Measurements	Failure States
F11/15	Labyrinth/Turbine Seal Leak	LPFP Discharge Pressure FPOV Valve Position HPFTP Turbine Discharge Temp.	Low High High
F67	HPOTP Turbine Interstage & Tip Seal Wear	HPOTP Discharge Temperature HPOP Discharge Pressure HPOTP Shaft Speed MCC Pressure	Low Low Low Low
F68	Intermediate Seal Wear	Secondary Seal Drain Temperature HPOTP Inter-Seal Drain Pressure	Low Low
F69	HPOP Primary Seal Wear	HPOP Primary Seal Drain Pressure Secondary Seal Drain Pressure Secondary Seal Drain Temperature	High High High
F70	Pump Cavitation	HPOP Discharge Pressure HPOTP Shaft Speed MCC Pressure	Low Low Low

Table 1. Failure Detectors Only Requiring Sensor Measurements for Failure Diagnosis

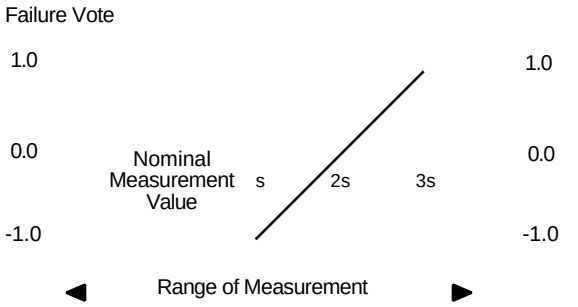


Figure 1. Voting Curve for Sensor Measurement with "High" Failure State

Each individual failure detector was implemented in CLIPS on a personal computer and its accuracy was tested and verified using simulated SSME sensor data. Once satisfactory results were achieved, an ANSI C program was written for the iPSC/860 hypercube computer which would initialize the CLIPS environment on five nodes of a 2^3 hypercube structure. These five nodes, referred to as the *failure detector nodes*, load the constructs for one failure detector each, and use CLIPS as an embedded application as described in the CLIPS Advanced Programming Guide [20]. In this way, CLIPS will only be used for evaluation of the REREDS rules. All other programming requirements, including opening and closing of sensor measurement data files, preliminary data analysis, and program flow control are handled in C language. By embedding the CLIPS modules within ANSI C code, context focusing and process interruptions can be more efficiently realized.

Coordination of data acquisition and distribution among the failure detector nodes is accomplished through a *server node* which is programmed to furnish sensor measurement data to requesting nodes. Since the data for this study originate from the SSME simulator test bed, data retrieval is accomplished simply by reading sequential data from prepared data files. The server node transfers incoming sensor measurements into an indexed memory array, or blackboard, from which data are distributed upon request to the failure detector nodes. When the blackboard

is updated, all requests for data are ignored until data transfer is completed. This assures that reasoning within the expert system is always performed on contemporaneous data. The server node does not invoke the CLIPS environment at any time. It is programmed entirely in C language code.

One additional node, referred to as the *manager node*, is used by the expert system to coordinate the timing between the failure detector nodes and the server node. Like the server node, the manager node does not invoke the CLIPS environment. Once the manager node has received a "ready" message from all failure detector nodes, it orders the server node to refresh the data in the blackboard. During this refresh, the failure detector nodes save their results to permanent storage on the system. The activities and process flow of all three types of nodes used in this research are illustrated in Figure 2. The asterisk denotes the point at which all nodes synchronize.

CONCLUSION

Profiling studies were conducted on the parallel implementation of the REREDS expert system. It was found that the system could process the sensor measurements and report confidence levels for all five failure modes in 18 milliseconds. A sequential implementation of the expert system on the same hardware was found to require over 50 milliseconds to process and report the same information, indicating that the parallel implementation can process data at nearly three times as quickly. Considering the fact that seven processors are being used in the parallel implementation, these results may seem somewhat disappointing, however, the profiling studies also indicate that additional speedup can be realized in future implementations of this expert system if the data blackboard is also parallelized. Using only one server node causes some hardware contention. Shortly after the nodes synchronize, the failure detectors tend to overwhelm the server with five (nearly) simultaneous data requests. By adding a second server node to the system, this contention can be greatly reduced.

Since the data can be processed at a fast, continuous rate, the validity of sensor measurements can be assured during processing. Consequently, truth maintenance is realized by suppressing data requests to the server node until all sensor measurements have been simultaneously updated. This guarantees that all data accessed by the failure detector nodes during any processing cycle is the same "age."

Due to the nature of the particular expert system selected for this research, the time required by the failure detectors to process SSME data remains constant regardless of whether or not a failure condition exists. Thus, predictability is always assured for this example. Also, the need for temporal reasoning is not explicitly indicated and is therefore not investigated. Since these aspects of the design are application specific, they and must be investigated in future work using different expert system models.

As discussed earlier, uncertainty handling is inherent to this expert system. The voting scheme and use of confidence levels permits reasoning, even in the presence of noisy, incomplete, or inconsistent data. Since the output from the system is a graded value rather than a binary value, the output carries with it additional information about the expert system's confidence that a particular failure is occurring.

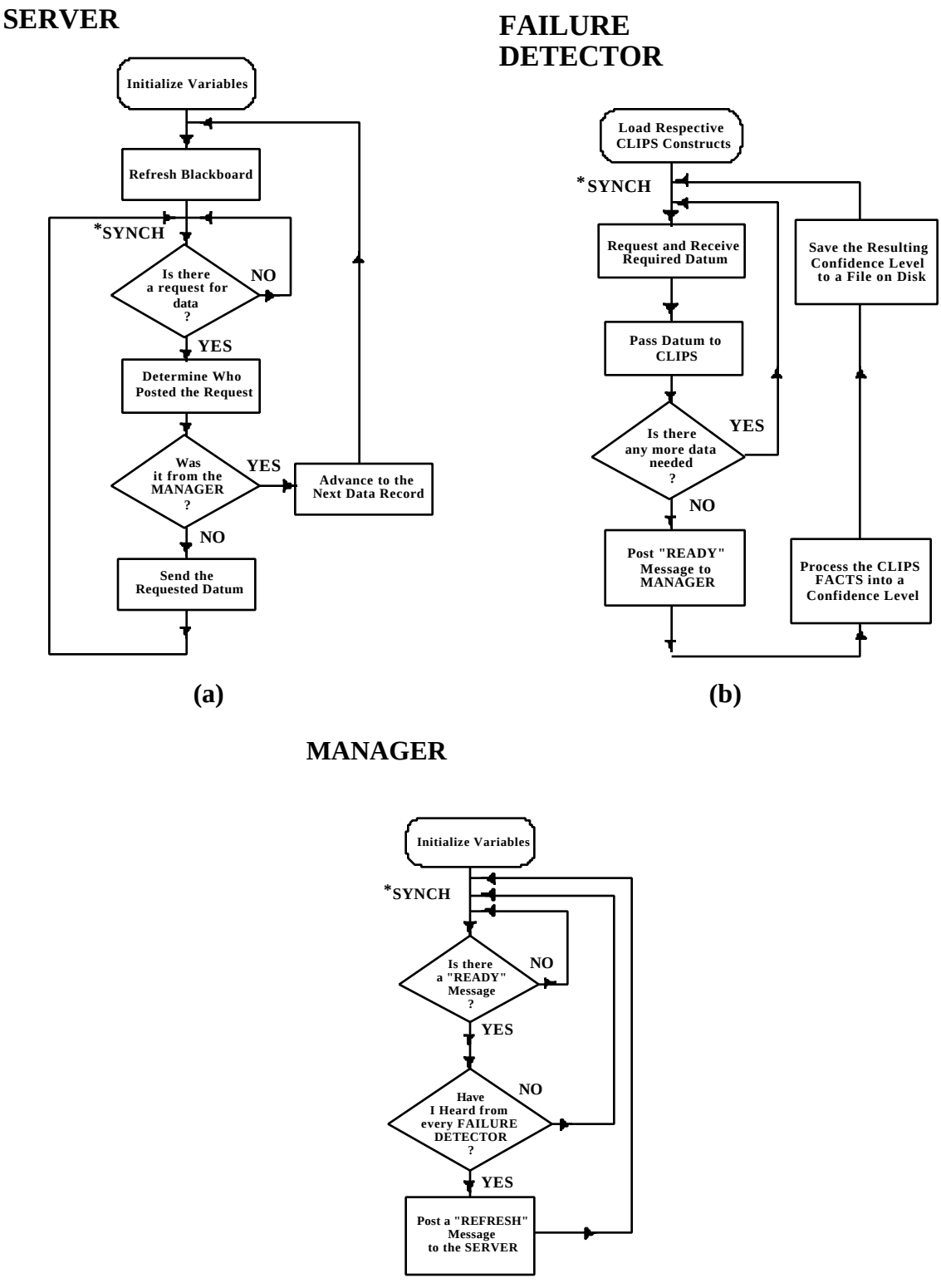


Figure 2. Process flow for the a) Server Node,
b) Failure Detector Nodes, and c) Manager Node

One of the most important features of this design is that program flow control and system I/O is accomplished in C language code. Using CLIPS as an embedded application within a fast, compiled body of C language code allows the expert system to be more easily integrated into a practical production system. Complex reasoning can be relegated directly and exclusively to the nodes invoking the CLIPS environment, while tasks which are better suited to C language code can be performed by the server and manager nodes. Thus, simple decisions can be realized quickly in C language rather than relying on the slower CLIPS environment. Based on fast preprocessing of the sensor measurements, the C language code can be used to initiate process interrupts during emergency conditions and even change the context focusing of the expert system. Those tasks which require complex reasoning can be developed and refined separately in CLIPS, taking full advantage of the debugging tools available in the CLIPS development environment.

While the rules for this particular expert system are somewhat simple compared to other applications considered in the literature, it is believed that the approach used in this study can be extended to other examples. This study demonstrates that parallel processing can not only speed up the execution of certain expert systems, but also incorporate other important features essential for real-time operation.

ACKNOWLEDGMENTS

We would like to thank NASA Lewis Research Center for its financial support of this research under cooperative agreement NCC-257 (supplement no. 3). We would also like to extend our thanks to the technical support staff of the ACCL and the internal reviewers at NASA Lewis Research Center, as well as Mr. Gary Riley of NASA Johnson Space Center. It is only with their patient and supportive contributions that this research was possible.

REFERENCES

1. Lugger, George and Stubblefield, William, "AI: History and Applications," *Artificial Intelligence and the Design of Expert Systems*, The Benjamin/Cummings Publishing Co. Inc., New York, NY, 1989, pp. 16-19.

2. Leitch, R., "A Real-Time Knowledge Based System for Process Control," *IEE Proceedings, Part D: Control, Theory and Applications*, Vol. 138, No. 3, May, 1991, pp. 217-227.

3. Koyama, K.,"Real-Time Expert System for Gas Plant Operation Support (GAPOS)," *16th Annual Conference of IEEE Industrial Electronics*, November, 1990.

4. Bahr, E., Barachini, F., Dopplebauer, J. Grabner, H. Kasperic, F., Mandl, T., and Mistleberger, H., "Execution of Real-Time Expert Systems on a Multicomputer," *16th Annual Conference of IEEE Industrial Electronics,*, November, 1990.

5. Wang, C. Mok, A., and Cheng, A., "A Real-Time Rule Based Production System," *Proceedings of the 11th Real-Time Systems Symposium*, December, 1990.

6. Moore, R., Rosenhof, H., and Stanley, G., "Process Control Using Real-Time Expert System," *Proceedings of the 11th Triennial World Congress of the International Federation of Automatic Control*, August, 1990.

7. Lee, Y., Zhang, L., and Cameron, R., "ESC: An Expert Switching Control System," *International Conference on Control*, March, 1991.

8. Jones, A., Porter, B., Fripp, R., and Pallet, S., "Real-Time Expert System for Diagnostics and Closed Loop Control," *Proceedings of the 5th Annual IEEE International Symposium on Intelligent Control*, September, 1990.

9. Borsje, H., Finn, G., and Christian, J., "Real-Time Expert Systems and Data Reconciliation for Process Applications," *Proceedings of the ISA 1990 International Conference and Exhibition*, Part 4 of 4, October, 1990.

10. Silagi, R. and Friedman, P., "Telemetry Ground Station Data Servers for Real-Time Expert Systems," *International Telemetering Conference*, ITC/USA, October, 1990.

11. Spinrad, M., "Facilities for Closed-Loop Control in Real-Time Expert Systems for Industrial Automation," *Proceedings of the ISA/1989 International Conference and Exhibit*, Part 2 of 4, October, 1990.

12. Hota, K., Nomura, H., Takemoto, H., Suzuki, K., Nakamura, S., and Fukui, S., "Implementation of a Real-Time Expert System for a Restoration Guide in a Dispatching Center," *IEEE Transactions on Power Systems*, Vol. 5, No. 3, 1990, pp. 1032-8.

13. Lau, S. and Yan, J., "Parallel Processing and Expert Systems," NASA Technical Memorandum, No. 103886, May, 1991.

14. Merrill, Walter and Lorenzo, Carl, "A Reusable Rocket Engine Intelligent Control," 24th Joint Propulsion Conference, Cosponsored by AIAA, ASME, and ASEE, Boston MA, July 1988.

15. Guo T.-H. and Merrill, W., "A Framework for Real-Time Engine Diagnostics," 1990 Conference for Advanced Earth-to-Orbit Propulsion Technology," Marshall Space Flight Center, May 1990.

16. Guo, T.-H., Merrill, W. , and Duyar, A., "Real-Time Diagnostics for a Reusable Rocket Engine," NASA Technical Memorandum No. 105792, August 1992.

17. Kreidler, D. and Vickers, D., "Distributed Systems Status and Control," Final Report to NASA Johnson Space Center, NASA CR 187014, September 1990.

18. Anex, Robert, "Reusable Rocket Engine Diagnostic System Design,"Final Report, NASA CR 191146, January 1993.

19. Randall, M.R., Barkadourian, S., Collins, J.J., and Martinez, C., "Condition Monitoring Instrumentation for Space Vehicle Propulsion Systems," NASA CP 3012, pp. 562-569, May 1988.

20. CLIPS 6.0 User's Manual, Volume II, Advanced Programming Guide, pp. 43-142, June, 1993.

USING CLIPS IN THE DOMAIN OF KNOWLEDGE-BASED MASSIVELY PARALLEL PROGRAMMING

Jiri J. Dvorak
Section of Research and Development
Swiss Scientific Computing Center CSCS
Via Cantonale, CH-6928 Manno, Switzerland
Email: dvorak@cscs.ch

ABSTRACT

The Program Development Environment PDE is a tool for massively parallel programming of distributed-memory architectures. Adopting a knowledge-based approach, the PDE eliminates the complexity introduced by parallel hardware with distributed memory and offers complete transparency in respect of parallelism exploitation. The knowledge-based part of the PDE is realized in CLIPS. Its principal task is to find an efficient parallel realization of the application specified by the user in a comfortable, abstract, domain-oriented formalism. A large collection of fine-grain parallel algorithmic skeletons, represented as COOL objects in a tree hierarchy, contains the algorithmic knowledge. A hybrid knowledge base with rule modules and procedural parts, encoding expertise about application domain, parallel programming, software engineering, and parallel hardware, enables a high degree of automation in the software development process.

In this paper, important aspects of the implementation of the PDE using CLIPS and COOL are shown, including the embedding of CLIPS with C++-based parts of the PDE. The appropriateness of the chosen approach and of the CLIPS language for knowledge-based software engineering are discussed.

INTRODUCTION

Massive parallelism is expected to provide the next substantial increase in computing power needed for current and future scientific applications. Whereas there exists a variety of hardware platforms offering massive parallelism, a comfortable programming environ- ment making programming of distributed-memory architectures as easy as programming single address space systems is still missing. The programmer of parallel hardware is typically confronted with aspects not found on conventional architectures, such as data decomposition, data and load distribution, data communication, process coordination, varying data access delays, and processor topology. Approaches to a simplification of parallel programming that have been used previously, in particular for shared-memory architectures, are transformational and compile-time parallelization [9, 10]. However, such code-level parallelization is extremely difficult for distributed architectures. Additionally, only a limited, fine-grain part of the opportunities for parallel execution within the program can be detected, and the results are dependent on the programming style of the programmer. The detection of conceptual, coarse-grain parallelism found in many applications from natural sciences and engineering is in general too complicated for current parallelization techniques.

Our Program Development Environment PDE [4, 6] is a tool for programming massively parallel computer systems with distributed memory. Its primary goal is to handle all complexity introduced by the parallel hardware in a user-transparent way. To break the current limitations in code parallelization, we approach parallel program development with knowledge-based techniques and with user support starting at an earlier phase in program development, at the specification and design phase.

In this paper we focus on the expert system part of the PDE realized in the CLIPS [7] language. At first, a short overview of the PDE is given. The representation and use of knowledge is the topic of section 3. This is followed by sections showing aspects of external interfaces, problem representation, and embedding with C++. The important role of an object-oriented approach will be elaborated in these parts. A discussion of results, the appropriateness of the CLIPS language, and the current state of the development concludes the paper.

OVERVIEW OF THE PDE

The basic methodology of programming with the PDE consists of three steps [5]:

1. Problem specification using a domain-specific, high-level formalism
2. Interactive refinement and completion of the specification (if needed)
3. User-transparent generation of compilable program code

According to this three-step approach to the programming process, the program development environment_consists of the three functional components shown in Fig. 1. In a typical session, the programmer gives an initial specification of the problem under consideration using a programming assistant interface. Currently, we have an interface SMPAI covering the domain of stencil-based applications on n-dimensional grids. Other interfaces are in preparation. The initial problem specification is decomposed into the purely computational features and the features relevant for the parallel structure. The first are passed directly to the Program Synthesizer PS, the latter go to the Programming Assistant PA. Then, in interaction with the user, the PA extracts and completes the information needed to determine an appropriate parallel framework. The PA is the central, largely AI-based component of the PDE, relying on various kinds of expert knowledge. The program synthesizer expands the parallel framework into compilable, hardware specific, parallel C++ or C programs.

REPRESENTATION OF ALGORITHMIC KNOWLEDGE

The PDE embodies a skeleton-oriented approach [3] to knowledge-based software engineering. A large collection of hierarchically organized fine-grain skeletons forms the algorithmic knowledge. The hierarchy spans from a general, abstract skeleton at the root to highly problem-specific, optimized skeletons at the leaf-level. Skeleton nodes in the tree represent a number of different entities. First, every non-leaf skeleton is a decision point for the rule-based descent. The descendants are specializations of the actual node in one particular aspect, the discrimination criterion of the skeleton. Every skeleton node contains information about the discrimination criterion and value that leads to its selection among the descendants of its parent node. The discrimination criteria represent concepts from parallel programming, application domain, and characteristics of the supported parallel hardware architectures. Second, a skeleton node holds information about requirements other than the discrimination criterion that have to be checked before a descent is done. Third, skeleton nodes have slots for other kind of information that is not checked during descent but that may be helpful for the user or for subsequent steps. Finally, the skeleton nodes have attached methods or message handlers that are able to dynamically generate the parallel framework for the problem to be solved with the chosen skeleton.

Based on these different roles, the question arises whether the skeleton knowledge base should be built using instance or class objects. To access information in slots, a skeleton node should be an instance of a suitable class. However, to attach specialized methods or message-handlers at any node, skeletons have to be represented using classes. Also, the specialization type of the hierarchy suggests the use of a class hierarchy for the skeleton tree. We have solved the conflict by using both a class and an instance as the representation of a single skeleton node. A simplified example skeleton class is:

```
(defclass skel-32 (is-a init-bnd-mixin skel-5)
   (pattern-match reactive)
   (slot discrimination-crit
      (default problem-type)
      (create-accessor read))
   (slot discrimination-value
      (default INIT_BND_VAL_PROBLEM)
      (create-accessor read))
   (multislot constraints
      (create-accessor read-write))
   (multislot required
      (default (create$ gridinstances_requirement))
      (create-accessor read)))
```

The required slot holds instances of the requirements mentioned above. Constraints are an additional concept used for describing restrictions in application scope.

All instances of the skeleton nodes are generated automatically with a simple function:

```
(deffunction create-instances (?rootclass)
   (progn$
      (?node (class-subclasses ?rootclass inherit))
      (make-instance ?node of ?node)))
```

The two basic operations on the skeleton tree are the finding of the optimal skeleton, i.e., the descent in the tree, and the generation of the computational framework based on a selected node.

Rule-based skeleton tree descent

The descent in the skeleton tree is driven by rules. Based on the discrimination criterion of the current node, rules match on particular characteristics of the application specification and try to derive a value for the criterion. If a descendant node with the derived criterion value exists, it will get the candidate for the descent. The following example rule tries to find a descendant matching the grid coloring property of the specification:

```
(defrule DESC::coloring-1
   ?obj <- (object (name [descent_object]) (curr_skel ?skel))
   ?inst <- (object (discrimination-crit coloring)
   (discrimination-value ?c))
   (test (member$ (class ?inst) (class-subclasses (class ?skel))))
   ?res <- (object (nr_of_colors ?c))
   ?app <- (object (is-a PAapp_class) (results ?res))
   =>
   (send ?obj put-next_skel ?inst)
   (send ?obj put-decision_ok t))
```

The pattern matching relies heavily on object patterns. The whole expert system is programmed practically without the use of facts. The rule first matches a descent object, then a skeleton which is a descendant of the current skeleton and finally the nr_of_colors slot of the problem specification. Note that the particular ordering of object patterns is induced by some peculiarities of the CLIPS 6.0 pattern matching procedure. In order to handle references to objects in other modules, it is advisable to use instance addresses instead of instance names. CLIPS does not match objects from imported modules when they are refered to by a name property and the module name is not explicitly given. On the other hand, CLIPS has no instance-address property for object patterns and the form ?obj <- (object ...) does not allow ?obj to be a bound variable.

So, the ordering of patterns is restricted in such a way that first an object gets matched and then it is verified whether it is the desired one. The consequence is probably some loss in pattern matching efficiency.

Although our initial problem domain is sufficiently restricted and well-defined to allow a complete, automatic descent in most cases, there is a collection of user interaction tools, called the Intelligent Skeleton Programming Environment ISPE, ready to handle the case when the descent stops before reaching a sufficiently elaborated skeleton. With the ISPE, the user can add missing information, select a descendant node manually, or even step through the tree guided by the expert system. For such a functionality, the ISPE needs a high integration with the expert system. The rules for the descent are divided into various modules, separating search for descendants, verification and actual descent in different rule modules. The object of class descent_class used in the rule above serves to keep track of the current state of descent and to coordinate between the rule modules.

Generating output from the skeleton tree

After successful selection of a skeleton node for the application under consideration, the parallel framework can be generated. This is done by calling message-handlers attached to all skeletons. An example parallel framework for a simple stencil problem is:

```
init(Grid);
   FOR (iter = 0; iter < nr_of_iterations; iter++) DO
      FOR (color = 0; color < 3; color++) DO
         INTERACTION
            fill_buf(Grid, w_obuf, west, color);
            Exchange(e_ibuf, east, w_obuf, west);
            scatter_buf(Grid, e_ibuf, east, color);
         END;
         CALCULATION
            update(Grid, color);
         END;
      ENDFOR;
   ENDFOR;
finish(Grid);
```

The building blocks of this formalism [2] are communication and computation procedure calls, grouped by sequencing or iteration statements.

Object-oriented concepts are realized for optimizing the representational efficiency in the skeleton tree. First, inheritance insures that information stored in the slots of the skeleton nodes is passed down the tree. If from a certain point on it does not apply any more, it can be overriden. Second, new behaviour is introduced with mixin classes. As it can be seen in the skel-32 node shown earlier, each node has both a mixin class and the parent class in its superclass list. In this way, it is possible to define a particular feature only once, but to add it at various points in the tree. Finally, instead of having methods that for each skeleton individually generate the complete parallel framework, an incremental method combination approach has been chosen. Every method first calls the same method of its parent node and then makes its own additions. The message-handler below shows this basic structure:

```
(defmessage-handler stencil-MS-mixin generate-MS ()
   (bind ?inst (call-next-handler))
   (send ?inst put-global_vars (create$ .... )))
```

Message-handlers instead of methods have to be used for this kind of incremental structure, as methods in CLIPS do not have dynamic precedence computation, whereas message-handlers do. Using methods, the sequence of skeleton and method definitions in the source file would be dominant for the precedence instead of the class hierarchy at runtime.

INTERFACES AND PROBLEM REPRESENTATIONS

The expert system of the PDE has basically three interfaces to the outside. It receives input from the problem specification tools, generates the parallel framework as output for the program synthesizer component, and it has an interface for user interaction.

Input

In order to omit a parser written in CLIPS, the formalism for the problem specification was chosen to be directly readable into CLIPS. A natural way to achieve this consists in using instance-generating constructs. We have two types of constructs in the input formalism, for example:

```
(make-instance global_spec of global_spec_class
   (grid_const 1.0)
   (dimension 2)
   (problem_type BND_VAL_PROBLEM))

(stencil_spec
   (assign_op
      (grid_var f (coord 0 0))
      (function_call median
         (grid_var f (coord -1 0))
         (grid_var f (coord 0 0))
         (grid_var f (coord 1 0)))))
```

In the first statement above, where global properties of the application are defined, the make-instance is part of the formalism. Obviously, reading such a construct from a file with the CLIPS batch command generates an instance of the respective class and initializes the slots. In the second case, where the stencil is defined, the instance-generating ability is not directly visible. However, for every keyword such as stencil_spec, assign_op or grid_var, there are functions creating instances of respective classes, e.g.:

```
(deffunction grid_var (?name ?coord)
   (bind ?inst (make-instance (gensym*) of grid_var))
   (send ?inst put-var_name ?name)
   (send ?inst put-coord ?coord))
```

The reasons to not use make-instance in all cases are that some constructs can have multiple entities of the same name, e.g., a stencil can have multiple assignment operations, and additional processing that is needed by some constructs.

The lack of nested lists in CLIPS was considered as a disadvantage at the beginning, in particular regarding the close relation between CLIPS and Lisp/CLOS. However, a recursive list construct can be easily defined with COOL objects and instance-generation functions similar to the one above. Certainly, this is not appropriate if runtime efficiency is critical. In the course of the PDE development, the lack of lists proved to have a positive effect on style and readability. For example, instead of using just the list (-1 1 2) for a 3-D coordinate, using the construct (coord -1 1 2) creates an instance of a specialized class for coordinate tuples. Such an instance can be easier handled than a list and appropriate methods or message-handlers can be defined. The

whole problem representation after reading the specification input is present in a number of nested, interconnected instances of respective classes.

Output

The result of the generation of the parallel framework is a problem representation by means of a number of instances, much in the sense of the input representation shown above. The writing of an output file as shown in section 3.2 is done with message-handlers attached to each of the relevant problem representation classes, e.g., a for-loop is written by:

```
(defmessage-handler for_loop_class write-out (?stream)
   (printout ?stream
         "  FOR (" ?self:varname " = " ?self:from ";  "
         ?self:varname " < " ?self:till
         ";  " ?self:varname "++) DO " t)
   (progn$ (?el ?self:subs)
      (send ?el write-out ?stream))
      (printout ?stream "  ENDFOR;" t))
```

EMBEDDING WITH C++

The global structure of the PDE consists of components written in C++, among them the main program and the graphical user interface, and an embedded CLIPS expert system for knowledge representation and reasoning. Additionally, two parsers use the Lex/Yacc utilities. Whereas some components, such as the parsers, are integrated only by means of intermediate files for input and output, the expert system is highly integrated with the C++-based ISPE and the graphical user interface. The CLIPS dialog itself is visible to the user through a CLIPS base window. Figure 2 shows both the CLIPS dialog window and a browser window for graphically browsing the skeleton tree.

Data integration

Both CLIPS and C++ offer objects for the representation of data. It is therefore a straightforward decision to use the object mechanism for the data integration between an expert system written in CLIPS and programs written in C++. The concept for the CLIPS-C++ integration relies on the decisions to represent common data on the CLIPS side using COOL objects and to provide wrapper classes on the C++ side for a transparent access to COOL objects. A class hierarchy has been built in C++ to represent CLIPS types, including classes and instances. Access to COOL classes is needed for example in the skeleton tree browser, where descendants of a node are only found by inspecting the subclasses list of the node.

The C++ wrapper classes consist of an abstract class clips_type with subclasses for the CLIPS data types integer, float, symbol, string, multifield, and for COOL classes and instances. The class coolframe used to represent COOL instances is shown below:

```
class coolframe : public clips_type
      char* framename;
   public:
      coolframe(char* name);
      clips_type* get_slot(char* slotname);
      int put_slot(char* slotname, clips_type* value);
      char* class_of(); ";
```

The creation of a C++ frontend to a COOL instance is performed by just instantiating the above class, giving the constructor the name of the COOL instance. Accesses to slots have to be done

by using the get_slot and put_slot member functions that refer to functions in the CLIPS external interface [8]. The class coolframe is a generic class, usable for any COOL instance, no matter what collection of slots the COOL instance has. A more sophisticated approach with separate members for all slots would be somewhat more convenient, as the distinction between accessing a C++ object and accessing a COOL object through the C++ wrapper would be completely blurred. However, defining classes on the fly, based on the structure of the COOL instances, is not possible in C++. The chosen system with general-purpose wrapper classes is already convenient for use in C++. Operators and functions in C++ can be overloaded to handle CLIPS data transparently. For example, the basic mathematical operations can be overloaded to combine C++ numbers with CLIPS numbers in one expression.

Care has to be taken to not introduce inconsistencies between the data stored in CLIPS and the C++ wrappers. To this end, the C++ interface to CLIPS does not cache any information, and before performing any access through the CLIPS external interface it is verified whether the COOL object to be accessed still exists.

Functional integration

The goal with functional integration is to achieve fine-grain control over the reasoning in the expert system from C++-based components whenever needed. For example, it is sometimes desirable to check whether one single descent from the current skeleton node is possible. Or, the user may prefer to step manually through the tree, getting support from the expert system. The detailed control needed for such tasks has been achieved with a partitioning of the rules into a number of rule modules. Then, the C++ component can set the focus to just the rule system desired and start the reasoning.

Two basic means to interact from the C++ components to the CLIPS expert system exist in the PDE. First, a C++ program can call any of the C functions from the CLIPS external interface. This happens without user visibility in the CLIPS dialog window, and a return value can be obtained from the call. Second, thanks to the graphical user interface written in C++, the C++ program can directly write to the dialog window in such a way that CLIPS thinks it received input at the command line. Using this alternative, no return value can be passed back to the C++ component, but the interaction is visible to the user in the window. It is thus most suited to starting the reasoning or other functions that produce output or a trace in the dialog window.

CONCLUSIONS

The realization of the parallel program development environment PDE has successfully achieved the primary goal of making parallel programming as simple as sequential programming within the initial problem domain of stencil-based applications. Moreover, thanks to programming environment support spanning from the design level up to automated code generation and thanks to the reuse of important software components, parallel programming with the PDE can be considered substantially simpler and more reliable than sequential programming in a common procedural language. Apart from the high-level, domain-oriented approach to programming, the PDE offers to the user efficiency preserving portability of software across platforms, reuse of critical software components, and a flexible and comfortable interface.

With a focus on the parts realized using CLIPS, various aspects of the implementation of our knowledge-based parallel program development tool PDE have been shown in this paper. CLIPS in its current version 6.0 [8] proved to have some critical properties making it particularly well-suited for the use within the PDE. Of highest importance are probably the object-oriented capabilities of CLIPS, enabling flexible interfaces to the outside, appropriate representations of knowledge and intermediate problem states, and, together with the CLIPS external interface, a

convenient embedding with the C++ components of the PDE. CLIPS is well suited for a rapid prototyping approach to system development, in particular due to the flexibility that the object-oriented mechanism can offer. The PDE development is currently in the fourth prototype. Apart from the first throw-away prototype done in CLOS [1], each prototype reuses large parts of the previous one, adding completely new components or new functionality.

The problems with the CLIPS language encountered during the PDE development relate in part to the resemblance of CLIPS to CLOS. Examples are the lack of the lists in the sense of Lisp, the static precedence determination for methods, or the inability to pass methods or functions as first-class objects. But alternatives offering similar functionality have been found in all cases. Apart from such CLOS-like items, a suggestion for improvement of the CLIPS language based on our experience is to focus more on object patterns in rules than on facts or templates. An useful extension of the CLIPS external interface, based on the popularity of the C++ programming language, would be the definition and documentation of a C++ frontend for COOL objects.

REFERENCES

1. D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, 1988.

2. H. Burkhart and S. Gutzwiller. Steps towards reusability and portability in parallel programming. In K.M. Decker and R.M. Rehmann, editors, Programming Environments for Massively Parallel Distributed Systems, pages 147 157. Birkh"auser Verlag, Basel, 1994.

3. M. Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA., 1989.

4. K.M. Decker, J.J. Dvorak, and R.M. Rehmann. A knowledge-based scientific parallel programming environment. In K.M. Decker and R.M. Rehmann, editors, Programming Environments for Massively Parallel Distributed Systems, pages 127 - 138. Birkhauser Verlag, Basel, 1994.

5. K.M. Decker and R.M. Rehmann. Simple and efficient programming of parallel distributed systems for computational scientists. Technical Report IAM-92019, IAM, University of Berne, 1992.

6. J. Dvorak. An AI-based approach to massively parallel programming. Technical Report CSCS-TR-93-04, Swiss Scientific Computing Center CSCS, CH6928 Manno, Switzerland, 1993.

7. J. Giarratano and G. Riley. Expert Systems: Principles and Programming. PWS Publishing, Boston, MA., 2nd. edition, 1994.

8. NASA Lyndon B. Johnson Space Center. CLIPS Reference Manual, 6.0 edition, 1993.

9. C.D. Polychronopoulos. Parallel Programming and Compilers. Kluwer Academic Publishers, Boston, 1988.

10. H. Zima and B. Chapman. Supercompilers for parallel and vector computers. Addison-Wesley, Workingham, 1990.

TRANSPORT AIRCRAFT LOADING AND BALANCING SYSTEM: USING A CLIPS EXPERT SYSTEM FOR MILITARY AIRCRAFT LOAD PLANNING

Captain James Richardson, Marcel Labbe, Yacine Belala, and Vincent Leduc
Group of Applied Research in Management Support Systems
College militaire royal de Saint-Jean
Richelain, Quebec, J0J 1R0
Vincent Leduc@CMR.CA

ABSTRACT

The requirement for improving aircraft utilization and responsiveness in airlift operations has been recognized for quite some time by the Canadian Forces. To date, the utilization of scarce airlift resources has been planned mainly through the employment of manpower-intensive manual methods in combination with the expertise of highly qualified personnel. In this paper, we address the problem of facilitating the load planning process for military aircraft cargo planes through the development of a computer-based system. We introduce TALBAS (Transport Aircraft Loading and BAlancing System), a knowledge-based system designed to assist personnel involved in preparing valid load plans for the C130 Hercules aircraft. The main features of this system which are accessible through a convivial graphical user interface, consists of the automatic generation of valid cargo arrangements given a list of items to be transported, the user-definition of load plans and the automatic validation of such load plans.

INTRODUCTION

The aircraft load planning activitiy represents the complex task of finding the best possible cargo arrangement within the aircraft cargo bay, given a list of various items to be transported. In such an arrangement, efficient use of the aircraft space and payload has to be made while giving due consideration to various constraints such as the aircraft centre of gravity and other safety and mission related requirements. The diversity of the equipment to be transported and the variety of situations which may be encountered dictate that highly trained and experienced personnel be employed to achieve valid load plans. Additionally, successful support of military operations requires responsiveness in modifying conceived load plans (sometimes referred to as "chalks") to satisfy last-minute requirements.

The development of a decision support system to assist load planning personnel in their task appears to be a natural approach when addressing the simultaneous satisfaction of the numerous domain constraints. The chief idea behind TALBAS consists of encoding the knowledge of domain experts as a set of production rules. The first section of this paper is devoted to a description of the fundamental characteristics of the load planning problem. A review of major contributions and research work in the field of computer-assisted load planning is then presented. The next section provides a detailed description of the TALBAS architecture and an overview of the modelled reasoning approach used to achieve valid load plans. Finally, a list of possible extensions to the current system is given.

PROBLEM STATEMENT

The problem of interest may be briefly stated as summarized by Bayer and Stackwick [1]:

> “there are various types of aircraft which may be used, each with different capabilities and restrictions and different modes in which it may be loaded, and finally, there is a large variety of equipment to be transported.”

Additionally, the load's center-of-balance must fall within a pre-defined Center of Gravity (CG) envelope for all loading methods, otherwise imbalance can render the airplane dynamically unstable in extreme cases.

First addressing the issue of diversity of equipment to be transported, we note that each aircraft can be loaded with cargo including any or all of the following items: wheeled vehicles, palletized cargo, tracked vehicles, helicopters, miscellaneous equipment and personnel. Each type of cargo item requires different considerations. The wide variety of cargo items to be loaded poses a significant difficulty in that a feasible load configuration has to be selected from a very large number of possible cargo arrangements, given a number of pre-defined areas within the aircraft cargo bay which may each impose different restrictions on the type of item which may be loaded depending on weight limitations and other constraints.

Secondly, the load arrangements are dependent upon the following delivery methods : strategic air/land, airdrop, low-altitude parachute extraction system (LAPES) and tactical air/land.

For strategic air/land missions, the emphasis is on maximum use of aircraft space and few tactical considerations are involved. In a tactical air/land mission however, speed and ease with which cargo can be on-loaded and off-loaded takes precedence.

Airdrop operations consist of off-loading equipment and personnel by parachute. LAPES is executed when the aircraft is flown very close to the ground over a flat, clear area. In this case, an extraction parachute is deployed from the rear of the aircraft, after which the aircraft reascends. The equipment, mounted on a special platform, is pulled from the aircraft by the drag on a parachute and skids to a halt. Airdrop and LAPES missions must maintain aircraft balance while dropping cargo and must also provide space between items for parachute cables.

A third issue concerns the characteristics and limitations pertaining to the type of aircraft being loaded. The load capacity of an aircraft depends on its maximum design gross weight (i.e., weight when rounded or upon take-off), its maximum landing gross weight and maximum zero fuel weight (i.e., weight of aircraft and its load when fuel tanks are empty). The maximum design gross weight includes the fuel load to be carried which is determined, at the outset, on the basis of the distance to be flown and other operational requirements. The established quantity of aircraft fuel in turn influences the allowable load which corresponds to the maximum cargo/personnel weight the aircraft can carry.

Finally, constraints pertaining to loading, unloading and in-flight limitations have to be checked when an item is loaded into the aircraft cargo bay. To model a realistic situation, safety and mission related restrictions and parameters such as transport of dangerous goods and cargo/personnel movement priorities, are also taken into account in the set of constraints.

A mathematical formulation of the aircraft loading problem may be obtained through the application of Operations Research techniques. The underlying closely related bin-packing problem (Friesen and Langston [2]) is known to be NP-hard in the strong sense, suggesting it is unlikely that a polynomial solution exists (Garey and Johnson [3]). However, if the requirement of finding the best solution is relaxed to finding a good solution, then heuristic techniques can be applied successfully (Pearl [4]). For a number of reasons to be presented later in this paper, we propose to adopt a heuristic approach based on the knowledge of domain experts. In the following section, we undertake a review of two successful endeavors aimed at facilitating aircraft load planning, that is AALPS and CALM.

The Automated Aircraft Load Planning System (AALPS)

AALPS was developed for the US Forces in the early 1980s, using the Sun workstation as the platform. This load planning tool is a knowledge-based system built using the expertise of highly trained loadmasters.

The underlying system architecture is designed to serve three basic functions: the automatic generation of valid loads, validation of user-defined load plans and user-modification of existing load plans. To perform these tasks, AALPS incorporates four components:

1) a graphical user interface which maintains a graphical image of the aircraft and equipment as cargo items are being loaded;
2) an equipment and aircraft databases which contain both dimensional information and special characteristics such as those required to compute the aircraft CG;
3) the loading module which contains the procedural knowledge used to build feasible aircraft loads; and
4) a database which allows the user to indicate his preference from a set of available loading strategies.

Despite its seemingly powerful features, AALPS would probably necessitate some improvements at significant costs to meet the requirements of the Canadian Forces.

Another initiative in the area of computer-assisted load planning led to the implementation of the Computer Aided Load Manifest (CALM), which was initiated by the US Air Force Logistics Management Centre in 1981.

The Computer Aided Load Manifest (CALM)

The project to develop CALM, formerly called DMES (Deployment Mobility Execution System), was launched at approximately the same time as AALPS but the established objectives were slightly different. In contrast with AALPS, CALM was to be a deployable computer-based load planning system. Consequently, the selected hardware was to be easily transportable and sufficiently resistant for operation in the field.

CALM uses a modified cutting stock heuristic to generate "first-cut" cargo loads, which the planner can alter through interactive graphics.

This system incorporates fewer functional capabilities than AALPS and runs on a PC compatible platform, thus resulting in significantly lower development costs. A major reproach, however, is that this system does not appear to take explicitly into account important load planning data such as the weight of the aircraft fuel being carried, hence leading to incomplete results in some cases. Additionally, the graphical interface would require some improvement to be considered sufficiently user-friendly. In spite of these deficiencies, CALM remains a very useful tool to aid in the load planning process.

In view of the successful research work accomplished in the area of automated aircraft load planning based on the emulation of expert behavior in this field, we have decided to concentrate our efforts on the development of an expert system. The intent in doing so, is to combine the most valuable features present in the existing systems, namely AALPS and CALM, and to adapt them to produce a tool tailored to the Canadian military environment. Our objective is to develop a system with the following features:

1) be deployable to and operable at remote sites;

2) be easy to use and provide a convivial user interface;
3) be capable of handling aircraft load planning problems involving a wide variety of items and several aircraft types used by the Canadian Forces;
4) deal with different types of mission, including the ones with more than one destination;
5) automatically generate valid load plans in a reasonable time;
6) allow the user to alter plans automatically generated by the system;
7) allow users to define their own load plans and issue warnings whenever constraints are violated.

The next section will describe the functional architecture and design of TALBAS which enable the integration of the above described desired features.

THE SYSTEM ARCHITECTURE

The functional architecture depicted at Figure 1 serves the same three basic functions as in AALPS: automatic generation of cargo arrangements, user-definition of load plans and automatic validation of existing load plans. TALBAS consists of an interactive user interface, a loading module, a mission and an aircraft database, and necessary communication links for access to some databases available within the Canadian Forces. The following provides a description of the various databases required to build a feasible cargo load:

1) the aircraft database contains detailed aircraft characteristics including dimensional information and a description of the constraint regions for the C130 Hercules aircraft;

2) the mission database contains detailed mission features mainly in the form of applicable loading strategies;

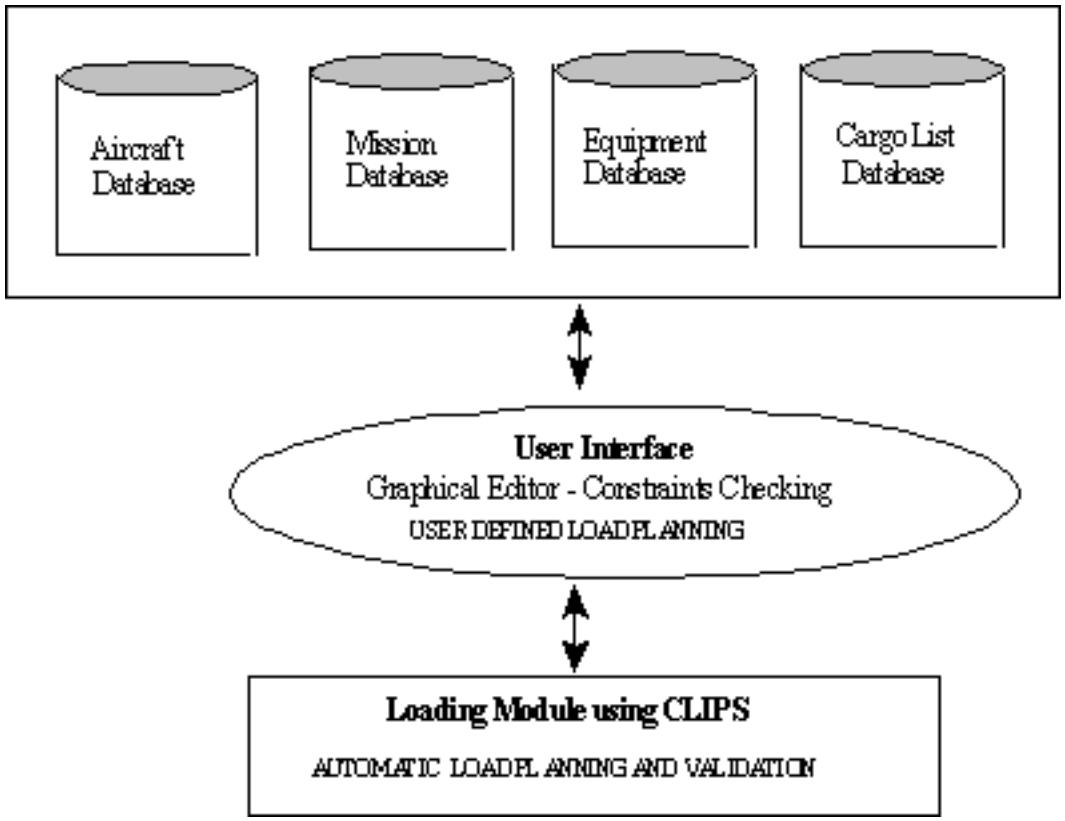


Figure 1. TALBAS Functional Architecture

3) the equipment and cargo list databases together provide a detailed listing of all cargo items and personnel to be deployed for a mission, including all necessary information about each particular object such as the weight and dimensional characteristics, and the order in which the items are to be transported To minimize the requirement for user input, a set of default values are provided for the majority of the objects' characteristics and these may be changed at the user's request as the situation dictates.

The user interface module performs two main tasks: the graphical display and the constraints checking. The aircraft floor and the items being loaded are graphically represented on the screen. Objects representing loaded cargo items and the various areas of the aircraft cargo floor are drawn on the basis of information available in the different databases described above. All onscreen cargo items are treated as active objects; they can be dragged and moved within the aircraft cargo bay after being selected with a mouse.

The constraints checker ensures that any violation of the center of gravity limits or of other in-flight restrictions is reported through the display of warning message dialog boxes. All the critical load planning data, such as the aircraft CG upon take-off, are computed every time an item is placed or moved within the aircraft cargo area.

The loading module basically consists of an expert system which automatically generates valid load plans. The load planning process will be reviewed in details in the next section.

TALBAS has been developed through the implementation of an incremental prototyping methodology whereby the end user is continuously involved in the refining process of the current prototype. The availability of an ever-improved GUI and loading module, allows for a fail-safe capture of user requirements. The PC platform has been selected for development purposes in order to produce, in a cost effective fashion, a deployable tool operable at remote sites as a stand-alone system. TALBAS has been designed with an object oriented approach to favor reusability.

The user interface has been implemented using Microsoft Windows and C++. The expert system portion of TALBAS is an enhanced application of the expert system shell CLIPS. This module has been created as a Dynamic-Link Library (DLL). A DLL is a set of functions that are linked with the main application at run time which is, in our case, the user interface. When faced with the problem of having CLIPS communicate with the graphical user interface, three alternatives were contemplated: embedding of CLIPS-based loading module into the main program; implementation of the CLIPS-based loading module as a DLL: or use of the Dynamic Data Exchange (DDE) concept.

DDE is most appropriate for data exchanges that do not require ongoing user interaction. However, the requirement for the devised expert system to constantly monitor any changes made by the user when modifying load plans onscreen has made the approach an unacceptable one. Embedding the CLIPS-based loading module within the main application required that both the user interface and the loading module fit within 64K of static data. This is not possible since the CLIPS-based application (version 6.0) uses all this amount of memory space. On the other hand, implementing the CLIPS-based application as a DLL allows the former and the user interface to be considered separately such that each of the two application codes can fit within the limit of 64K of static data.

The Loading Module

A heuristic approach has been selected as a solution to address the load planning problem. The loading methodologies applied by loadmasters and load planners have been encoded as a set of production rules. This choice was first motivated by the fact that rules of thumb are often the only means available to the experts when seeking a good first try---one requiring the fewest alterations--- to achieve a valid load. As an example, load planning experts generally agree to follow a "60--40" rule of thumb, namely 60% of the load weight has to be located at the front of the aircraft and 40% at the rear. The second reason for the choice of a knowledge-based system is that the expert knowledge represented in the form of production rules can be easily maintained and updated to incorporate new assets as they are acquired by the Canadian Forces or to adapt to new situations.

Among the three basic objectives, namely the automatic generation of valid load plans, the automatic validation of user-defined load plans and the user-definition of load plans, the first two are achieved using the expert system in TALBAS. The third objective is achieved through the implementation of a graphical user interface allowing the user to manipulate objects onscreen. The expert system is made of two distinct modules as can be seen from Figure 2. The Loading knowledge-base contains all the information related to aircraft loading procedures while the rules contained in the Validation knowledge-base allow the system to recover from situations characterized by a non-balanced aircraft or violated constraints.

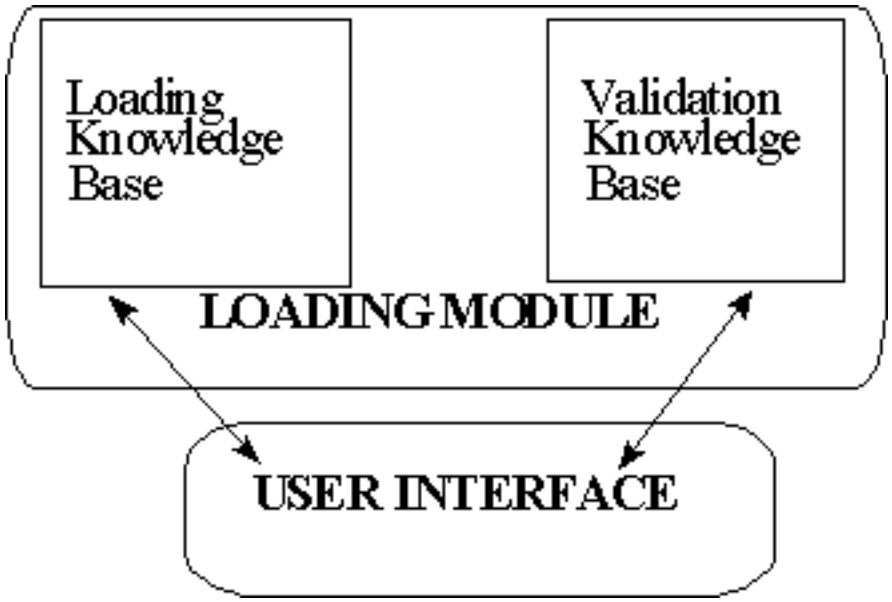


Figure 2. TALBAS Expert System

The Loading knowledge-base contains all rules which are essential to initially identify one possible load configuration. As in AALPS, an initial cargo state contained in a database of facts is iteratively transformed by our system into an acceptable cargo arrangement by applying a sequence of operators representing the loadmaster's knowledge. In generating a load configuration, pre-defined areas of the aircraft floor are assigned to each category of items or passengers to be transported. For instance, when both cargo and passengers have to be loaded, passengers are usually assigned to the front of the aircraft. Next, a selection of the appropriate item in each category (e.g. vehicles) is made, based on weight, width and length considerations. Since the aircraft balance is a primary concern, a preliminary selection among all cargo items is accomplished on the basis of weight followed by consideration of dimensional information

depending on the type of item selected. The integrated computation module will subsequently compute the achieved CG and verify that numerical constraints have not been violated.

At this step, the coded heuristics contained in the loading knowledge-base are likely to have produced an acceptable "first-cut" load plan. However, the system will try to find a cargo arrangement characterized by an optimal CG value while ensuring that all constraints are satisfied. If the initial load is not acceptable, the system will make use of rules stored in the validation knowledge-base to first slide the loaded items, if there is sufficient aircraft space remaining to do so, or attempt to rearrange these items if sliding operations are not feasible. TALBAS will stop when it has identified either an optimal load plan or an acceptable one. In such cases where no possible solutions may be found for the list of given items to be loaded and no more permutations can be made, the system issues a warning message indicating that no valid load plans can be found with that combination of items.

A second role played by the expert system module concerns the validation of user-defined load plans. In this case, it operates in the same fashion as described above when automatically generating valid load plans with the difference that the initial load plan is produced by the user himself.

CONCLUSION AND FUTURE WORK

In this paper, we have presented a knowledge-based alternative to facilitate load planning of military aircraft. We have successfully incorporated most of the valuable features present in the existing systems, AALPS and CALM, and adapted them to produce a tool tailored to meet the specific requirements of the Canadian military environment. The major concerns which have been addressed were the deployability and conviviality of the designed system. The CLIPS-based expert system module automatically generates valid load configurations and validates user-defined/modified load plans. The developed graphical user interface allows for the easy alteration of existing plans.

Efforts in the design of the developed system have been primarily focused on the loading of the C130 Hercules aircraft, since it is currently the principal airlift resource used by the Canadian Forces for the transport of cargo and personnel. The system may however, be easily adapted to accommodate other existing or future types of Canadian Forces transport aircraft. Expansion of TALBAS to permit the loading of several aircraft, while giving due consideration to the movement priority level assigned to each item to be airlifted, is currently under development.

REFERENCES

[1] Bayer, W.K., and Stackwick, D.E., "Airload Planning Made Easy," *Army Logistician*, May-June, 1984, pp. 34-37.

[2] Friesen, D.K., and Langston, M.A., "Bin Packing: On Optimizing the Number of Pieces Packed," *BIT*, 1987, pp. 148-156.

[3] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, California, 1979.

[4] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, Massachusets, 1984.

[5] Anderson, D., and Ortiz, C., "AALPS: A Knowledge-Based System for Aircraft Loading," *IEEE*, Winter 1987, pp. 71-79.

[6] Cochard, D.D., and Yost, K.A., "Improving Utilization of Air Force Cargo Aircraft," *Interfaces*, January-February, 1985, pp. 53-68.

[7] Ignizio, J.P., *Introduction to Expert System: The Development and Implementation of Rule-Based Expert Systems*, McGraw-Hill, Houston, Texas, 1991.

[8] Ng, K.Y.K., "A Multicriteria Optimization Approach to Aircraft Loading," *Operations Research*, November-December, 1992, pp. 1200-1205.

PREDICTING AND EXPLAINING THE MOVEMENT OF MESOSCALE OCEANOGRAPHIC FEATURES USING CLIPS

Susan Bridges
Liang-Chun Chen
Computer Science Department
Mississippi State University[3]

Matthew Lybanon
Naval Research Laboratory
Stennis Space Center, MS

ABSTRACT

The Naval Research Laboratory has developed an oceanographic expert system that describes the evolution of mesoscale features in the Gulf Stream region of the northwest Atlantic Ocean. These features include the Gulf Stream current and the warm and cold core eddies associated with the Gulf Stream. An explanation capability was added to the eddy prediction component of the expert system in order to allow the system to justify the reasoning process it uses to make predictions. The eddy prediction and explanation components of the system have recently been redesigned and translated from OPS83 to C and CLIPS and the new system is called WATE (Where Are Those Eddies). The new design has improved the system's readability, understandability and maintainability and will also allow the system to be incorporated into the Semi-Automated Mesoscale Analysis System which will eventually be embedded into the Navy's Tactical Environmental Support System, Third Generation, TESS(3).

INTRODUCTION

One of the major reasons CLIPS is so widely used is the ease with which it allows a rule base to be incorporated as one component of a larger system. This has certainly been the case with the eddy prediction component of the Semi-Automated Mesoscale Analysis System (SAMAS) (3). SAMAS is an image analysis system developed by the Naval Research Laboratory that includes a variety of image analysis tools that enable the detection of mesoscale oceanographic features in satellite images. Unfortunately, in the North Atlantic, many of the images are obscured by cloud cover for lengthy periods of time. A hybrid system for use when features cannot be detected in images has been developed that consists of a neural network that predicts movement of the Gulf Stream and a rule base that predicts movement of eddies associated with the Gulf Stream. The Gulf Stream and eddy prediction components were both originally implemented in OPS83 (4). The Gulf Stream Prediction Module has been replaced by a neural network (3) and an explanation component has recently been added to the OPS83 version of the eddy prediction component (1). The eddy prediction rule base, WATE (Where Are Those Eddies), has been translated to CLIPS because of the ease of integrating a CLIPS rule base into a larger system, the ability to access routines written in C from CLIPS rules, and the support CLIPS provides for the forward chaining reasoning used by the eddy prediction system. The explanation component of WATE uses meta rules written in CLIPS to compose either rule traces or summary explanations of the predicted movement of eddies.

[3]This work was supported by the Naval Research Laboratory, Stennis Space Center under contract NAS 13-330.

SYSTEM ARCHITECTURE

External Interfaces

The WATE component interacts with other SAMAS components as shown in Figure 1. WATE interacts with the User Interface in two ways. First, WATE is invoked from the User Interface when the user requests a prediction of Gulf Stream and eddy movement for a specified time. Second, as WATE predicts the movement of the Gulf Stream by calling the Gulf Stream Prediction Module and eddies by running the rule base, WATE calls User Interface routines to update the graphical display of the positions of the Gulf Stream and eddies. The eddy prediction rules call the Geometry Routines to compute distances and locations. WATE invokes the Gulf Stream Prediction Module to predict the movement of the Gulf Stream for each time step. The position of the Gulf Stream predicted by the neural network component must be accessed by the eddy prediction rule base since the movement of eddies is influenced by the position of the Gulf Stream.

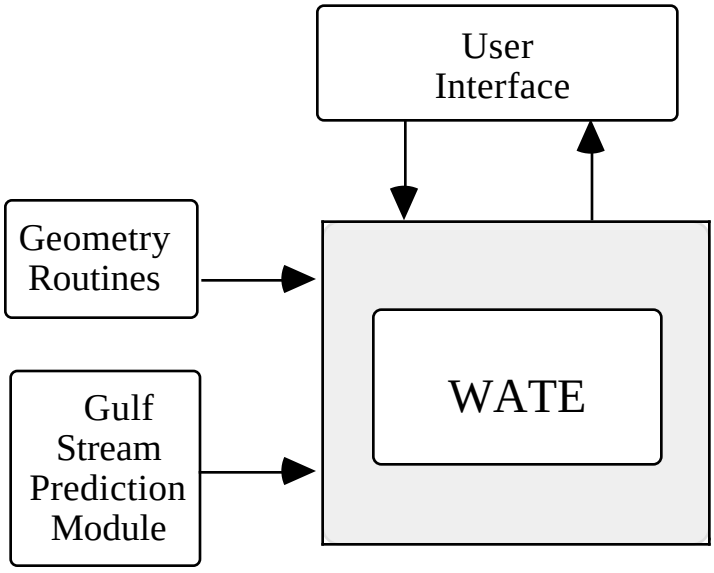


Figure 1. External Interfaces of WATE

Redesigned Control Structure

The control structure for the original expert system was written in procedural OPS code and had been modified a number of times as the graphical user interface, eddy prediction, Gulf Stream prediction, and explanation components were either modified or added. The result was a control structure that was not modular and that contained a substantial number of obsolete variables and statements. When the system was converted from OPS83, the control structure was completely redesigned and rewritten in C. Pseudo code for the redesigned control structure is shown in Figure 2. The resulting code was more efficient because it was written in compiled C code rather than interpreted OPS83 procedural code.

```
Initialization
        GetUserOptions
        SetUpCLIPS
        SetUpPVWave
        SetUpGulfStream

Prediction
        Display initial positions of GS and eddies
        for each time step do
                update time
                MoveGulfStream
                Update display of GS
                MoveEddies
                Update display of eddies

Explanation
        Explanation Composition
        Explanation Presentation

FinalOutput
```

Figure 2. Control Structure of WATE

Translation from OPS83 to CLIPS

The original OPS83 working memory elements and rules had been completely restructured to support an explanation component (1). The translation of this restructured OPS83 code into CLIPS was fairly simple since CLIPS has evolved from the OPS line. There is a very straightforward translation from OPS83 working memory elements to CLIPS fact templates and from OPS rules to CLIPS rules. In some cases, the OPS83 rules called OPS83 functions or procedures. These functions and procedures were translated to C.

EXPLANATION COMPONENT

The explanation component allows the user to ask for either a rule trace or summary explanation for the prediction of the movement of each eddy at the end of each prediction cycle. The rule trace explanations give a detailed trace of the instantiation of all rules that were fired to predict the movement of an eddy. Although this type of trace has proven to be very useful in debugging the system, it was immediately apparent that it contained a great deal of information that would be of little interest to most users. Interviews with domain experts were used to determine the information that would be of most interest to a user. The types of information they identified was used to design the summary explanations. Presentation of these explanations requires that the line of reasoning of the system be captured as the rules fired and that information from this rule trace be extracted and organized for presentation to the user.

Rule Firing Capture

Capturing the rule trace for this domain in a usable form is simplified because all explanations (both trace and summary) are focused on a particular eddy. This means that all of the rule-firings

pertaining to the movement of one eddy can be stored together and presented as one explanation. This is accomplished by asserting a *rule-fire-record* template fact for each eddy for each time step with the following deftemplate definition:

```
(deftemplate rule-fire-record
   (field ringtype
      (allowed-symbols wcr ccr))
   (field refno
      (type INTEGER))
   (field time
      (type INTEGER))
   (multifield rules-fired
      (type SYMBOL)))
```

The ringtype, eddy identifier (*refno*), and time stamp (*time*) uniquely identify each rule-fire-record. The *rules-fired* multifield is used to store a list of the names of the rules that fired to predict the movement of the eddy during this time step. Each time a rule fires as part of the prediction process for a particular eddy, the rule name is added to the end of the *rules-fired* list .

A second set of template facts is used to record the instantiation of each rule that fires. Each time a rule fires, a *values-for-explanation* template fact is asserted which gives the value bound to each variable when the rule was fired. The deftemplate definition for *values-for-explanation* is:

```
(deftemplate values-for-explanation
   (field rule-name
      (type SYMBOL))
   (field ringtype
      (allowed-symbols wcr ccr))
   (field refno
      (type INTEGER))
   (field time
      (type INTEGER))
   (multifield var-val
      (type SYMBOL)))
```

This template contains slots for the rule name, the eddy identifier and type, and the time stamp. In addition, it contains a multifield slot whose value is a sequence of variable value pairs that gives the name of each variable used in the rule-firing and the value bound to that variable when the rule fired. This approach can be used in this domain because a single rule will never fire more than one time for a particular eddy during one time step, and all slots in templates used by the eddy prediction rules are single value. The records of the rules that fired for a particular eddy are used by meta rules to produce the explanations.

Explanation Construction and Presentation

If the user requests an explanation for a specific eddy, a set of explanation meta rules are used to construct an explanation for the predicted movement of an eddy. The user may request either a rule-trace explanation or a summary explanation. When the user makes the request, a sequence of *explain-eddy* facts for that eddy are asserted each with a progressively higher time stamp. The fact template for an *explain-eddy* fact is:

```
(explain-eddy  ccr | wcr  <ref-no>  <time>  summary | rule-trace).
```

The presence of this fact causes the *explain-single-eddy* rule to fire one time for each rule that fired to predict the movement of the eddy for that time step. Each *explain-single-eddy* rule firing

matches a rule name from the *rules-fired* slot of the *rule-fire-record* with a *values-for-explanation* template for that eddy type, number, and time stamp. A *fill-template* fact is then asserted into working memory which contains all of the information needed to explain that rule firing--either in rule-trace or summary form. For each rule, there are two explanation meta rules. The first is used when a rule-trace has been requested and is a natural language translation of the rule. It will give the value of all variables used in the rule instantiation. The second is used when a summary has been requested. It gives a much shorter summary of the actions of the rule. A few rules that are used to control the sequence of rule-firing produce no text for a summary explanation. When an explanation metarule fires, it causes a natural language translation of the rule to be sent to the user interface for presentation to the user.

The user may request an explanation of the movement of all eddies instead of just a single eddy. In this case, the process above is simply repeated for each eddy.

SUMMARY AND FUTURE WORK

WATE has been successfully converted from OPS83 to C and CLIPS. This conversion will facilitate the incorporation of WATE into SAMAS 1.2 which will eventually be embedded in TESS(3). The modular control structure of WATE is easier to understand and maintain than that of the previous system. The explanation component has been implemented using CLIPS metarules. This causes some additional maintenance burden since the two metarules that correspond to each rule must be modified if a rule is modified. In the present system, the *rule-fire-record* and *values-for-explanation* template facts are asserted by each individual rule. We are currently modifying the CLIPS inference engine to capture this information automatically as the rules fire.

Explanations produced by the current system have two major shortcomings. First, there is still a great deal of room for improvement in the summarization capabilities of the system. In particular, the system should be summarizing over both temporal and spatial dimensions. If an eddy's predicted movement is essentially in the same direction and speed for each time step, then all of this information should be collapsed into one explanation. Likewise, if several eddies all have similar movement over one or more time steps, this should be collapsed into a single explanation. The second shortcoming deals with the lack of explanation of the predictions of the neural network component. Some recent results reported in the literature have addressed this sort of problem.

REFERENCES

1. Bridges, S. M., and Lybanon, M. "Adding explanation capability to a knowledge-based system: A case study," pp. 40-49, Applications of Artificial Intelligence 1993: Knowledge-Based System in Aerospace and Industry, Orlando, FL, April, 1993.

2. Lybanon, M. "Oceanographic Expert System: Potential for TESS(3) Applications," Technical Note 286, Naval Oceanographic and Atmospheric Research Laboratory, Stennis Space Center, MS, 1992.

3. Peckinpaugh, S. H. Documentation for the Semi-Automated Mesoscale Analysis System 1.2, 1993.

4. Thomason, M. G., and Blake, R. E. NORDA Report 148, Development of An Expert System for Interpretation of Oceanographic Images. Naval Ocean Research and Development Activity, Stennis Space Center, MS, 1986.

KNOWLEDGE BASED TRANSLATION AND PROBLEM SOLVING IN AN INTELLIGENT INDIVIDUALIZED INSTRUCTION SYSTEM

Namho Jung and John E. Biegel
Intelligent Simulation Laboratory
Department of Industrial Engineering and Management Science
University of Central Florida
Orlando, Florida 32816

namho@oak.ists.engr.ucf.edu
biegel@oak.ists.engr.ucf.edu

ABSTRACT

An Intelligent Individualized Instruction (I^3) system is being built to provide computerized instruction. We present the roles of a translator and a problem solver in an intelligent computer system. The modular design of the system provides for easier development and allows for future expansion and maintenance. CLIPS modules and classes are utilized for the purpose of the modular design and inter module communications. CLIPS facts and rules are used to represent the system components and the knowledge base. CLIPS provides an inferencing mechanism to allow the I^3 system to solve problems presented to it in English.

INTRODUCTION

The Intelligent Individualized Instruction (I^3) system is an intelligent teaching system that makes possible the knowledge transfer from a human to a computer system (knowledge acquisition), and from the system to a human (intelligent tutoring system (ITS)). The ITS portion of the I^3 system provides an interactive learning environment where the system provides self-sufficient instruction and solves the problem presented in a written natural language. Self-sufficient instruction means that no instructor is required during the learning cycle. Solving problems written in a natural language means that the system is able to 'understand' the natural language: It is not an easy task, especially without any restriction on the usage of vocabulary and/or format.

Two I^3 system modules, a Translator and an Expert (a problem solver and a domain glossary), understand a problem presented to it in English by translation and keyword pattern matching processes. The I^3's pattern matching method uses the case-based parsing [Riesbeck and Schank 1989] that searches its memory (knowledge base) to match the problem with stored phrase templates. Unlike other case-based parsers (e.g., CYC project [Lenat and Feigenbaum, 1989]) the I^3 system does not understand the problem statement as a human does. A human uses common sense or other background knowledge to understand it. Rather the I^3 system 'understands' enough about the problem for the system to be able to solve the problem. We will discuss how the system 'understands' the human language.

AN INTELLIGENT INDIVIDUALIZED INSTRUCTION (I^3) SYSTEM

The I^3 system is a Knowledge Based System that is composed of domain dependent modules, domain independent modules, and a User interface module. The domain dependent modules (the Domain Expert and the Domain Expert Instructor) carry the domain expertise that enables the other modules remain domain independent. The separation of these domain dependent modules from the rest of the system makes the system reusable. Whenever the I^3 system is applied to another domain, only the domain-dependent knowledge of the new domain is needed.

The goal of the I^3 system is to provide a student with individualized learning to attain competency [Biegel 1993]. The I^3 knowledge presentation subsystem generates self-sufficient instructions. The I^3 system contains instructional components (Student Model, Teacher, Domain Expert Instructor, Control, and User Interface) and problem solving components (Translator and Domain Expert).

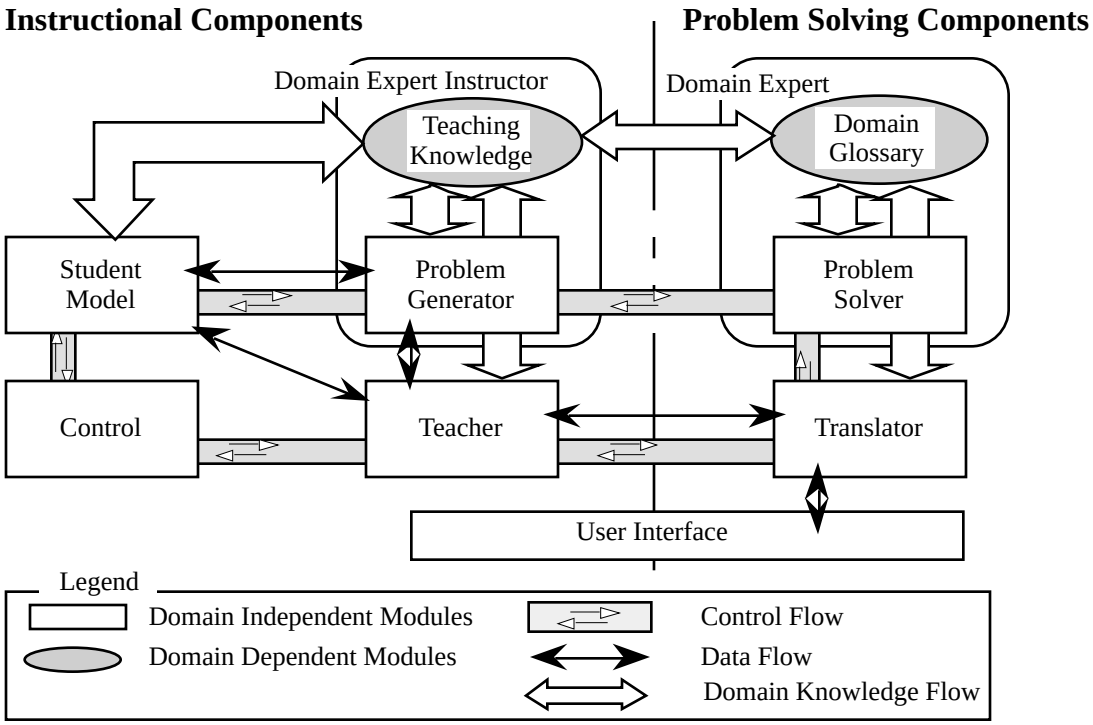


Figure 1. System architecture of I^3's intelligent teaching system

- The Student Model module evaluates and maintains the trainee's performance overall or on individual lessons.
- The Teacher module contains the knowledge about generic didactic strategies to customize each lesson by selecting and sequencing the instruction material.
- The Domain Expert Instructor (DEI) module represents the domain dependent teaching methodology. The DEI module provides the Teacher module with the teaching strategies, and the Student Model module with the evaluation criteria for the individual and/or overall lesson.
- The Control module manages the operation and maintains the modules' communications within the system.
- The User Interface module handles all communications between the user and the system.
- The Translator module parses a trainee's input and translates it into a system-understandable format.
- The Domain Expert (DE) module contains a knowledge base representing the problem-solving knowledge of a human domain expert.

DESIGN OF THE TRANSLATOR AND THE PROBLEM SOLVER

The Translator module and the Domain Expert module provide the user with the proper interpretation of the problem and the correct solution. Together, they allow a system to apply

domain expert heuristics to solve problems by pattern matching. Pattern matching allows the system to 'understand' a problem statement within the domain for which the system has been built. The problem statement can be translated into a set of rules and facts. Since the problem solver cannot translate English (or other natural language) directly into computer readable code, it relies on a translator to provide a communication mechanism between the user and the computer.

The Translator provides a full duplex communication medium between the human and the computer. The user will not be constrained in the format of the input problem and will not be required to do any parsing or be restricted to a limited syntax. The Translator module translates the domain jargon of an input problem into a system understandable format.

The major tasks of the translator are: (1) to convert text into computer readable code, and (2) to provide a knowledge base conversion and problem representation process. The text conversion process includes checking for correct spelling and removing all unnecessary symbols. The knowledge base conversion process includes (1) the conversion of a written number to a numeric value, (2) filtering out unnecessary words, and (3) replacing words with stem (root) or exemplar words. The knowledge based process uses a number conversion list, a list of words unnecessary for problem solving, and a domain thesaurus. The Intelligent Individualized Instruction system separates the knowledge base (domain glossary) which is the collection of the domain thesaurus, the domain template dictionary, and the unnecessary word listing, from the translation process.

The Domain Expert interprets the translated input using a domain vocabulary as a reference, selects a suitable method from a list of solution methods, and finds a solution. The Domain Expert (DE) consists of three parts, a Domain Glossary (DG), a set of Managers, and a Problem Solver (PS). The DE imitates the expert's methodologies of problem solving: The DG acts as the expert's memory by providing the necessary problem solving knowledge. Each Manager acts as the expert's procedural knowledge by solving a routine of the problem in one specific area. The PS acts as the scheduling and reasoning processes by controlling and scheduling the Managers in proceeding toward the solution of a problem.

The Domain Glossary represents the domain expertise. The DG consists of a domain template dictionary, a domain thesaurus, an unnecessary word listing, a domain symbol list, and a domain theory list. The DG represents relational knowledge (e.g., one year is twelve months), factual knowledge (e.g., 'interest rate' means domain variable i), and a list of words and symbols (e.g., '%' has a special meaning of interest rate in the Engineering Economy domain).

Special dictionaries for the domain provide benefits such as faster look up access than a general dictionary, and a higher priority to find the correct interpretation. The problem solver does not have to consider all different combinations of words' variables. It searches its own smaller dictionary that contains the necessary information in the application domain. English text (problem statement) is interpreted/translated by using the domain thesaurus without knowing the general meaning of the word. The thesaurus contains all words relevant to the domain. Each word is connected to a list of possible interpretations. Each word in the statement is looked up in the thesaurus and replaced by all relevant symbols.

The number of words in the domain vocabulary will vary between domains, but a vocabulary of 500 or so words and symbols will most likely cover most of the undergraduate engineering domains.

Each Manager handles one specific area of problem solving. It is a self-sufficient object that contains procedural knowledge (rules) and factual knowledge (facts) attached to it, and that knows how to handle situations upon request. When activated, a Manager searches the input statement for matched patterns in its templates. If a match is found, the Manager processes or

interprets that portion of the problem statement. For example, a Date Manager knows how to interpret key terms that are related to the date, how to compare two date instances, and how to calculate the period from two date instances. When a problem statement contains "… invest $5000 on January 1, 1994, … will be accumulated in 5 years hence … ", the Date Manager replaces the statement with " … invest $5000 on [D1] … will be accumulated in [D2] … " where [D1] and [D2] are instances of a Date class and are represented as:

```
([Date::D1] is a Date                ([Date::D2] is a Date
      (year        1994)                 (year        5)
      (month  1)                         (month  0)
      (day         1)                    (day         0)
      (base        none) )               (base        [Date::D1]) )
```

Some mangers handle both domain dependent and independent situations based on the factual knowledge they have. Communication among managers can be made through dedicated communication channels, such as CLIPS class objects or templates.

The Domain Expert module is of a modular design and maintains the separation of strategic knowledge from factual knowledge. The domain expertise can be categorized into three levels: high level control knowledge (a Problem Solver), middle level procedural knowledge (Managers), and low level factual knowledge (a Domain Glossary). By nature, the low level factual knowledge tends to be domain specific, and the high level control knowledge tends to be a domain independent. Any addition to the knowledge base can be accomplished by adding a Manager and its associated knowledge into the DG.

PROBLEM SOLVING IN THE I^3 SYSTEM

The problem solver applies a separate-and-solve method that breaks a problem statement into several small blocks, interprets each block, and then logically restructures them. The problem solving steps include interpretation of the problem statement, selection of the formula, and mathematical calculation. The steps are depicted in Figure 2 in which boxes on the left hand side represent the changes of the problem statement from input English text to the answer. The middle ovals show the problem solving processors. The right hand side boxes represent the domain expertise of the domain glossary. The problem solving process is generic so it can be used in other domains if the new domain expertise is available.

The I^3 system problem solving routine is performed by the problem solving components: the User Interface, the Translator, and the Domain Expert (the Problem Solver, the Managers, and the Domain Glossary). The routine includes initial domain independent processes (translating and filtering an input problem), and main domain dependent processes (interpreting the problem, selecting a solution method, and deriving an answer).

A user enters an engineering economics problem through the User Interface, as shown in Figure 3. The Translator performs filtering process by checking correct spelling using its English dictionary. The Translator converts the input problem statement into system understandable format; plural words to singular; past tense to present; uppercase words to lowercase; verbal numbers to numeric values; and separates symbols from numeric values (Figure 4). For example, part of the problem statement "If $*10,000* is *invested* at 6% interest *compounded* annually" becomes "if $ 10000 is invest at 6 % interest compound annual". Now, all the elements in the problem statement are known to the system.

The problem statement is divided into several blocks in order to distribute the complexity of the problem (Figure 5). Each block is a knowledge unit that contains a domain variable and a numeric value. The knowledge unit contains necessary as well as unnecessary information for

interpreting the problem statement. Any unnecessary word such as 'at' must be removed before reasoning, because they only required overhead on the system during the process of reasoning.

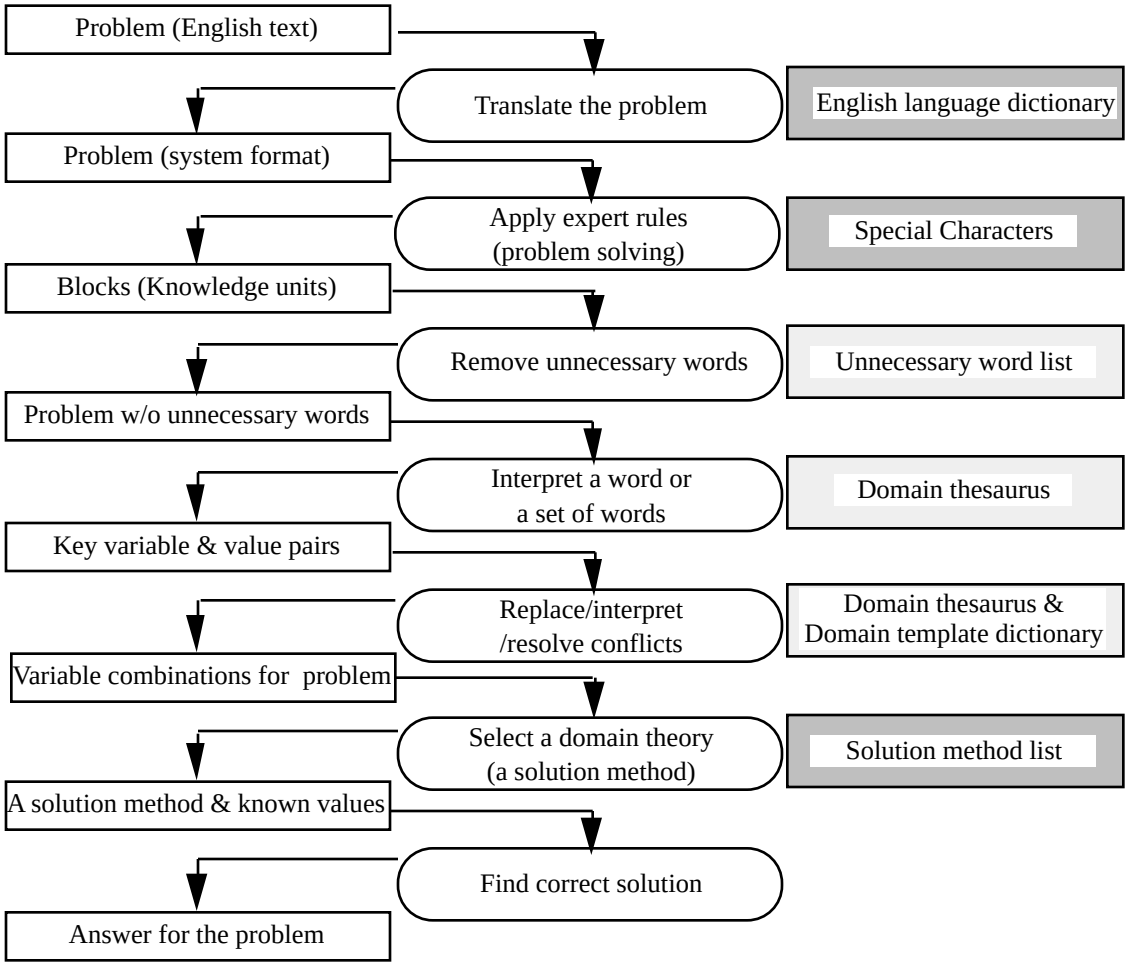


Figure 2. problem solving process

	How much money will you have accumulated three years from now if $10,000 is invested at 6% interest compounded annually?

Figure 3. A sample problem in engineering economics

	Plural to singular:	years => year
	past tense to present tense:	invested => invest compounded => compound accumulated => accumulate
	Upper case to lower case:	How => how
	verbal number to numeric number:	three => 3
	remove comma within a number:	10,000 => 10000
	separate symbol from number:	$10000 => $ 10000 6% => 6 %

Figure 4. Conversion process

Block (Knowledge unit)	Unnecessary word
1. how much money will you have accumulate	you have
2. three year from now	
3. if $10,000 is invest	if, is
4. at 6% interest compound annually	at

Figure 5. Removing unnecessary words from each knowledge unit

Knowledge unit	Interpretation
1. how much money will … accumulate	Find F
2. 3 year from now	N = 3
3. … $ 10000 … invest	P = 10000
4. … 6 % interest compound annual	i = 6%

Figure 6. Knowledge Units of the Sample Problem

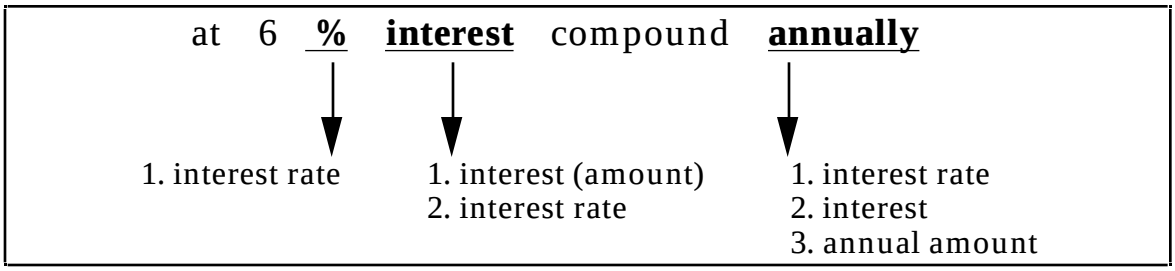


Figure 7. Domain Thesaurus Interpretation Example

Given P, i, N, and Find F. Solution strategy is F = P (F/ P, i%, N))

Figure 8. Finding a Solution Strategy

As an instance, a block “if $10000 is invest” will be interpreted as a present worth “P = $10000” because ‘if’ and ‘is’ are unnecessary, ‘invest’ is used previously as present worth, and ‘$10000’ is a value of the variable. Unnecessary words can be found in all problems in the domain, but not in the list of domain templates. A domain template is a sequence of words, a knowledge unit, that is used to interpret a domain variable. The unnecessary word is not used uniquely: it could be found in the templates for all different variables.

The Problem Solver interprets each knowledge unit by applying the domain thesaurus and domain template dictionary. For example, when a text block, ‘at 6% interest compound annual’ is given to the system, the knowledge base provides interpretation of the block: 1) word by word: The word 'at' is an unnecessary word for solving the problem. Next word, '%', will be interpreted as 'interest rate,' and 'compound' as 'interest rate.' The word 'interest' has two meanings: 'interest' (amount of money) and ‘interest rate’ (rate). The last word 'annual' could be represented in three different variables: 'interest,' 'interest rate,' and 'annual value' (Figure 7). 2) as a template 'interest compound annual' meaning 'interest rate.' Such conflicts will be resolved by selecting an interpretation with the highest priority among all different possibilities. The knowledge base provides the necessary knowledge to determine which one has higher priority.

The Problem Solver sends the interpretation of the problem statement to the domain theory selector. The interpretation of the problem (for example, P = $10000, i = 6%, N = 3 year, and F is

unknown) is used to select an appropriate solution method (F = P (F/ P, i%, N)) (Figure 8). The system applies the interpretation of the problem to the solution method (F = 10000 (F/ P, 6%, 3)). The solution found is presented to the user through the User Interface.

CONCLUSION

The Translator and the Problem Solver in the I^3 system have demonstrated that the knowledge based interpretation of natural language is feasible. Modular design of the Problem Solver provides the system's expandability and reusability. Expanded problem solving capability of the system can be accomplished by adding more knowledge to the Domain Glossary. Reusability can be enhanced by replacing or adding managers to the Problem Solver without reprogramming other parts of the system. Combining rule based processing with objects (or an integration of object oriented system with an intelligent system) makes it possible to define domain knowledge about the application further than with rules alone.

The I^3 system is being developed on an IBM compatible 486 machine using the C/C++ programming language (Microsoft Visual C++) and CLIPS 6 (C Language Integrated Production System, by NASA Lyndon B. Johnson Space Center, a forward chaining expert system shell based on the Rete algorithm).

REFERENCE

1. Biegel, John, "I^3: Intelligent Individualized Instruction," *Proceedings of the 1993 International Simulation Conference*, San Francisco, CA., OMNIPRESS, Madison, Wisconsin, November, 1993, 243-249.

2. Chung, Minhwa, and Moldovan, Dan, "Applying Parallel Processing to Natural-Language Processing." *IEEE Expert Intelligent Systems & Their Applications*, IEEE Computer Society, Vol. 9, Number 1, February, 1994, 36-44.

3. Martin, Charles, "Case-based Parsing," *Inside Case-Based Reasoning*, Riesbeck Christopher K. and Schank, Roger C., Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1989, 319-352.

4. Lenat, Douglas, and Feigenbaum, Edward, "On the Thresholds of Knowledge," *Applications of Expert Systems*, Vol. 2, Ed. Quinlan, J. Ross, Addison-Wesley Publishing Company, Sydney, Australia, 1989, 36-75.

MIRO: A DEBUGGING TOOL FOR CLIPS INCORPORATING HISTORICAL RETE NETWORKS

Sharon M. Tuttle
Management Information Systems, University of St. Thomas
3800 Montrose Blvd., Houston, TX, 77006

Christoph F. Eick
Department of Computer Science, University of Houston
Houston, TX 77204-3475

ABSTRACT

At the last CLIPS conference, we discussed our ideas for adding a temporal dimension to the Rete network used to implement CLIPS. The resulting historical Rete network could then be used to store 'historical' information about a run of a CLIPS program, to aid in debugging. MIRO, a debugging tool for CLIPS built on top of CLIPS, incorporates such a historical Rete network and uses it to support its prototype question-answering capability. By enabling CLIPS users to directly ask debugging-related questions about the history of a program run, we hope to reduce the amount of single-stepping and program tracing required to debug a CLIPS program. In this paper, we briefly describe MIRO's architecture and implementation, and the current question-types that MIRO supports. These question-types are further illustrated using an example, and the benefits of the debugging tool are discussed. We also present empirical results that measure the run-time and partial storage overhead of MIRO, and discuss how MIRO may also be used to study various efficiency aspects of CLIPS programs.

INTRODUCTION

In debugging programs written in a forward-chaining, data-driven language such as CLIPS, programmers often have need for certain *historical* details from a program run: for example, when a particular rule fired, or when a particular fact was in working memory. In a paper presented at the last CLIPS conference [4], we proposed modifying the Rete network, used for determining which rules are eligible to fire at a given time, within CLIPS, to retain such historical information. The information thus saved using this *historical Rete network* would be used to support a debugging-oriented question-answering system.

Since the presentation of that paper, we have implemented historical Rete and a prototype question-answering system within MIRO, a debugging tool for CLIPS built on top of CLIPS. MIRO's question-answering system can make it much less tedious to obtain historical details of a CLIPS program run as compared to such current practices as rerunning the program one step at a time, or studying traces of the program. In addition, it turns out that MIRO may also make it easier to analyze certain efficiency-related aspects of CLIPS program runs: for example, one can much more easily determine the number of matches that occurred for a particular left-hand-side condition in a rule, or even the number of matches for a subset of left-hand-side conditions (those involved in *beta memories* within the Rete network).

The rest of the paper is organized as follows. Section two briefly describes MIRO's architecture and implementation. Section three then gives the currently-supported question-types, and illustrates how some of those question-types can be used to help with debugging. Empirical results regarding MIRO's run-time and partial storage overhead costs are given in section four, and section five discusses some ideas for how MIRO might be used to study various efficiency aspects of CLIPS programs. Finally, section six concludes the paper.

MIRO'S ARCHITECTURE AND IMPLEMENTATION

To improve CLIPS' debugging environment, MIRO adds to CLIPS a question-answering system able to answer questions about the current CLIPS program run. We used CLIPS 5.0 as MIRO's basis. Figure 1 depicts the architecture of MIRO. Because questions useful for debugging will often refer to historical details of a program run, MIRO extends the CLIPS 5.0 inference engine to maintain historical information about the facts and instantiations stored in the working memory, and about the changes to the agenda. Moreover, in order to answer question-types we provided query-operators that facilitate answering questions concerning past facts and rule-instantiations, and an agenda reconstruction algorithm that reconstructs conflict-resolution information from a particular point of time.

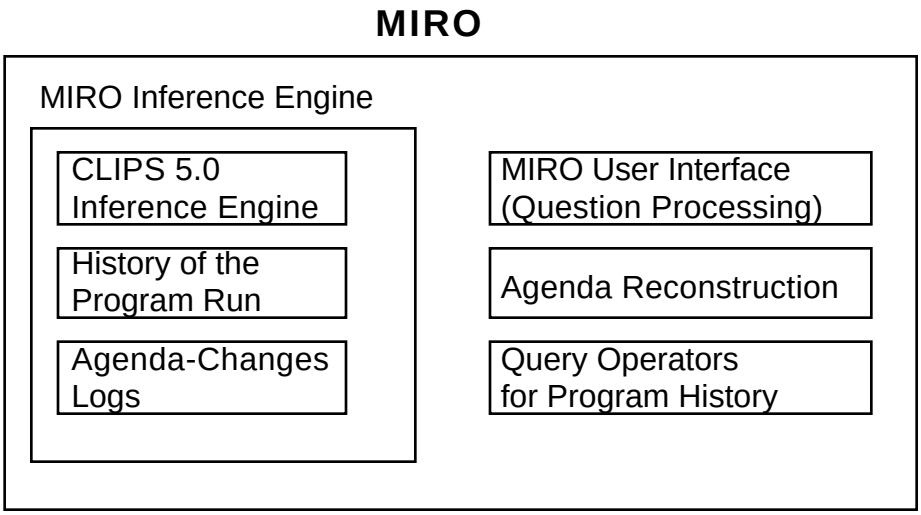


Figure 1. The MIRO Debugging Environment

One could describe MIRO as a tool for helping programmers to analyze a program run; it assists them by making the acquisition of needed low-level details as simple as asking a question. Where, before, they gathered clues that might suggest to them a fault's immediate cause by searching traces and single-stepping through a program run, now they can simply ask questions to gather such clues. The programmers still direct the debugging process, but the question-answering system helps them to determine the next step in that process. By allowing programmers to spend less time single-stepping through program runs and searching traces for historical details, this question-answering system can let programmers save their attention and energy for the high-level aspects and intuition needed in debugging.

As already mentioned, CLIPS 5.0 forms the basis of MIRO: the CLIPS 5.0 source code was modified and augmented to implement MIRO. To quickly obtain an operational prototype, we used existing code whenever possible, and we patterned the historical Rete and agenda reconstruction additions after those used for regular Rete within CLIPS. This software reuse included replicating the code for existing data structures when creating new data structures, augmenting existing data structures, calling existing functions from new functions whenever possible, and modifying existing functions when their functionality was almost, but not quite, what was needed.

So, when implementing, for example, historical Rete's past partitions, the past partitions were patterned after the current (formerly only) partitions. The data structures for facts, instantiations, and rule instantiations were all augmented with time-tags, and rule instantiations were also

augmented with a fired flag, set if that rule instantiation was actually fired. We added analogous functions to those used for maintaining the current working memory for maintaining the past working memory, and so on. This approach reduced programming overhead, and, because CLIPS 5.0 was already quite efficient, the added historical components were also quite efficient.

We also added code to measure various run-time characteristics, such as the number of current and past facts, rule instantiations, and Rete memory instantiations, to better compare program runs under regular CLIPS and MIRO, as shown at the end of section four.

The bulk of MIRO was implemented over the course of a single year, by a single programmer; however, other research-related activities were being done concurrently with this development. It probably took about seven programmer-months to bring MIRO to the point where it had 19 question-types. Note, however, that this does not take into account the time spent designing the historical Rete and agenda reconstruction algorithms.

The version of CLIPS 5.0 that we developed MIRO from, and that we used for comparisons with MIRO, contained 80497 lines of code and comments; this includes the source code files, include files, and the UNIX make file. The same files for MIRO contained 86099 lines, also including comments; so, we added about 5600 additional lines of code and comments, making MIRO about 7% larger than CLIPS 5.0.

MIRO'S QUESTION TYPES

Adding the question-answering system itself to MIRO was as easy as adding a new user-defined function to the top-level of CLIPS; we constructed a CLIPS top-level function called askquestion. The difficult part was determining the form that this question-answering would take. Our primary goals of constructing a prototype both to demonstrate the feasibility of, and to illustrate how, the information from the historical Rete network and other history-related structures could be used to answer programmers' debugging-related questions had a strong impact on the design that we decided to use.

We assume that the kinds of questions that can be asked are limited to instances of a set of fixed question-types; each question-type can be thought of as a template for a particular kind of question. This allows the question-answering system to have a modular design: each question-type has an algorithm for answering an instance of that type. This also allows additional question-types to be easily added, and to be tested as they are added, as it is discovered which are desirable for debugging.

The design of the interface for getting programmers' questions to the explanation system is a worthy research topic all by itself; to allow us to concentrate more on answering the questions, we use a very simple interface, with the understanding that future work could include replacing this simple interface with a more sophisticated front-end. Since we are more interested in designing a tool for debugging and less interested in natural language processing, the question-answering system uses a template-like approach for the various question-types that the programmers will be able to ask. That is, each question-type has a specific format, with specific ``blanks'' which, filled in, result in an instance of that question-type. Furthermore, to avoid requiring the programmers to memorize these question-type formats, we use a menu-based approach: when the programmers enter the command (askquestion), a list of currently-supported question-types is printed on-screen. They then enter the number of the desired question-type, and the explanation system queries them to fill in that question-type's necessary blanks.

Implementing this approach was quite straightforward, because regular CLIPS already has some tools for obtaining top-level command arguments. We only had to modify them a little to allow

for the optional askquestion arguments. A main askquestion function prints the menu if no arguments are given, and then asks which question-type is desired; a case statement then uses either that result or the first askquestion argument to call the appropriate question-answering function, which is in charge of seeing if arguments for its needed values have already been given, or must be asked for. After each particular question-type's question-answering function obtains or verifies its needed values, it then tries to answer the question, and print the result for the programmer.

We currently support 19 question-types, as shown in Figure 2. However, question 9, why a particular rule did not fire, currently only tells if that rule was eligible or not at the specified time.

1. What fact corresponds to fact-id <num>?
2. When did rule <name> fire?
3. What rule fired at time <num>?
4. What facts were in memory at time <num>?
5. How many current facts are in memory now?
6. How many past facts are in memory now?
7. How many current rule activations are on the agenda now?
8. How many past rule activations are in memory now?
9. Why did rule <name> not fire at time <num>?
10. How many current alpha instantiations are in memory now?
11. How many past alpha instantiations are in memory now?
12. How many current beta instantiations are in memory now?
13. How many past beta instantiations are in memory now?
14. What are the Rete memory maximums?
15. What were the agenda changes from time <num> to time <num>?
16. How many current not-node instantiations are in memory now?
17. How many past not-node instantiations are in memory now?
18. What was the agenda at the end of time <num>?
19. How many agenda changes were there from time <num> to time <num>?

Figure 2. Currently-Supported Question-Types in MIRO

We will now give some examples of MIRO's question-answering system at work. We will describe some hypothetical scenarios, to illustrate how MIRO might be useful in debugging; the responses shown are those that would be given by MIRO in such situations.

Consider a program run in which the program ends prematurely, that is, without printing any output. One can find out the current time-counter value with a command that we added to MIRO specifically for this purpose --- if one types (time-counter) after 537 rule-firings, it prints out:

```
time_counter is: 537
```

The programmers can now ask, if desired, which rules fired at times 537, 536, 535, etc. If they type (askquestion), the menu of rules will be printed; if they choose question-type number 3, ``What rule fired at time <num>?", then it will ask what time-counter value they are interested in; if 537 is entered, and if a rule named ``tryit" happened to be the one that fired at that time, then MIRO would print an answer like:

```
Rule tryit fired at time 537
```

with the following rule activation:

```
0     tryit: f-30,f-15,f-47 time-tag: (530 *)(activn time-tag: 530 537))
```

This tells the programmers that rule tryit fired at time 537, and that the particular instantiation of rule tryit that fired had LHS conditions matched by the facts with fact-identifiers f-30, f-15, and f-47. This instantiation of rule tryit has been eligible to fire since time 530 --- before the 531st rule firing --- but, as shown, the rule instantiation's, or activation's, time-tag is now closed, with the time 537, because it was fired then, and a rule that is fired is not allowed to fire again with the same fact-identifiers.

Now, if the programmers suspect that this rule-firing did not perform some action that it should have performed --- to allow another rule to become eligible, for example --- then they can use the regular CLIPS command ``pprule" to print the rule, so that they can examine its RHS actions. If it should not have fired at all, then they may wish to see why it was eligible. For example, in this case, they may want to know what facts correspond to the fact-identifiers f-30, f-15, and f-47. One can look at the entire list of fact-identifiers and corresponding facts using the regular CLIPS (facts) command, but if the program has very many facts, it can be quite inconvenient to scroll through all of them. So, MIRO provides the question-type ``What fact corresponds to fact-id <num>?". On first glance, this question appears to have no historical aspect at all; however, it does include the time-tag for the instance of the fact corresponding to this fact-identifier. This can be helpful to the programmers, if they suspect that one of the facts should not have been in working memory --- then, the opening time of that fact's time-tag can be used to see what rule fired at that time, probably resulting in this fact's addition. Since this question-type is the first in the list, and requires as its only additional information the number of the fact identifier whose fact is desired, typing (askquestion 1 47) will ask the first question for fact-identifier f-47, giving an answer such as:

```
Fact-id 47 is fact:
   (p X Y) time-tag: (530 *)
```

If the programmers suspect that fact f-47, now known to be (p X Y), should not be in working memory --- if they think that it is a fault that it exists, and is helping rule tryit to be able to fire --- then they can again ask the question-type ``What rule fired at time <num>?" to see what rule fired at time 530, when this instance of (p X Y) joined working memory. They can then see if the rule that fired at time 530 holds the cause of the fault of (p X Y) being in working memory, and enabling the faulty firing of tryit at time 537.

COMPARISONS BETWEEN MIRO AND CLIPS 5.0

As already mentioned, we implemented MIRO by starting with CLIPS 5.0; we then generalized its Rete inference network [2] into a *historical Rete* network, added an agenda reconstruction capability, and added the prototype question-answering capability. Historical Rete and agenda reconstruction are discussed in more detail in [5] and [6]. We also made some other modifications, to allow for experimental measures; for example, we added code to measure various run-time characteristics such as the number of current and past instantiations, and the number of current and past facts. We then ran a number of programs under both MIRO and CLIPS 5.0.

The programs that we used range fairly widely in size, and behavior. Four of the programs --- dilemma1, mab, wordgame, and zebra --- are from CLIPS 5.0's Examples directory. The dilemma1 program solves the classic problem of getting a farmer, fox, goat, and cabbage across a stream, where various constraints must be met. The mab program solves a planning problem in which a monkey's goal is to eat bananas. The wordgame program solves a puzzle in which two six-letter names are ``added" to get another six-letter name, and the program determines which digits correspond to the names' letters. Finally, the zebra program solves one of those puzzles in which five houses, of five different colors, with five different pets, etc., are to each have their specific attributes determined, given a set of known information.

The AHOC program was written by graduate students in the University of Houston Department of Computer Science's graduate level knowledge engineering course COSC 6366, taught by Dr. Eick in Spring 1992. AHOC is a card game with the slightly-different objective that the players seek to win *exactly* the number of tricks bid. The program weaver ([1], [3]) is a VLSI channel and box router; we obtained a CLIPS version of this program from the University of Texas' OPS5 benchmark suite. Finally, can_ordering_1 is a small program that runs a rather long canned beverage warehouse ordering simulation, also from the Spring 1992 COSC 6366 knowledge engineering class; it was written by C. K. Mao.

We ran each program three times under MIRO and under CLIPS 5.0, on a Sun 3 running UNIX, with either no one else logged in, or one other user who was apparently idle. For every run in both CLIPS and MIRO, we used the (watch statistics) command to check that the same number of rules fired for each run of the program; for every MIRO run of a program, we also made sure that all runs had the same number of instantiations at the end of the run. The run-times are given in Table 1.

The run-times for programs run using MIRO were usually only slightly slower than those using regular CLIPS 5.0; one program, mab, took 11.4% more time in MIRO, but on average, the MIRO runs only took 4.1% more time. Interestingly enough, AHOC ran, on average, slightly faster under MIRO than under regular CLIPS. This could be because regular CLIPS 5.0 returns the memory used for facts and instantiations to available memory as they are removed, which MIRO does not do until a reset or clear, because such facts and instantiations are instead kept, and moved into the past fact list or past partitions. The average 4.1% additional time required by MIRO to run a program seems quite reasonable, especially since the additional time is only required while debugging, and allows programmers the benefits of the MIRO tool.

Another feature of the (watch statistics) command under regular CLIPS, besides printing the number of rules fired and the time elapsed, is that it also prints the average and maximum number of facts and rule instantiations during those rule-firings. We enhanced this command in MIRO so that it also keeps track of the average and maximum number of past facts and past rule instantiations, as well as the averages and maximums for different kinds of Rete memory instantiations, both past and current. To get some idea of the overhead needed to store historical information from a run, we compared the average number of current facts during a run to the average number of past facts, and compared the average number of current rule instantiations to the average number of past rule instantiations.

[Table Deleted]

Table 1. Run-Times

Another feature of the (watch statistics) command under regular CLIPS, besides printing the number of rules fired and the time elapsed, is that it also prints the average and maximum number of facts and rule instantiations during those rule-firings. We enhanced this command in MIRO so that it also keeps track of the average and maximum number of past facts and past rule instantiations, as well as the averages and maximums for different kinds of Rete memory instantiations, both past and current. To get some idea of the overhead needed to store historical information from a run, we compared the average number of current facts during a run to the average number of past facts, and compared the average number of current rule instantiations to the average number of past rule instantiations. (Analogous comparisons for different kinds of Rete memory instantiations, as well as comparisons of the maximum number of current items to the maximum number of past items, can be found in [6]).)

Table 2 shows these averages for facts and rule instantiations. It also gives, where appropriate, how many times bigger the average number of past items is than the average number of current

items. Compared to the average number of current items during a run, fewer times as many past facts than past rule instantiations will need to be kept. This suggests that there is more overhead to storing rule instantiation history than there is to storing fact history.

Table 2 shows that the space to store past facts and rule instantiations, compared to the average space to store current facts and rule instantiations, can, indeed, be high, especially for long runs. But, long runs should be expected to have more historical information to record. Also, it should be noted that in running these programs on a Sun 3, we never ran into any problems with space, even with the additional historical information being stored. During program development, the storage costs should be acceptable, since they facilitate debugging-related question-answering. The programmers can also limit the storage costs by only running their program using MIRO when they might want to obtain the answers to debugging-related questions; at other times, they can run their program using regular CLIPS.

USING MIRO IN STUDYING CLIPS PROGRAMS

We hope that MIRO's question-answering system can make debugging a CLIPS program easier and less tedious. Here, we consider another use of MIRO: to analyze certain performance aspects of CLIPS programs. With a shift of viewpoint, such analysis may be involved in a variant of debugging—if, for example, a program takes so long to run that the programmers consider the run-time to be a problem, then such performance analysis may help in determining how to modify the program so that it takes less time. Such aspects may also be of interest in and of themselves, both to programmers and to researchers studying the performance aspects of forward-chaining programs in general.

[Table Deleted]

Table 2. Average No. of Facts and Rule Instantiations

Note that a Rete network, historical or regular, encodes a program's rules' LHS conditions; the RHS conditions of those rules are not represented, except perhaps by pointers from a rule's final beta memory to its actions, to help the forward-chaining system to more conveniently execute the RHS actions of a fired rule. So, MIRO could be helpful primarily in analyzing the efficiency involved in the match step of the recognize-act cycle. Being able to locate Rete memories whose updating could cause performance trouble-spots could be useful for improving the overall performance of a CLIPS program.

Using MIRO, it is much easier to discover some of the dynamic features of a CLIPS program run, such as the number of instantiations within the network during a run. One can study worst-case and average-case behavior within a CLIPS program run by looking at the number of average, and maximum, facts, rule instantiations, and alpha, beta, and not-memory instantiations. For example, a great disparity between the average number of current beta instantiations, and the maximum number reached during a run could indicate volatility in beta memory contents that could have a noticeable performance impact.

One might consider the total number of changes to a beta memory during a run to be the total number of additions to and deletions from that memory --- or, the total number of current instantiations at the end plus two times the number of past instantiations. Averaged over the number of rule-firings, this would give the number of beta memory changes per rule-firing, which, if high, might very well correlate with more time needed per rule-firing; and, averaged over the number of total beta memories, this would give us a rough average of the number of changes per beta memory. We could even determine a number of working memory changes this

way, by adding the number of current facts to two times the number of past facts, and use this to obtain an average number of fact changes per rule firing.

Another measure that might be very telling would be the average number of memory changes per rule-firing, computed by counting each past instantiation at the end of a run as two memory changes—since each was first added to a current instantiation, and then moved to a past partition—and each current instantiation at the end of a run as a single memory change, and then dividing the sum by the number of rule-firings. We can also determine the average number of fact changes per rule-firing similarly. In measurements we made using the seven programs discussed in the previous section, the most important factor that we found that correlated with performance was that a high number of instantiation changes per rule-firing did seem to correlate with more time needed per rule-firing [6].

Although one can reasonably obtain some of the information mentioned above by using regular CLIPS, much of it would be very inconvenient to gather using it. For example, determining the average and maximum number of past facts and past rule instantiations would be difficult to obtain using regular Rete. We modified the existing CLIPS (watch statistics) command in MIRO so that it also keeps track of this additional information.

Note that historical Rete can also be used to support trouble-shooting tools and question-types, in addition to supporting question-answering for debugging. For example, any time that the programmer can specify a particular time of interest, historical Rete searches can be made that focus only on instantiations in effect at that time.

The discussed examples show the potential that MIRO has as a tool in analyzing CLIPS program performance, as well as in debugging. Information about what occurred during Rete network memories can be more reasonably retrieved, making such analysis more practical, across larger samples of programs. Note, too, that a programmer can choose to look at averages over all of a program's memories, or for a single memory, or for a particular rule's memories, as desired.

CONCLUSIONS

In this paper, we have followed up on our work reported at the last CLIPS conference, describing how we have implemented historical Rete and question-answering for debugging in MIRO, a debugging tool built on top of CLIPS 5.0. We have described MIRO, and have hopefully given a flavor for how it may be used to make debugging a CLIPS program easier and less tedious, by allowing programmers to simply ask questions to determine when program events --- such as rule firings, or fact additions and deletions --- occurred, instead of having to depend on program traces or single-stepping program runs. We have further described how MIRO might be used to study certain performance-related aspects of CLIPS programs.

The empirical measures included also show that MIRO's costs are not unreasonable. Comparisons of programs run in both MIRO and CLIPS 5.0, which MIRO was built from, have been given; on average, the run-time for a program under MIRO was only 4.1% slower than a program run under CLIPS 5.0, when both were run on a Sun 3. Comparing the average number of past facts and rule instantiations to the average number of current facts and current rule instantiations, there were, on average, 3.53 times more past facts than current facts, and 72.12 times more past rule instantiations than current rule instantiations. But this historical information permits the answering of debugging-related questions about what occurred, and when, during an expert system run. We also gave examples of the kinds of question-types that MIRO can currently answer, as well as examples of the kinds of answers that it gives. We hope that this research will encourage others to also look into how question-answering systems can be designed to serve as tools in the development of CLIPS programs.

REFERENCES

1. Brant, D. A., Grose, T., Lofaso, B., Miranker, D. P., "Effects of Database Size on Rule System Performance: Five Case Studies," *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, 1991.

2. Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artifical Intelligence* , Vol. 19, September, 1982, pp. 17-37.

3. Joobbani, R., Siewiorek, D. P., "WEAVER: A Knowledge-Based Routing Expert," *IEEE Design and Test of Computers*, February, 1986, pp. 12-23.

4. Tuttle, S. M., Eick, C. F., "Adding Run History to CLIPS," *2nd CLIPS Conference Proceedings*, Vol. 2, Houston, Texas, September 23-25, 1991, pp. 237-252.

5. Tuttle, S. M., Eick, C. F., "Historical Rete Networks to Support Forward-Chaining Rule-Based Program Debugging," *International Journal on Artificial Intelligence Tools*, Vol. 2, No. 1, January, 1993, pp. 47-70.

6. Tuttle, S. M., "Use of Historical Inference Networks in Debugging Forward-Chaining Rule-Based Systems," Ph.D. Dissertation, Department of Computer Science, University of Houston, Houston, Texas, December, 1992.

OPTIMAL PATTERN DISTRIBUTIONS IN RETE-BASED PRODUCTION SYSTEMS

Stephen L. Scott

Hughes Information Technology Corporation
1768 Business Center Drive, 4th Floor
Reston, VA 22090
(703) 759-1356

sscott@mitchell.hitc.com

ABSTRACT

Since its introduction into the AI community in the early 1980's, the Rete algorithm has been widely used. This algorithm has formed the basis for many AI tools, including NASA's CLIPS. One drawback of Rete-based implementations, however, is that the network structures used internally by the Rete algorithm make it sensitive to the arrangement of individual patterns within rules. Thus while rules may be more or less arbitrarily placed within source files, the distribution of individual patterns within these rules can significantly affect the overall system performance. Some heuristics have been proposed to optimize pattern placement, however, these suggestions can be conflicting.

This paper describes a systematic effort to measure the effect of pattern distribution on production system performance. An overview of the Rete algorithm is presented to provide context. A description of the methods used to explore the pattern ordering problem area are presented, using internal production system metrics such as the number of partial matches, and coarse-grained operating system data such as memory usage and time. The results of this study should be of interest to those developing and optimizing software for Rete-based production systems.

INTRODUCTION

The Rete algorithm was developed by Charles Forgy at Carnegie Mellon University in the late 1970's, and is described in detail in [Forgy, 1982]. Rete has been used widely in the expert system community throughout the 1980's and 1990's, and has formed the basis for several commercial and R&D expert system tools [Giarratano & Riley, 1989] [ILOG, 1993]. Recent enhancements have been proposed based on parallel processing [Miranker, 90] and matching enhancements [Lee and Schor, 1992]. Rete provides an efficient mechanism for solving the problem of matching a group of facts with a group of rules, a basic problem in a production system.

In this section, an overview of the Rete algorithm is given in order to provide context for the discussion to follow. This presentation, however, is not intended to be a rigorous analysis of the Rete algorithm.

Rete based systems assume a working memory that contains a set of facts and a network of data structures that have been compiled from rule definitions. The rules contain a set of condition elements (CE's) that form the left-hand-side (LHS), and a right-hand-side (RHS) that performs actions. The RHS actions may be side-effect free, such as performing a computation, invoking an external routine, performing I/O to the input or output streams or file. Other actions on the RHS may cause changes in the working memory, such as insertions, deletions, or modifications of facts. The Rete network actually contains two main structures: a pattern network, and a join network. The pattern network functions to identify which facts in working memory are

associated with which patterns in the rules. The join network is used to identify which variables are similarly bound within a rule across CE's.

Within the pattern network, elements of the individual CE's are arranged along branches of a tree, terminating in a leaf node that is called an alpha node. The join network consists of groups of beta nodes, each containing two nodes as inputs and one output that can be fed to subsequent beta nodes. Finally, the output of the join network may indicate that one or more rules may be candidates for firing. Such rules are called activations, and constitute inputs to the conflict set, which is a list of available rules that are ready for execution. Typically, some arbitration mechanism is used to decide which rules of equal precedence are fired first. When a rule fires, it may of course add elements to or delete elements from the working memory. Such actions will repeat the processing cycle described above, until no more rules are available to be fired.

Consider the following small set of facts and a rule. For simplicity, additional logical constructs, such as the TEST, OR, or NOT expressions are not considered, and it is assumed that all CE's are ANDed together, as is the default. Note that myRule1 has no RHS, as we are focusing only on the LHS elements of the rule.

```
(deffacts data
   (Group 1 2 3)
   (Int 1)
   (Int 2)
   (Int 3))

(defrule myRule1
   (Group ?i ?j ?k)
   (Int ?i)
   (Int ?j)
   (Int ?k)
   =>)
```

This rule can be conceptualized in a Rete network as follows (see Figure 1). There are two branches in the pattern network, corresponding to the facts that begin with the tokens "Group" and "Int", respectively. Along the "Group" branch of the tree, there are nodes for each of the tokens in the fact, terminating with an alpha node that contains the identifier "f-1" corresponding to the first fact in the deffacts data defined above. Similarly, along the "Int" branch, there is one node for all the facts that have "Int" as a first token, and then additional nodes to show the various values for the second token. Alpha nodes along this branch also contain references to the appropriate facts that they are associated with, numbered in the diagram as "f-2" through "f-4". Note that the "Int" branch has shared nodes for structurally similar facts, i.e. there is only on "Int" node even though there are three facts with "Int" as a first token.

On the join network, myRule1 has three joins to consider. The first CE of myRule1 requires a fact consisting of a first token equal to the constant "Group" followed by three additional tokens. The alpha node of the "Group" branch of the pattern network supplies one such fact, f-1. The second CE of myRule1 requires a fact consisting of a first token equal to the constant "Int" followed by another token, subject to the constraint that this token must be the same as the second token of the fact satisfying the first CE. In this case, the fact f-2 meets these criteria, hence the join node J1 has one partial activation. This is because there is one set of facts in the working memory that satisfy its constraints. Continuing in this fashion, the output of J1 is supplied as input to J2, which requires a satisfied join node as a left input and a fact of the form "Int" followed by a token (subject to the constraint that this token must be equal to the third token of the first CE). The fact f-3 meets these criteria, so join node J2 has one partial activation as well. This process continues until we finish examining all CE's in myRule1 and determine that there are indeed facts to satisfy the rule. The rule is then output from the join network with

the set of facts that satisfied its constraints and sent on to the agenda, where it is queued up for execution.

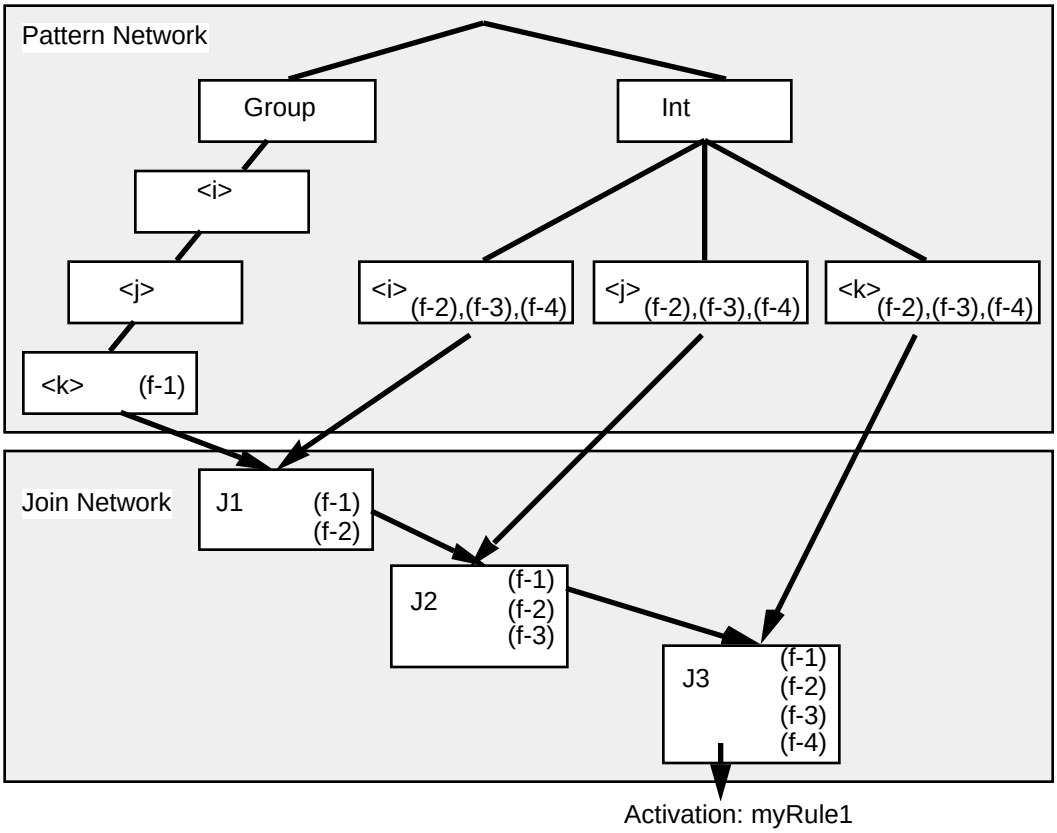


Figure 1. Rete Network for myRule1. This diagram depicts the Rete pattern and join networks for a rule with four CE's.

Consider the same set of data and a rule with the CE's arranged in a different order. Semantically, myRule1 and myRule2 are the same, however, the number of partial matches generated by myRule2 is much greater than that generated by myRule1.

```
(defrule myRule2
   (Int ?i)
   (Int ?j)
   (Int ?k)
   (Group ?i ?j ?k)
   =>)
```

With this rule, we have three facts that match the first CE. Examining the first two CE's, there are three possible facts that can match the second CE, hence there are nine possible ways to satisfy the first two CE's. Moving to the third CE, there are again three ways to satisfy the third CE, but each of these must be considered with the nine possibilities that preceded it, hence there are 27 possible ways to satisfy the first three CE's. Fortunately, the fourth CE is satisfied by only one fact, so the number of partial activations for CE's one through four is only one; it is the fact that matches the fourth CE (f-1), coupled with exactly one set of the 27 possibilities available for CE's one through three. Summarizing, there are 40 partial activations for this rule (3 + 9 + 27 + 1).

From the above discussion, we have seen that pattern ordering in the LHS of rules can have significant impact on performance. Unfortunately, there are a large number of possible orderings one can try in even a small rule. Since in general, in a rule with N CE's, there are N ways to place the first CE, N-1 ways to place the second CE, and so on, the number of possible pattern arrangements is given by N!. As there may be many rules in an expert system, each with a large number of possible CE orderings, it should be clear that it is prohibitively expensive to do an exhaustive search of all possible arrangements of all rules in an attempt to optimize performance.

There may be some reduction in the number of arrangements if one considers that from the pattern network point of view, some arrangements produce an equal number of partial activations and thus can be considered together for analysis purposes. For example, the rule aRule1

```
(defrule aRule1
   (Group ?i ?j)
   (Int ?i)
   (Int ?j)
   => )
```

has the same number of partial activations as aRule2,

```
(defrule aRule2
   (Group ?i ?j)
   (Int ?j)
   (Int ?i)
   => )
```

because the CE's in slots 2 and 3 are similar with respect to effect on the join network. So even though there are 6 possible arrangements of CE's in this rule, only 3 actually produce different numbers of partial activations in the join network. This property is used extensively in the Analysis section that follows, as it allows valid results to be obtained using a manageable subset of the possible pattern orderings.

ANALYSIS AND RESULTS

In an effort to better understand the effects of pattern ordering on production system performance, a series of tests were conducted. This section describes the various experiments and the results obtained.

Partial Activations

From the discussion above, it is evident that some CE orderings are considerably more efficient than others with respect to partial activations. A test suite was developed using a rule with N CE’s and a data set of N facts, where N-1 CE’s are syntactically similar and one CE joins across the remaining N-1 CE’s. This is the configuration used in the example myRule1 and myRule2 above; note that in that case N=4 because there are four CE’s.

To interpret the data in the following table, match the number of CE’s in the rule LHS (identified by the row labeled N = <n>) with the position of the constraining fact (the fact that has elements to match all other CE’s in the rule). For the example myRule1 cited above, the row “N=4” is matched with “Fact Pos 1”, giving 4 partial activations. Similarly, the example myRule2 cited above has a constraining fact in position 4, hence for N=4, the number of partial activations is 40. The following table shows the results of the number of partial activations for rules with the number of CE’s varying from N=2 to N=8.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	2	2						
N = 3	3	4	7					
N = 4	4	6	14	40				
N = 5	5	8	23	86	341			
N = 6	6	10	34	158	782	3906		
N = 7	7	12	47	262	1557	9332	55987	
N = 8	8	14	62	404	2804	19610	137258	960800

Table 1. Partial Activations in Rule Sets. This table shows the increase in partial activations observed in rules with various numbers of CE's, where constraining fact is located at the position indicated by the column heading.

From this, at least two observations may be made. First, it is clear that the number of partial activations grows very rapidly. For this example set of rules and data, the number of partial activations for a rule with N CE's is given by

$$\sum_{i=0}^{N-1} (N-1)^i \qquad (1.0)$$

With such growth, on small computer systems, this may result in unexpected termination of a program, and even on large systems, performance may be degraded as the system attempts to accommodate the memory demands through paging or swapping. The second observation is the smaller the number of CE's on the LHS, the smaller the upper limit on partial activations. This suggests that a system with a larger number of smaller rules is better, at least from the vantage point of partial activations, than a system with a smaller number of larger rules.

Memory Usage

Within the Rete network implementation, data is maintained about partial activations. This data requires memory allocation, and as expected, the required memory grows in proportion to the number of partial activations. To examine this, the same suite of rules used above for partial activation testing was used, however, in this case, calls were made to the CLIPS internal function (mem-used) in order to calculate the memory required to store a network. The following table shows the results of these tests.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	376	376						
N = 3	548	560	560					
N = 4	596	620	620	960				
N = 5	836	872	872	1912	8008			
N = 6	960	1008	1008	3228	18180	105652		
N = 7	1016	1076	1076	5076	36132	253832	1746792	
N = 8	1292	1364	1364	7864	65440	536008	4300744	33947716

Table 2. Memory Requirements for Various Rule Sets. This table shows the increase in memory requirements observed in rules with various

numbers of CE's, where a constraining fact is located at the position indicated by the column heading. Memory allocation values are in bytes.

As expected, the amount of memory required to represent a rule varies in proportion to the number of partial activations. The two observations given for partial activations also hold here: some rule LHS orderings will require much less memory than others, and it is in general more memory efficient to have more small rules than a few large rules.

Reset Time

After rules and data are read into the system, the network must be updated to reflect the state required to represent these constructs. Data must be filtered through the network in order to determine facts are available, and comparisons must be made across CE's to determine which rules are eligible for firing. In order to investigate the time these processes take, the same test suite describe above was used, however, in this case, an operating system call was used to time the execution of the load and reset operations for the various rules. The "timex" command, available on many systems, gives operating system statistics about the real time, system time and user time required to execute a process. The following table shows the results of this test, giving real time in seconds, for the test suite.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	0.1	0.1						
N = 3	0.1	0.1	0.1					
N = 4	0.1	0.1	0.1	0.1				
N = 5	0.1	0.1	0.1	0.1	0.11			
N = 6	0.1	0.11	0.11	0.11	0.11	0.17		
N = 7	0.1	0.1	0.13	0.11	0.12	0.25	0.96	
N = 8	0.11	0.1	0.11	0.11	0.15	0.45	2.51	17.88

Table 3. Reset Time for Rule Sets. This table shows the increase in reset time observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading.

As the reset times do not grow as rapidly as N increases, these results suggest that reset time is not as great a consideration as memory or number of partial activations. Also the granularity of timex is only 1/100 of a second, making more precise measurements difficult.

Placement of Volatile Facts

One heuristic that has been proposed concerns the placement of volatile facts in a rule. In data sets where a particular type of pattern is frequently asserted or retracted (or modified if the tool supports this), it is best to put these patterns at the bottom of the LHS of the rule. A typical example is a control fact containing constants, typically used to govern processing phases. The justification given is that because Rete attempts to maintain the state of the system across processing cycles, by placing the volatile fact at the bottom of the LHS, Rete does not need to check most of the rest of the network and can realize some performance gain. To test this, the following scenario was used. The data set consisted of a set of facts of the form

```
(Int  val  <n>  isPrime Yes)
```

where <n> contained a prime number in the range 1 <= n <= 1000. A volatile counter fact of the form

```
(Counter <n>)
```

was used, where n again ranged from 1 <= n <= 1000. This fact was asserted and retracted for each value of n in the range. The rules to test whether or not n was prime were

```
(defrule isPrime
   (Int  val ?n   isPrime Yes)
   ?x <- (Counter ?n)
   =>
   (retract ?x)
   (assert (Counter =(+ ?n 1))
   (printout t ?n " is a prime " crlf))

(defrule notPrime
   (Int  val ?n   isPrime Yes)
   ?x <- (Counter ?ctrVal&:(!= ?n ?ctrVal))
   =>
   (retract ?x)
   (assert (Counter =(+ ?ctrVal 1))
   (printout t ?ctrVal " is not a prime " crlf))
```

The results below indicate run times in seconds for systems that searched for primes up to size K. The column 100, for example, indicates that primes between 1 and 100 were sought by using the volatile fact (Counter <n>) 100 times.

The example rules isPrime and notPrime given above correspond the rules used for the "volatile fact at bottom" row of the table. The "volatile fact at top" rules are virtually the same, except that the (Counter <n>) fact appears as the first CE instead of the second as illustrated above.

	100	250	500	750	1000
volatile fact at top	1.67	4.74	8.17	15.01	20.31
volatile fact at bottom	1.39	3.92	8.06	12.19	16.43

Table 4. Run Times for Rules with Volatile Facts. This table shows the differences in run times observed in rules with volatile facts placed at the top or bottom of the LHS. Times are in seconds.

This example shows that placing volatile facts at the bottom of a rule improves runtime performance, even for a small rule set and small amounts of data. The improvement is more obvious as the problem size grows, as the observed difference for K=100 is slight, whereas the difference for K=1000 is almost 4 seconds.

Placement of Uncommon Facts

Another heuristic suggests that facts that are relatively rare in the system should be placed first on the LHS. To test this, the following scenario was used. A data set contained three classes of facts: sensor facts, unit facts, and controller facts. These facts were distributed in the system in various proportions. Two rules were compared, one organized so that its CE's matched the distribution of the facts, and the other exactly opposite. In the following rules, rareFirst is tailored to perform well when the number of Ctrl facts is less than the number of Unit facts and the

number of Unit facts is less than the number of Sensor facts. Conversely, rareLast is not expected to perform as well under this arrangement of data.

```
(defrule rareFirst
   (Ctrl    Id ?cid  Status ?cStat)
   (Unit    Id ?uid  Ctrl ?cid  Status ?ustat  Value ?uVal)
   (Sensor  Id ?sid  Unit ?uid  Value ?sVal)
   =>)

(defrule rareLast
   (Sensor  Id ?sid  Unit ?uid  Value ?sVal)
   (Unit    Id ?uid  Ctrl ?cid  Status ?ustat  Value ?uVal)
   (Ctrl    Id ?cid  Status ?cStat)
   =>)
```

The following table shows the number of partial activations generated for these rules given various distributions of matching Ctrl, Unit, and Sensor facts. The nomenclature i:j:k indicates that there were i Ctrl facts, j Unit facts, and k Sensor facts.

	Ctrl:Unit: Sensor 3:10:20	Ctrl:Unit: Sensor 5:20:50	Ctrl:Unit: Sensor 10:50:100	Ctrl:Unit: Sensor 25:125:500	Ctrl:Unit: Sensor 50:200:1000
rarest fact at top	33	75	160	650	1250
rarest fact at bottom	60	150	300	1500	3000

Table 5. Partial Activations for Rules with Rare Facts. This table shows the differences in partial activations observed in rules with patterns that match rarest facts at the top or bottom of the LHS.

This test shows that placing less common facts at the top of the LHS reduces the number of partial activations for the rule. Another point is worthy of mention here: had the distribution of facts been different, rareLast might have outperformed rareFirst rule. This points out a potential problem, as attempting to optimize a system based on one set of data may not have optimal results on other sets of data. Given that expert systems are typically much more data driven than other forms of software, this kind of optimization may not be effective if the data sets vary widely.

CONCLUSIONS

This paper has described a number of tests performed to investigate the effects of pattern ordering on production system performance. The results have borne out widely held heuristics regarding pattern placement on the LHS of rules. The results have quantified various aspects of the problem of partial activation growth by measuring the number of partial activations, memory requirements, system reset and run time for a variety of pattern configurations.

In general, the conclusions that can be drawn are as follows. Partial activations can vary exponentially as a result of pattern ordering. This suggests that (1) rules should be written with some regard to minimizing partial activations, and (2) systems should use larger numbers of small rules rather than smaller numbers of large rules. The second suggestion helps to reduce the risk of having potentially large numbers of partial activations. The growth of partial activations as a result of pattern ordering affects memory requirements, and, to a lesser extent, reset time. As the number of partial activations increases, the memory required and the reset time also increase.

Placing patterns that match volatile facts at the bottom of a rule LHS improves run-time performance. Placing patterns that match the least common facts in a system at the top of a rule LHS reduces the number of partial activations observed. It may be difficult to use these methods in practice, however, since both of them depend on knowing the frequency with which certain facts appear in the system. In some cases, this may be readily apparent, but in other cases, especially where the form of the data may vary widely, these may not be practical. Long term statistical analysis of the system performance may be required to make use of these optimizations.

REFERENCES

1. "CLIPS Programmer's Guide, Version 6.0," JSC-25012, NASA Johnson Space Center, Houston, TX, June 1993.

2. "CLIPS User's Guide, Version 6.0," JSC-25013, NASA Johnson Space Center, Houston, TX, May 1993.

3. "ILOG Rules C++ User's Guide, Version 2.0," ILOG Corporation, 1993.

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

5. Giarratano, Joseph and Gary Riley, *Expert Systems: Principles and Programming*, PWS-Kent Publishing Company, Boston, MA, 1989.

6. Lee, Ho Soo, and Schor, Marshall, "Match Algorithms for Generalized Rete Networks," *Artificial Intelligence*, Vol 54, No 3, 1992, pg 249-274.

7. Miranker, Daniel, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.

8. Schneier, Bruce, "The Rete Matching Algorithm," *AI Expert*, December 1992, pg 24-29.

SIMULATION IN A DYNAMIC PROTOTYPING ENVIRONMENT: PETRI NETS OR RULES?*

Loretta A. Moore and Shannon W. Price
Computer Science and Engineering
Auburn University
Auburn, AL 36849
(205) 844 - 6330
moore@eng.auburn.edu

Joseph P. Hale
Mission Operations Laboratory
NASA Marshall Space Flight Center
MSFC, AL 35812
(205) 544-2193
joe.hale@msfc.nasa.gov

ABSTRACT

An evaluation of a prototyped user interface is best supported by a simulation of the system. A simulation allows for dynamic evaluation of the interface rather than just a static evaluation of the screen's appearance. This allows potential users to evaluate both the look (in terms of the screen layout, color, objects, etc.) and feel (in terms of operations and actions which need to be performed) of a system's interface. Because of the need to provide dynamic evaluation of an interface, there must be support for producing active simulations. The high-fidelity training simulators are normally delivered too late to be effectively used in prototyping the displays. Therefore, it is important to build a low fidelity simulator, so that the iterative cycle of refining the human computer interface based upon a user's interactions can proceed early in software development.

INTRODUCTION

The Crew Systems Engineering Branch of the Mission Operations Laboratory of NASA Marshall Space Flight Center was interested in a dynamic Human Computer Interface Prototyping Environment for the International Space Station Alpha's on-board payload displays. On the Space Station, new payloads will be added to the on-board complement of payloads in ninety day increments. Although a payload starts its development and integration processes from two to four years before launch, a set of new payloads' displays are due every ninety days. Thus, this drives the need for an efficient and effective prototyping process. The functional components of a dynamic prototyping environment in which the process of rapid prototyping can be carried out have been investigated.

Most Graphical User Interface toolkits allow designers to develop graphical displays with little or no programming, however in order to provide dynamic simulation of an interface more effort is required. Most tools provide an Application Programmer's Interface (API) which allows the designer to write callback routines to interface with databases, library calls, processes, and equipment. These callbacks can also be used to interface with a simulator for purposes of

* This research is supported in part by the Mission Operations Laboratory, NASA, Marshall Space Flight Center, MSFC, AL 35812 under Contract NAS8-39131, Delivery Order No. 25. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of NASA.

evaluation. However, utilizing these features assumes programming language knowledge and some knowledge of networking. Interface designers may not have this level of expertise and therefore need to be provided with a friendlier method of producing simulations to drive the interface.

A rapid prototyping environment has been developed which allows for rapid prototyping and evaluation of graphical displays [2]. The components of this environment include: a graphical user interface development toolkit, a simulator tool, a dynamic interface between the interface and the simulator, and an embedded evaluation tool. The purpose of this environment is to support the process of rapid prototyping, so it is important that the tools included within the environment provide the needed functionality, but also be easy to use.

This paper describes two options for simulation within the dynamic prototyping environment: petri nets and rule-based simulation. The petri net system, PERCNET [3], is designed to be used as a knowledge-based graphical simulation environment for modeling and analyzing human-machine tasks. With PERCNET, task models (i.e., simulations) are developed using modified petri nets. The rule based system is a CLIPS [1] based system with an X windows interface for running the simulations. CLIPS executes in a non-procedural fashion making it ideal for representing random and concurrent events required by the simulation. Its C language-based design allows external communication to be programmed directly into the model. In order to compare the two approaches for simulation, a prototype of a user interface has been developed within the dynamic prototyping environment with both simulation architectures. This paper compares the two systems based upon usability, functionality, and performance.

ARCHITECTURE OF THE DYNAMIC PROTOTYPING ENVIRONMENT

There are four components of the Human Computer Interface (HCI) Prototyping Environment: (1) a Graphical User Interface (GUI) development tool, (2) a simulator development tool, (3) a dynamic, interactive interface between the GUI and the simulator, (4) an embedded evaluation tool. The GUI tool allows the designer to dynamically develop graphical displays through direct manipulation. The simulator development tool allows the functionality of the system to be implemented and will act as the driver for the displays. The dynamic, interactive interface will handle communication between the GUI runtime environment and the simulation environment. The embedded evaluation tool will collect data while the user is interacting with the system and will evaluate the adequacy of an interface based on a user's performance. The architecture of the environment is shown in Figure 1.

[Figure Deleted]

Figure 1. HCI Prototyping Environment Architecture

Interface Development Tool

The Graphical User Interface (GUI) tool for the prototyping environment will allow the designer to create the display through direct manipulation. This includes the creation of static and dynamic objects, windows, menus, and boxes. The tool also allows objects created to be linked to a data source. During execution, the interface objects send and receive data and commands to the simulator by way of the data server. The user interface objects and their associated data access description are defined independent of the actual source of data. This first allows the development of the interface and the simulator to occur concurrently. Second, an interface developed with the GUI tool can later be connected to a high fidelity simulator and then to the actual flight software.

Simulator Development Tool

The simulator development tool provides the capability to develop a low fidelity simulation of a system or process. The development of a simulation has two important functions. First, the simulation helps the designer identify and define basic system requirements. Second, potential users can evaluate both the look (in terms of the screen layout, color, objects, etc.) and feel (in terms of operations and actions which need to be performed) of a system. The simulator provides realistic feedback to the interface based on user inputs.

Dynamic, Interactive Interface

This interface will handle communication between the GUI prototyping tool and the simulation tool during execution. The interface is a server which has been developed using the GUI's Application Programmer's Interface. Messages and commands can be sent and received both ways between the GUI and the simulator. The server also services requests from the embedded evaluation process, providing information as to which actions the user has taken and which events and activities have fired.

Embedded Evaluation Tool

An important aspect of the prototyping process is the ability to evaluate the adequacy of the developed graphical user interfaces. The embedded evaluation tool communicates with the server to receive information on the interaction between the user and the system. The types of data collected include user actions, simulator events and activities, and the times associated with these items. The collected data is analyzed to determine task correctness, task completion times, error counts, and user response times. The data is then analyzed to provide feedback as to which features of the interface the user had problems with and therefore need to be redesigned.

An Example: The Automobile Prototype

In order to assess the architecture described above a system was chosen to be prototyped in the environment. The system chosen for empirical evaluation of the HCI prototyping environment was an automobile. An automobile has sufficient complexity and subsystems' interdependencies to provide a moderate level of operational workload. Further, potential subjects in the empirical studies would have a working understanding of an automobile's functionality, thus minimizing pre-experiment training requirements.

An automobile can be considered a system with many interacting components that perform a task. The driver (or user) monitors and controls the automobile's performance using pedals, levers, gauges, and a steering wheel. The dashboard and controls are the user interface and the engine is the main part of the system. Mapping the automobile system to the simulation architecture calls for a model of the dashboard and driver controls and a separate model of the engine. Figure 2 demonstrates how an automobile system could be mapped into the architecture described. The main component of the automobile is the engine which responds to inputs from the driver (e.g. the driver shifts gears or presses the accelerator pedal) and factors in the effects of the environment (e.g. climbing a hill causes a decrease in the speed of the car). The driver changes inputs to obtain desired performance results. If the car slows down climbing a hill, pressing the accelerator closer to the floorboard will counteract the effects of the hill.

[Figure Deleted]

Figure 2. Automobile Prototype

The dashboard and controls have been modeled using Sammi [5], a graphical user interface development tool developed by Kinesix. Two options have been investigated for simulation: petri nets and rules. Petri nets provide a graphical model of concurrent systems. The petri net system which has been used is PERCNET [3], developed by Perceptronics. PERCNET is designed to be used as a knowledge-based graphical simulation environment for modeling and analyzing human-machine tasks. With PERCNET, task models are developed using modified petri nets, a combination of petri nets, frames, and rules. The rule based system which has been used is CLIPS [1], a rule based language primarily used for the design of expert systems, developed by NASA. CLIPS executes in a non-procedural fashion making it ideal for representing random and concurrent events. The automobile system has been prototyped using both the petri net and rule-based systems as simulators and comparisons have been made based upon functionality, usability, and performance.

SIMULATION IN THE DYNAMIC PROTOTYPING ENVIRONMENT

Because of the need to provide dynamic evaluation of an interface rather than just static evaluation, there must be support provided for producing active simulation. Most GUIs, including Sammi, provide some sort of Application Programmer's Interface (API) which allow the developer to write call back routines which interface with databases, library calls, other processes and equipment. We would like to provide a means of building a low fidelity simulation of the system to drive the interface which requires little programming.

Basic simulation requirements include the ability to model events and activities, both sequentially and concurrently. The system should provide the ability to create submodels within the main model. The simulator clock must be linked to the system clock, and support should be provided for the creation of temporal events. The process must be able to communicate with UNIX processes using the TCP/IP protocol. Real-time communication must also be provided to allow the tool to communicate with the GUI tool on a separate platform via Ethernet. The ability for two-way asynchronous communication between the runtime versions of the interface and the simulator must be provided. The simulator must be capable of receiving data from the GUI tool to dynamically control temporal events, to modify the values of variables, and trigger events and activities. The ability to specify and send commands, data, and alarms to the GUI tool must also be provided. A simulator director should be able to send commands (e.g., start simulation, trigger scenario event, etc.) to the simulator from a monitoring station. An interface should be provided in order to bind interface objects to simulation objects in order to set the values of variables, trigger events or activities, and set temporal variables.

Simulation Using Petri Nets

PERCNET is a very powerful system analysis software package designed by Perceptronics, Inc. It provides an easy-to-use, graphical interface which allows users to quickly lay out a petri net model of the system. PERCNET uses "modified" petri nets, which allow each state to describe pre-conditions for state transitions, modify global variables, perform function calls and maintain a global simulation time.

Pictorially, Petri nets show systems of activities and events. Ovals represent activities which describe actions performed by the system. Activities are joined by events, represented by vertical bars, that occur during execution. Events are associated with user actions and environmental conditions. Execution is shown by tokens propagating through the system. Flow of control passes from activities to events. Before an event can fire all incoming arcs must have tokens. When this occurs, the event places tokens on all outgoing arcs passing control to activities. The behavior that an event exhibits during execution is dependant on the data contained in its frame. Frames record data related to each activity and event. Event frames may contain rules and functions.

Activity frames allow the designer to specify a time to be associated with each activity. Figure 3 shows the top-level petri net of the automobile simulator.

[Figure Deleted]

Figure 3. Top-Level Petri Net of the Automobile Simulator

The starter is the component that is activated by the turning of the key. Before the starter can begin working, however, the key should be turned on, the driver must be wearing his/her seat belt, the car must be in neutral and the battery must have a sufficient charge to start the starter. When all three pre-conditions are true, the starter is activated and control advances to the right in the Petri net. Once the starter has been activated, it must do its part to start the automobile. The starter allows electricity to flow into the distributor where it is channeled into the spark plugs. As long as the starter is functioning, the distributor and spark plugs are activated. Finally, as long as the spark plugs and distributor are working properly and there is gasoline, the spark from the spark plugs ignites the gasoline mixture in the engine and ignition is achieved. Now that ignition has been accomplished, the engine is running. The concentric circles representing the engine_running activity in Figure 3 indicate that the state is shown in a sub-net.

The petri net representing the automobile passes from the ignition portion to the engine running state and remains in the running state until some condition causes the engine to stop running. The engine will stop running if the engine runs out of gas, runs out of oil, the temperature rises above a certain threshold, the key is turned off, the engine stalls (when the automobile is in some gear and the rpms fall below a threshold amount), the battery loses its charge or the fuel pump, oil pump, spark plugs or alternator fail.

The major components of the engine modeled are: fuel pump, oil pump, water pump, distributor, spark plugs, starter, battery, alternator, and fan. The condition of these components is modeled using a boolean variable indicating either that they are functioning or they are not. The boolean variables are then used as conditions within events occurring during the simulation. Details of the Petri Net implementation can be found in [2].

Simulation Using Rules

Since CLIPS is rule-based, it is completely non-procedural. Furthermore, it allows programmers to pick the strategy by which successive rule-firings are chosen. Certain rules may be designated fired by different priority levels (rules with the highest priority fire before rules with lower priority). Other rule-selection strategies govern how rules with equal priority are selected. Events and activities are represented by the pre- and post-conditions of rules. For example, the rule for activating the starter is:

```
(defrule TURN_KEY
   ?tick <- (clock_tick)
   (test (= 1 ?*key*))
   (test (= 1 ?*seatbelt*))
   (test (= ?*gear* 0))
   (test (> ?*battery* 10.0))
   (test (= ?*state* ?*READY*))
   =>
   (bind ?*state* ?*STARTER*)
   (retract ?tick)
   (assert (clock_tick))
   (printout t "ACTIVATE STARTER (" ?*time* ")" crlf)
   (tick_tocks 2)
   (assert (updated TRUE)))
```

In this project, CLIPS has been extended to include communication capabilities [4]. Two sockets have been provided for reading from and writing to the server. C functions have been developed to eliminate redundant information from the messages passed to the server. Another improvement compiled into the CLIPS executable has been a control process that allows a user to start, stop and quit CLIPS execution through a graphical interface.

The project also demonstrates some programming techniques used in CLIPS to support the simulation. A global simulation time should be maintained and a mechanism for keeping simulation execution time has been demonstrated. Another important feature that makes use of the timer is the periodic update feature. This ensures that CLIPS execution pauses (i.e., no rules may fire) every few seconds to send and receive information from the server. When this happens, control returns to the main routine which initializes communication with the server.

Writing CLIPS programs to take advantage of this strategy requires the incorporation of several techniques. These techniques include rules, variables, and functions which may be used in subsequent simulation designs. The first choice involves determining which values will be passed to or received from the server. All global variables (defined using the "defglobal" command) are passed to the server. No other values are passed. Facts and local variables may be used to store values which do not need to be passed to the server. It will be shown later how communication has been further streamlined for efficiency. The most important rule is the clock rule.

The clock rule stays ready at all times, but because the salience (i.e. priority) of the rule is kept low, it will not block the firing of other rules. When execution begins, the current system time is retrieved and stored. The current simulation time is always known by retrieving the system time and comparing it to the starting time. The new simulation time is temporarily stored in a variable called “new_time”and is compared to the last calculated time. If the two values are the same, then the clock rule has fired more than once within one second. In that case, the time is not printed and facts are reset to allow the clock rule to fire again.

A “clock_tick” fact is used in the preconditions of rules to allow them to become ready for firing. Without the clock_tick fact, a rule may never fire. Another time feature provided is the tick_tocks function. Often a programmer would like to force a rule to consume clock time. A call to the tick_tocks function forces execution to enter a side loop where the required time elapses before execution continues.

COMPARISON

Usability

Most features of PERCNET are easy-to-learn and use While some study of petri-net theory would benefit designers, much could be done with very minimal knowledge of petri-nets. One difficulty in working with PERCNET was the lack of available documentation on the Tool Command Language (TCL). All function calls, calculations, communication and ad-hoc programming are done using this language. Perceptronics provides only minimal documentation on the use of the language within PERCNET making it very difficult to perform anything more than the most basic operations. However, PERCNET's graphical interface is very appealing to users.

CLIPS is a rule-based language, which means that there may be a larger learning curve than there is with PERCNET's point-and-click interface. After the initial learning stages, however, CLIPS leaves a developer with an immensely powerful simulation tool. The main advantage is flexibility. CLIPS was written in the C programming language and is completely compatible and

extendible with C functions. Knowing C in advance can significantly lessen the learning curve. Many of the "non-C" features of CLIPS resemble LISP. CLIPS has been a tremendous surprise to work with. A basic proficiency with CLIPS may be gained quickly and one can learn to do very useful things with the language. Writing the rules for the simulation was actually the easiest part of the project. As proficiency with the language developed, more advanced features provided tremendous possibilities. The manuals present the language in a very easy to read format, contained extensive reference sections and sample code. Furthermore, the manuals outline how CLIPS may be easily extended to include C (and other) functions written by programmers.

Functionality

As this project began, PERCNET was a closed package, that is, there was no provision for communicating with other applications. NASA contracted Perceptronics to modify PERCNET to allow for such a feature. The final result was a revision of PERCNET which would allow communication with other applications through the use of sockets. Applications are allowed to request that global variables be retrieved and/or modified. PERCNET essentially opened it's blackboard (i.e., global data store) to other applications. The other application in this case being the server.

After several functions were added to CLIPS (see descriptions in previous sections), the CLIPS system performed the same functions as the Petri Net simulator. If a new system is prototyped, the only changes which would be needed are to the knowledge base. The communication link developed for the Sammi-CLIPS architecture uses the blackboard paradigm to improve modularity, flexibility, and efficiency. This form of data management stores all information in a central location (the blackboard), and processes communicate by posting and retrieving information from the blackboard. The server manages the blackboard, allowing applications to retrieve current values from the board and to request that a value be changed. The server accepts write requests from valid sources and changes values. The comparison of the two architectures goes much further than comparing the two simulation designs. The design of the communication link significantly affects the flexibility and performance of the architecture.

Performance

The performance within the Petri Net architecture was not acceptable for real-time interface simulation. Interfaces running within this architecture exhibit a very slow response rate to user actions when PERCNET is executing within its subnets. The PERCNET execution is also using excessive amounts of swap space and memory which also affect the refreshing of displays.

Early analysis attempted to find the exact cause of the poor performance; however, only limited work could be done without access to PERCNET's source code. Since PERCNET's code was unavailable, we could only speculate about what was actually happening to cause the slow responses. It was determined that the cause of much of the problem was that PERCNET was trying to do too much. In the PERCNET simulation architecture, PERCNET is actually the data server for the environment. The global blackboard is maintained within PERCNET. The server only provides a mechanism for passing information between PERCNET and other applications. The server is connected to PERCNET by a socket and the server is actually on the "client" end of the connection-oriented socket. The server establishes connections with PERCNET and Sammi and then alternately receives information from each. Any data or commands received from Sammi are passed immediately to PERCNET. Commands from PERCNET for Sammi are passed immediately through, as well. Finally, the server sends Sammi copies of all variables. Since PERCNET is the blackboard server, as well as the simulator, PERCNET's performance would naturally be affected by the added burden. Lastly, the method provided for sending variables to

the server was terribly inefficient. When a calculation was performed in the simulation model for a variable that was needed by the interface, that variable was passed to the server whether or not it's value had changed from the previous iteration. No mechanism was provided for restricting the number of redundant values passed across the communication link. As a result, PERCNET passed every value back to the server when only a few had actually changed.

Each of these limitations was addressed in the design of the server and blackboard in the rule-based architecture. The server program may be divided into three portions: blackboard management, Sammi routines, CLIPS routines. The Sammi and CLIPS routines are provided to communicate with the respective applications. These routines map data into a special "blackboard entry" form and pass the data to the blackboard management routines. The blackboard routines also return information to the Sammi and CLIPS routines for routing back to the applications. The blackboard management routines require that each application (many more applications may be supported) register itself initially. Applications are assigned application identification numbers which are used for all subsequent transactions. This application number allows the blackboard to closely monitor which variable values each application needs to see. It also provides a mechanism for installing a priority scheme for updates.

The overwhelming advantage of the CLIPS and blackboard combination is the flexibility and potential they provide. Features are provided that allow modifications which can affect performance. The ability to tune the performance has allowed the simulation architecture to be tailored to specific running conditions (e.g., machine limitations, network traffic and complexity of the interface being simulated). Several parameters may be modified to alter performance. Tuning tests have improved performance. More detailed performance testing is planned to verify the results.

CONCLUSION

The goal of the architecture has been to provide simulation of user interfaces so that they may be designed and evaluated quickly. An important portion of the dynamic prototyping architecture is therefore the simulator. Ease-of-use is very important, but performance is critical. The Petri Net architecture's ease-of-use is currently its only advantage over the Rule-Based architecture. The Rule-Based design overcomes this with power and flexibility. Work currently in progress include a detailed analysis of the performance of the communication link and a design of a graphical interface to CLIPS.

REFERENCES

1. CLIPS Reference Manual, NASA Johnson Space Flight Center, Houston, Texas, 1993.

2. Moore, Loretta, "Assessment of a Human Computer Interface Prototyping Environment," Final Report, Delivery Order No. 16, Basic NASA Contract No. NAS8-39131, NASA Marshall Space Flight Center, Huntsville, Alabama, 1993.

3. PERCNET User's Manual, Perceptronics Inc., Woodland Hills, California, 1992.

4. Price, Shannon, "An Improved Interface Simulation Architecture," Final Report for Master of Computer Science and Engineering Degree, Auburn University, Auburn, Alabama, 1994.

5. Sammi API Manual, Kinesix Corporation, Houston Texas, 1992.

COLLABORATIVE ENGINEERING-DESIGN SUPPORT SYSTEM

Dong Ho Lee and D. Richard Decker
Electrical Engineering and Computer Science Department
19 Memorial Drive West
Lehigh University
Bethlehem, PA 18015

ABSTRACT

Designing engineering objects requires many engineers' knowledge from different domains. There needs to be cooperative work among engineering designers to complete a design. Revisions of a design are time consuming, especially if designers work at a distance and with different design description formats. In order to reduce the design cycle, there needs to be a sharable design description the engineering community, which can be electronically transportable. Design is a process of integrating that is not easy to define definitively. This paper presents Design Script which is a generic engineering design knowledge representation scheme that can be applied in any engineering domain. The Design Script is developed through encapsulation of common design activities and basic design components based on problem decomposition. It is implemented using CLIPS with a Windows NT graphical user interface. The physical relationships between engineering objects and their subparts can be constructed in a hierarchical manner. The same design process is repeatedly applied at each given level of hierarchy and recursively into lower levels of the hierarchy. Each class of the structure can be represented using the Design Script.

INTRODUCTION

Design is a fundamental purposeful human activity with a long history. Design can include creative artistic and engineering components. Knowledge-based design systems deal with factors that tend to be limited to the engineering aspects of design. Many researchers have developed knowledge-based engineering design systems. Many of these systems were developed for the specified design domain using their own unique problem solving method [2, 5, 9, 12, 21]. Some have tried to develop domain independent knowledge-based design systems. The DIDS (Domain Independent Design System) by Runkel [18] was developed as set of tools that can provide a configuration-design system from a library. DOMINIC [10] treats design as best-first search by focusing on the problem of iterative redesigning of a single object. GPRS (Generative Prototype Refinement Shell) was developed by Oxman [15], which used Design Prototype [8, 22] as a knowledge representation. In the real world, most engineering designs are so complex that a single individual cannot complete them without many other engineers' help. Cooperation between different engineering designers is not a simple process because each designer may have a different perspective for the same problem, and multiple revisions of a design are needed in order to finish a project. Designers may have different interpretations of the same design value or may want to access special programs to determine values for the design variables in which they are interested. In order to achieve the above goals, there needs to be a design knowledge representation that can be shared between designers and that can be modified to the designers' needs. In addition, if a designer needs to execute a special program, the design system should provide a scheme to do so. This paper describes a Design Script as an abstract model of the design process that is based on hierarchical design structure and shows how to capture design knowledge and integrate data and tools into a knowledge based design system.

MODELS OF DESIGN PROCESSES

Dieter [7] describes the design process in his book as follow: “There is no single universally acclaimed sequence of steps that leads to a workable design.” But it is possible to make the fundamental design process as simple as an iterative process of analysis, synthesis, and evaluation (Fig. 1). Analysis is required to understand the goals of the problem and to produce explicit statements of functions. The synthesis phase involves finding plausible solutions through the guidance of functions that are produced from the analysis phase. The evaluation process checks the validity of solutions relative to the goals. The evaluation phase can be divided into two different types of jobs. One is to compare the solution with existing data if the solution is composed of comparable data; and the other is to compare the solution values derived from the current design solution through simulation process executions with the given goals.

[Figure Deleted]

Figure 1. Goal, Function, Form and Behavior Relationship

Problem decomposition is a well-known problem solving strategy in knowledge-based design systems. Once a complex design problem (a complete object) is decomposed into simple design problems (subparts), it is much easier to handle. The engineering design object is considered as being composed of sub-parts in a hierarchical structure. In other words, the problem of designing a complete object is comprised of designing number of subparts recursively until the designing process reaches the elemental subparts. The extent and complexity of a design problem can be different at different levels of hierarchy, but the identical design process can be applied at all hierarchical levels.

DESIGN SCRIPT

The DS (Design Script) is presented in this paper as an engineering design knowledge representation scheme. The DS is implemented in COOL (CLIPS Object Oriented Language) of NTCLIPS which is a Windows-NT version of CLIPS developed by the authors [20]. The common design process which is composed of analysis, synthesis, and evaluation phases can be abstracted by encapsulating design activities, such as goal decomposition and invoking methods. The abstracted model Design Script is placed at the top of the design knowledge for a domain. The major components of the DS are the goals, functions, and form of the part which is designed (Fig. 2). These abstractions include how the goals are decomposed and passed to lower level objects, how the functions of subparts are represented and used, and how explicit knowledge about reasoning and transforming process are formed and used. The instances of design sub-part can inherit these abstractions.

[Figure Deleted]

Figure 2. Main Structures of Design Script

Usually, the initial goal of the design object is represented using natural language rather than by using a technical representation such as “power amp can drive as low as 2-ohm impedance loud speaker with 80 dB SPL.” Then the goal is refined to the functions such as “require an amplifier which can provide 20A current at peak with at least 200 W/ch power.” The functions of the initial stage of the design process are regarded as goal of the next stage of design. The relations between goal and its functions are acquired knowledge from the designer’s point of view. When a designer sets goals (or requirements) for the design object, the designer has what should be required to meet the goals. Although, the relations between goals and functions are intrinsic knowledge, they should be represented explicitly in the knowledge-based design process. The

Goal slot of DS will hold a bound instance of the *Goal-Design* class as a slot value. Each instance of the *Goal-Design* class has its intrinsic functions kept as a multi-value slot to meet current goals. The *Attributes* slot of DS is a multi-value slot which keeps all the design variables and values for the design problem. DS provides various manipulation methods such as read-out and write-in the value of the attribute, decomposing the design into sub-parts, and creating instances of attributes and sub-parts. There always exist tradeoffs between generality and efficiency. To improve the efficiency of the design system, it could be built as a very domain specific. On the other hand, to make it general may require a sacrifice of efficiency. The DS has been developed to be very general, in that it can be applicable to different design domains.

GENERALIZATION AND AGGREGATION

Generalization is a powerful abstraction that can share commonalties among classes. Usually the generalization is represented as a superclass. The Design Scrip is a generalization of the design process that is based on the hierarchical structure of engineering objects. Each subclass inherits the features of its superclass. The inheritance mechanism of most object-oriented programming languages is based on a *sub-type* or *is-a* relation between classes. The COOL of CLIPS is not an exception. All the parts of a design object are represented as a subclass of DS so that they can inherit the properties of DS by using the built-in *is-a* hierarchy relation. But, the hierarchy relations among the parts themselves can not be represented by using *is-a* or *kind-of* relations, because they are actually *part-of* hierarchies (Fig. 3). Aggregation is the *has-part* or *part-of* relationship in which objects are part of the whole assembly. For example, let's take an example of a SO-8 package. We can say "*An SO-8 package is a kind of IC package.*" and "*IC is a kind of electronic component.*" ("SO-8" stands for "Small Outline 8 pin" and "IC" stands for "Integrated Circuit") We can see the *is-a* hierarchy relations between *an SO-8 package*, *IC package,* and *electronic component.* If we build the hierarchical structure for *SO-8,* we can see that it is composed of a mold material, a lead frame, wire bonds, etc. The lead frame is composed of lead1, lead2, die pad, etc. The relationship between these parts can not be represented with ***i****s-a* relations. We can say the lead frame is not a type of SO-8, but a sub-part of the SO-8. So, in order to express the design knowledge, a mechanism must be available to handle the *part-of* relationship hierarchy. The slots *PART-OF* and *HAS-PART* of DS are used to define the *part-of* relations relative to super and subparts of the part under consideration.

DESIGN KNOWLEDGE MANAGEMENT

Many design engineers have stressed the uselessness of unmodifiable design knowledge. If fixed engineering design knowledge doesn't have a self-adaptive function for new situations, it may not work properly in these cases. For our system, the DS needs to follow the syntax of CLIPS when building a design knowledge base for a specific application design domain, because DS is

[Figure Deleted]

Figure 3. Hierarchical Structure of SO8 IC Package and Design Script

implemented in CLIPS. The syntax of CLIPS is similar to that of the LISP language which uses lots of parentheses. CLIPS uses prefix notation in numerical computations rather than infix notation, which is generally used to represent algebraic expressions. If the developer is not familiar with this type of environment, it could be difficult to build the knowledge base. Considering that most design engineers are not computer scientists who can easily adapt to new programming environments, a user interface for knowledge entry and building is provided as a most essential part of the design system. The DS provides a GUI (Graphical User Interface) empty form as part of its functionality (Fig. 4). The design knowledge can be built easily by using the knowledge entry dialog-box on a part by part basis. What the designer has to do is fill

in each field of the dialog box with the necessary information for designing or analyzing the object or subpart. It is an essential process to specify the design object in the hierarchical structure before entering design knowledge. Each dialog-box can edit design knowledge for a single (composite or elemental) part, such as its part-of and has-part relations, its goal, functions of the goal, design attributes for the part, and its numerical methods to get the implicit values of its design variables. When the user edits the dialog-box, the contents of the fields are stored temporarily in instances of the *KENTRY* class. Because the purpose of the *KENTRY* class is to provide an editable form of design knowledge that is easy to use, the contents of the design knowledge are kept as text in the corresponding slot. Once this information is entered by the user, it can be saved into a file in the form of instances of the *KENTRY* class from the main menu. Later, the user can modify the design knowledge by loading a previously created version from the file. Whenever the user wants to run the design system with the current design knowledge, he first compiles the knowledge from the instance form to the form of classes and their message-handlers which are in CLIPS syntax.

[Figure Deleted]

Figure 5. Knowledge Entry Class

DESIGN APPLICATION: LEAD-FRAME DESIGN FOR PQFP

The DS has been applied in the domain of Lead-Frame design for a plastic quad flat pack IC package (PQFP). Acquiring design knowledge for the lead-frame of a PQFP is not a simple process. First of all, manufacturers tend to use their own knowledge to produce packages for their chips. Moreover, each company wants to keep this knowledge as proprietary information. There is available a limited amount of public design knowledge such as the standard package outlines contained in the *JEDEC Package Standard Outline*, research paper [11], or handbooks [19, 23]. This standard is widely used throughout the community of semiconductor manufacturers. The JEDEC standard is just for the outside dimensions, such as body size, length of outside lead, lead pitch, etc. The JEDEC manual doesn't refer to the inside dimensions of the package. This limitation reveals one difficulty of standardization in the packaging world. The major goal of the lead-frame design system for PQFP is to define the geometry of a lead-frame which can satisfy the JEDEC standard and the electrical and mechanical design constraints.

Fig. 6 describes the design knowledge for a lead-frame in graphical format. The values of lead-pitch, lead-width, and outside lead length are decided from JEDEC standard dimensions which are stored in a JEDEC database. The size of the chip bonding pad is decided by reference to the chip design technology base in use. The maximum wire span of the bonding wire from the bonding pad on the chip to the lead tip of the lead-frame is 100 mil. The lead tip pitch is around 10 mil. The width of the lead tip is around 6 mil. The length of the lead tip is dependent on its position. Each lead has an anchor hole. The fineness of the lead tip is also limited by the sheet metal blanking technology. The whole area of the metal part inside the plastic package is equal to or a slightly less than half of the entire plastic area.

[Figure Deleted]

Figure 6. Design Knowledge for the Lead-Frame of the PQFP.

When a user loads or edits the design knowledge using the *Knowledge Entry* dialog box and compiles it, the design knowledge is translated into CLIPS syntax (Fig. 7.). The first part of the knowledge is for the definition of LF-PQFP (Lead Frame of Plastic Quad Flat Pack) class which has *GOAL* and *HAS-PART* slots. The default value of the *GOAL* slot is the name of an instance of the GOAL-DESIGN class which has information about how to achieve the goal. Another

multi-value slot contains the name and number of the subpart class. The sub-part of the lead-frame of the PQFP is composed of one die-bond-pad and the number of I/O leads in one octant of the package.

```
(defclass LF-PQFP (is-a DESIGN-SCRIPT)
    (role concrete)
    (slot GOAL (create-accessor read-write) (default "LF-1stGoal"))
    (slot WIRE-SPAN (create-accessor read-write) (default 100))
    (multislot HAS-PART (create-accessor read-write)
                        (default "DieBondPad" "Lead1" "Lead2" "Lead3" . . . "Lead11"))
)

(message-handler LF-PQFP set-attributes()
    (bind ?self-i (instance-name ?self))
    (bind ?ins-name (symbol-to-instance-name (sym-cat ?self-i "LEADTIPPITCH")))
    (make-instance ?ins-name of AN-ATTRIBUTE
        (ATTR-NAME "LEAD-TIP-PITCH")
        (value 0.254))
    (send ?self add-attr ?ins-name)
)

(message-handler LF-PQFP decide-Die-Size ()
    (bind ?SB (send ?self get-attr-value (send ?self get-attr-instance "Die-SB"))
    (bind ?Nio (send ?self get-attr-value (send ?self get-attr-instance "N-IO"))
    (bind ?Pd (send ?self get-attr-value (send ?self get-attr-instance "PAD-PITCH"))
    (bind ?DieSize (+ (* 2 ?SB) (*?Pd (+ 1 (/ ?Nio 4)))))
    (send ?self put-attr-value (send ?self get-attr-instance "PAD-PITCH") ?DieSize)
)
```

Figure 7. CLIPS syntax for the Class and Message-Handler.

The reason for designing only an octant of I/O leads is that the geometry of the remaining leads can be derived from this portion of the geometry by considering the symmetry of the lead-frame. The next message-handler takes care of creating and storing the instances of the attribute. The last message-handler is an example of a method used to decide the design value of the attribute.

[Figure Deleted]

Figure 8. Design Result: Lead-Frame of an 84 pin PQFP

Figure 8 shows the lead-frame of an 84 pin PQFP that was created by running the above lead-frame design system. The outer dotted square represents the plastic body of the PQFP and the inner dotted square shows the minimum size of the chip. The figure is drawn using the GNU plot program by providing the geometry that is produced by the lead-frame design system.

CONCLUSIONS AND FUTURE WORK

The main contribution of this research is in providing a general design-process control framework and a general design-knowledge representation scheme for physical objects to be designed using knowledge-based design systems. The Design Script developed here can be applied in any design domain because it contains domain knowledge independent about the design process. Usually, design is not only an individual activity but also requires cooperative team work that involves a number of designers from different fields. To support cooperative design work, DS provides an excellent framework that can be used in different domains. The capability of DS has been demonstrated in the domain of leadframe design for PQFPs.

This work is a part of the ongoing project. Much remains to be done to enhance the functionality of DS. Especially, management of design knowledge is a good candidate for future work. The current knowledge entry process doesn't provide any debugging functions. The design knowledge may be easily broken without careful knowledge entry. But there is also a need for a version management method like RCS (Revision Control System) or Aegis to enhance the productivity of cooperative design work. This enhancement is being considered to provide such functionality to the DS.

ACKNOWLEDGMENTS

The financial support from Semiconductor Research Corporation under the contract number MP-071 is gratefully appreciated.

REFERENCES

1. Akagi, S., "Expert System for Engineering Design Based on Object-Oriented Knowledge Representation Concept," *Artificial Intelligence in Design (ED. PHAM D. T.)*, 1991, 61-95.

2. Brown, D. C., and Chadrasekaran, B., "Design Problem Solving: Knowledge Structures and Control Strategies," *Pitman/Morgan Kaufmann*, 1989.

3. Chandrasekaran, B., "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design," *IEEE Expert*, 1986, 1(3), 23-30.

4. Chandrasekaran, B., "Design Problem Solving: A Task analysis," *AI Magazine*, 1990, 11(4), 59 - 71.

5. Choi, C-K., and Kim, E-D., "A Preliminary Model of I-BUILDS: An Intelligent Building Design System," *Knowledge Based Expert Systems in Engineering: Planning and Design (Ed. Sriram D. and Adey R.A.)*, 1987, 331-343.

6. Coyne, R. D. et al., "Knowledge-Based Design Systems," *Addison-Wesley Publishing Co.*, 1990.

7. Dieter, G. E., "Engineering Design A Materials and Processing Approach," McGraw-Hill Book Co., 1983.

8. Gero, J. S., "Design Prototypes: A Knowledge Representation Schema for Design," *AI Magazine*, 1990, 11(4), 26-36.

9. Harty, N., "An Aid to Preliminary Design," *Knowledge Based Expert Systems in Engineering: Planning and Design (Ed. Sriram D. and Adey R. A.)*, 1987, 377-392.

10. Howe, A. E. et al., "Dominic: A Domain-Independent Program for Mechanical Engineering Design, *Artificial Intelligence in Engineering Design*, July 1986, 1(1), 23-28.

11. Jahsman, W. E., "Lead Frame and Wire Length Limitations to Bond Densification," *Journal of Electronic Packaging*, December 1989, Vol. 111, 289-293.

12. Maher, M. L., "HI-RISE and Beyond: Directions for Expert Systems in Design," *Computer Aided Design*, 1985, 17(9), 420-427.

13. Maher, M. L., "Process Models for Design Synthesis," *AI Magazine*, 1990, 11(4), 49-58.

14. Nguyen, G. T. and Rieu D., "SHOOD: A Design Object Model," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 221-240.

15. Oxman, R., "Design shells: a formalism for prototype refinement in knowledge-based design systems," *Artificial Intelligence in Engineering Design Progress in Engineering Vol.12*, 1993, 92-98.

16. Oxman, R., and Gero J. S., "Using an Expert System for Design Diagnosis and Design Synthesis," *Expert Systems*, 1987, 4(1), 4-15.

17. Rumbaugh, James et al., Object-Oriented Modeling and Design, Prentice Hall, 1991.

18. Runkel, J. T., et al., "Domain-Independent Design System, Environment for rapid development of configuration-design systems," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 21-40.

19. Seraphim, D. P.(Ed.), Principles of Electronic Packaging, *McGraw-Hill*, 1989.

20. STB, CLIPS Reference Manual Ver. 6.0, *Lyndon B. Johnson Space Center*, 1993.

21. Sriram, D. et al., "Knowledge-Based Expert Systems in Structural Design," *Computers and Structures*, 1985, 20(13) : 1-9.

22. Tham, K. W. and Gero, J. S., "PROBER -A Design System Based on Design Prototypes," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 657-675.

23. Tummala, R. R.(Ed.), Microelectronics Packaging Handbook, *Van Nostrand Reinhold*, 1989.

CHARACTER SELECTING ADVISOR FOR ROLE-PLAYING GAMES

Felicia Berlanga,
Independent Study Mentorship at Holmes H.S.

Carol L. Redfield
Southwest Research Institute
6220 Culebra Road
San Antonio, TX 78228

Role-playing games have been a source of much pleasure and merriment for people of all ages. The process of developing a character for a role-playing game is usually very, very time consuming, delaying what many players consider the most entertaining part of the game. An expert system has been written to assist a player in creating a character by guiding the player through a series of questions. This paper discusses the selection of this topic, the knowledge engineering, the software development, and the resulting program that cuts the time of character development from about 4 hours to 20 minutes. The program was written on a PC and an Apollo in CLIPS 4.3 and currently runs on the Apollo. Gamers have used the program and their comments are reported on in this paper.

THE COMPUTER AIDED AIRCRAFT-DESIGN PACKAGE (CAAP)

Guy U. Yalif
(617) 973-1015

ABSTRACT

The preliminary design of an aircraft is a complex, labor-intensive, and creative process. Since the 1970's, many computer programs have been written to help automate preliminary airplane design. Time and resource analyses have identified, "a substantial decrease in project duration with the introduction of an automated design capability" (Ref. 1). Proof-of-concept studies have been completed which establish "a foundation for a computer-based airframe design capability" (Ref. 1). Unfortunately, today's design codes exist in many different languages on many, often expensive, hardware platforms. Through the use of a module-based system architecture, the Computer Aided Aircraft-design Package (CAAP) will eventually bring together many of the most useful features of existing programs. Through the use of an expert system, it will add an additional feature that could be described as indispensable to entry level engineers and students: the incorporation of "expert" knowledge into the automated design process.

INTRODUCTION

It is widely recognized that good engineers need not only the textbook knowledge learned in school, but also a good "feel" for the designs with which they are working. This "feel" can only be gained with both time and experience. An expert system is an ideal way to codify and capture this "feel". This idea is the key feature of CAAP. With this package, engineers will be able to use the knowledge of their predecessors, as well as learn from it. The potential value of such a program in aiding the engineering professionals as well as the student is great.

The ultimate goal of CAAP is to design a plane in an intelligent way based on user specifications. A rough-sizing configuration is created from user inputs and then analyzed using rule based programming. Throughout the design process, the user is given total access to the CAAP database, which is implemented using object oriented programming. The user can see how variables affect each other, view their present values, and see, create, and arrange rules in a customizable fashion using "Toolbox" files. CAAP exists as a core program with Toolbox files that add functionality to that core, similarly to the popular program "MATLAB". CAAP's core program has been written while its Toolbox files are still in development.

SYSTEM OVERVIEW

Preliminary aircraft design, as described in above, is a multi-faceted problem whose features have driven the choice of software platform used to implement CAAP. This section will detail the features that led to a CLIPS based implementation for CAAP. One aspect of the usefulness of an expert system to the CAAP package has already been discussed.

The design process is a *potentially iterative* procedure. This is best explained with an example. During a hypothetical airplane design, one might re-size the wing five times. On the other hand, it is possible that the engineer will not alter the original fuselage. The possibility of iterative re-design for some components and not for others defines a potentially iterative process. Airplane design is such a process, and it is therefore well modeled by the rule based programming syntax of an expert system.

As other designers have noted, "tremendous amounts of data and information are created and manipulated [during the aircraft design process] to produce numerous parts which are eventually

assembled to a sophisticated system. ... It is becoming clear that a critical issue to effective design is the efficient management of design data" (Ref. 2). The data produced during the design process is voluminous at the very least, but it is not haphazardly arranged. The information needed to design a plane falls into organized patterns. Specifically, a hierarchical structure exists for most components of an airplane. One such component is engine classification, as illustrated in Figure 1. This figure diagrams the hierarchy that is used to describe engines in CAAP. This type of logical hierarchy exists throughout the airplane design process.

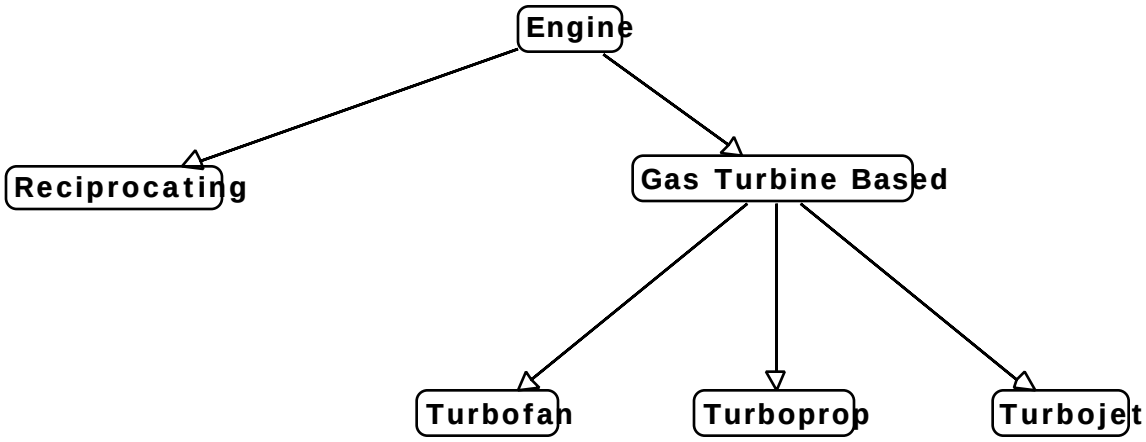


Figure 1. Engine Classifications in CAAP

The data used during airplane design can be very complicated. Each part of the plane, such as the engine, has its own specifications. Each part also contains subparts, just as the engine has a turbine, compressor, and fuel pumping system. Each of these subparts has its own specifications in addition to sub-subparts. Therefore, design data needs to be arranged in an ordered manner that is readily accessible and understandable to the user. Object Oriented Programming is useful for storing the complex, voluminous, and hierarchically arranged data produced during airplane design. The usefulness of OOP has been recognized elsewhere in the aerospace industry. In a study entitled "Managing Engineering Design Information" (Ref. 2), ten different data storage methods were examined. The conclusion: "The object-oriented data model was found to be a better data modeling method for modeling aerospace vehicle design process" than any of the others studied (Ref. 2).

OOP also facilities the organization of the large number of routines available to aid in aircraft design. Effective routine organization is a desirable quality of any airplane design program. CAAP seeks to accomplish routine organization in two ways. First, routines are grouped into the Toolbox files introduced above. Second, within each Toolbox, different equations are applied to different parts of the airplane as is appropriate. Having the ability to separate the equations according to airplane component aids in the logical organization of the program. Such separation also increases the efficiency of CAAP. For example, it would be a waste of computational time to have the aspect ratio rule searching instances of the FUSELAGE class for possible span and area values. OOP and CLIPS run-time modules allows the programmer to implement such class-specific routine separation.

As discussed above, the order of execution of the routines that analyze an airplane cannot be determined before run-time because of the potentially iterative nature of design. The routines themselves, however, are composed of equations that *do* need to be executed in a predetermined order. For example, the routine that determines the range-payload curve needs to add up the range covered during climb, cruise, and descent over and over again until the desired limiting

payloads are reached. This is an ordered process that is best modeled by a procedural programming paradigm.

The desirability of using multiple programming paradigms has been discussed above. Because of these needs, CLIPS was chosen to implement CAAP. CLIPS provides useful and effective rule based, object oriented, and procedural programming paradigms as well as a high level of cross-paradigm interactions. CLIPS is also offered on a wide variety of hardware platforms, ensuring its ability to the student and professional alike.

A Macintosh platform was used to implement these program design goals. The Macintosh was chosen because of the widespread access to Macs, as opposed to the limited access available to more powerful UNIX workstations such as IRIS's or Sun's. Nonetheless, if a user had access to these workstations, CAAP would be fully portable to their environments. CAAP's text based user interface has been written completely in CLIPS. A second, graphically based user interface, however, has been designed strictly for Macintosh use. CAAP has been successfully run on many different Macintosh models, although no porting tests have been performed for other platforms.

It has been recognized that a modular layout "will preserve the ability for independent expansion at a later date" (Ref. 3). This key concept is reflected in CAAP's design. As Kroo and Takai have succinctly stated, "If the founding fathers did not envision that future aircraft might employ maneuver load alleviation, winglets, or three lifting surfaces, it may be more expedient to rewrite the code than to fool the program into thinking of winglets, for example, as oddly shaped stores" (Ref. 4). With a sectioned program, such problems can be quickly alleviated with the addition of another Toolbox or an upgrade to an existing one.

As mentioned above, CAAP is organized into a central program and a variety of Toolboxes which add functionality to the base program. The core program represents all of the code that is necessary to run an expert system which utilizes CAAP's data structures. As a result, this code has the possibility of being recycled in the future. The Toolboxes will add functionality to the core program. If someone wishes to run CAAP at all, they need to possess the core program. If someone also wishes to perform weight sizing of their aircraft, they also need to possess the Weight Toolbox. If a certain Toolbox is not present while a user is designing an airplane with CAAP, the part of the analysis that the Toolbox is responsible for will not be completed. Continuing with the previous example, if the Weight Toolbox is missing, no weights will be assigned to the various components of the plane. Missing Toolboxes could prevent an airplane from being completely designed. Nonetheless, arranging a program in this fashion allows users to customize their personal version of CAAP as they desire. The user will, as a hypothetical example, have the ability to choose a Roskam Weight Toolbox instead of a Raymer Weight Toolbox, if they so desired. In addition, if the user does not want CAAP to perform a certain kind of analysis, the Toolbox based program allows them to disable a segment of code analysis easily. It is worth noting that no time consuming re-compilation is presently necessary to remove a Toolbox from CAAP. A simple menu option allows users to choose which Toolboxes will be loaded at start-up.

Mimicking the real engineer, CAAP's core program will prompt the creation of a rough, initial sizing for an airplane. The code will then analyze this initial configuration of the plane. If the given configuration does not meet one of the user's performance specifications, or if the plane does not pass the appropriate FAR regulations, CAAP will modify a particular part of the plane.

A diagram of the system architecture that will accomplish these tasks is presented in Figure 2. The diagram depicts the interactions between CAAP's run-time modules. Into each of these run-time modules will be loaded routines that perform certain functions in the airplane design process. For example, the Performance Module will contain routines that determine performance

estimates. Run-time modules are not to be confused with Toolbox files. Toolbox files are files that contain routines organized in an arbitrary manner chosen by the user. Run-time modules group routines by functionality only.

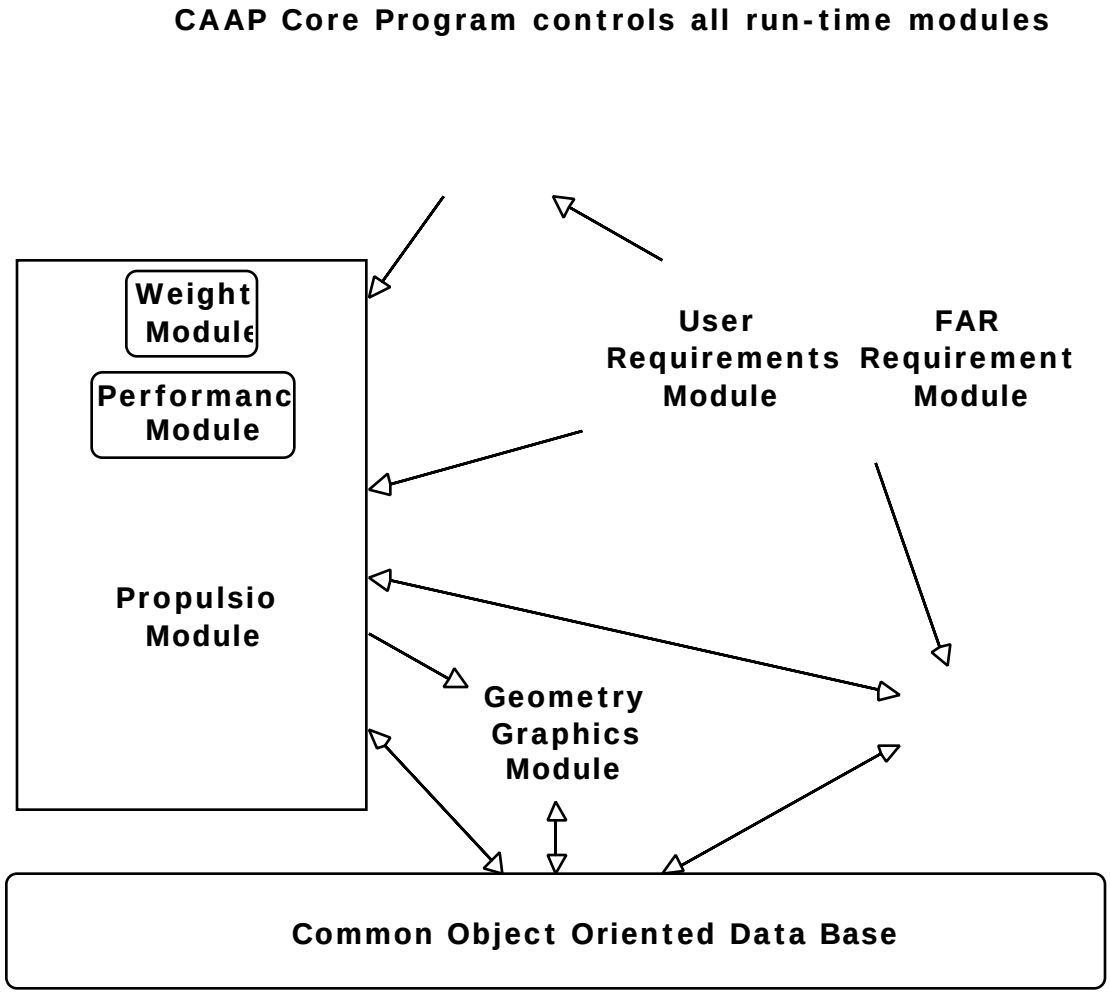


Figure 2. System Architecture

The CAAP core program is presented around all of the run-time modules in order to emphasize that it is the core code that drives all of the routines within the run-time modules and allows them to perform their allotted analytical tasks. The Initial Sizing Module produces the initial parameters for the first configuration. The rest of the modules then analyze, alter, and re-analyze the subsequent configurations. The final plane will be presented to the user through the routines in the Geometry Graphics Module. The user will then be able to change the proposed solution/configuration, and the process will start over again. This time, however, the "old" solution with the user's modifications will become the "initial configuration".

THE RULE WRITING UTILITY

The rules that make up CAAP need not only to perform their prescribed functions, but also to provide variable dependency information to the routine that performs variable assignments for CAAP. As described later in the *Consistency Maintenance* section, if a variable in the system is calculated, all of the variables that depend on its value must be recalculated. It is useful to

maintain a data storage system which can provide CAAP with these variable dependencies. Rules are no longer written in their traditional format. In previous versions of CAAP, when a program developer wanted to add a rule to the expert system, they have had to learn the syntax of rule writing in CLIPS and then how to hard code the rule into the system. This required coding some standard constructs that perform some of the repeated type checking that goes on within CAAP. These constructs were usually very long and "messy", and therefore very time consuming to write.

In CAAP Version 2.0, programmers can add rules to the system by using the Rule Writing Utility (RWU). In order to add a rule, the programmer creates an instance of the class RULE_INFORMATION. They can do this manually or with option 8 of CAAP's main menu. Both methods create objects containing several slots: one set of slots is created listing the input variables and another set is created listing the output variables of the rule that is represented by the instance. The input variables can be restricted to be equal, not equal, less than, greater than, less than or equal to, or greater than or equal to some other value. The text of the calculation that will be executed by the actual rule is also stored in a slot of the instance of RULE_INFORMATION. Therefore, the programmer needs only enter two sets of variables (input and output) and a string representing a calculation (and some housekeeping information). The rest is handled by the RWU.

Before system operation beings, the RWU code creates the class RULE_INFORMATION and the rule "make_rules." "Make_rules" creates expert system rules and places them into the Rule Base based on the existing instances of RULE_INFORMATION. It also adds all of the constraint checking that is necessary for proper CAAP operation. Such a utility could be useful in other Expert Systems that involve the same type of input and output for each rule. As previously mentioned, such code recycling opportunities are an important aspect of CAAP.

Once "make_rules" has fired, each rule is represented in two places: one sense of the rule exists in the Rule Base as an actual expert system rule. Another sense of the rule exists in the Object Oriented Database as an instance of the class RULE_INFORMATION. This second representation of the rule is used by the assignment routine to satisfy its need to know how variables depend on each other. Assignment routine operation and the way the routine uses variable dependency information are described in the *Consistency Maintenance* section.

The present method of double representation is more efficient than what was possible with CLIPS 5.1. Previously, if a programmer wanted to add a rule with five inputs and five outputs, they would have to check that 5 X 5 or twenty-five separate variable dependencies were included in the dependency functions (in addition to the constraint checking information). In Version 2.0, the programmer simply needs to list the five inputs and the five outputs of the rule at the time of RULE_INFORMATION instance creation. This brings CAAP a large step closer towards decreasing the future programmer's work load. Additions to the Rule Base can now be generated more easily and more quickly than before.

THE CORE PROGRAM

CAAP's Core Program has been given the basic abilities needed to design an airplane. At the time of this writing, the Core Program still requires the addition of large amounts of data in the form of Toolbox files in order to be able to design an airplane. Nonetheless, the full functionality of the Core Program has been implemented. The main menu of the package appears in Figure 3. Descriptive English equivalents are used at every point throughout the user interface. A lot of attention was focused on making CAAP user friendly in order not to loose any potential users due to the newness of the program.

```
Main Menu
---------
1) Create an airplane
2) Modify requirements or plane
3) Analyze airplanes
4) Save an airplane to disk
5) Load an airplane from disk
6) Look at an airplane
7) Look at slot dependencies
8) Add a rule to the database
9) Look at internal variable representation
10) Manipulate Toolboxes
11) Miscellaneous
12) Quit

Please choose an option (0 to reprint menu)>
```

Figure 3. CAAP Main Menu

The user is presently allowed to create airplanes, modify the specifications the airplane is designed to, and change the airplane's describing variable values. The user may save and load airplanes from disk. The user is also given complete access to the CAAP database. Designers are allowed to look at the variable values that represent an airplane, either individually or in a summary sheet format. They may also look at the dependencies that exist within the CAAP database. This valuable tool would tell the user, for example, that the aspect ratio depends on the wing area and the wing span for its values (reverse dependencies), and that when the wing span is changed, the aspect ratio will have to be recalculated (forward dependencies).

The user is allowed to see a list of all variables which have been defined to CAAP and which are not used in any of the presently loaded rules. This can help designers load the appropriate Toolboxes or add rules where necessary. Users can write rules during run-time and add them to Toolbox files. This feature allows for simple expansion of CAAP by individual users in the future, and, combined with the Toolbox manipulation functions, has allowed CAAP to become a self-modifiable program. With this ability, CAAP can evolve to meet individuals needs as they create and change Toolbox files and the rules that populate them.

The user is given some aesthetic controls over CAAP output. The user can look at CAAP's internal variables representations. Toolboxes can be created, deleted, and loaded during run-time. Users can view which Toolboxes have been loaded into the CAAP database as well as choose which Toolboxes to load each time CAAP starts up. The dynamic use of Toolbox files presents some interesting situations. The files can act as a medium for knowledge exchange in the future. For example, Joe can design a plane with the Toolboxes he has built over time. He can then add Mary's rules to his personal Toolbox, and redesign his plane in order to discover how Mary's know-how can improve his model. Such an interactive exchange of information could be very useful, especially in an teaching environment.

CONSISTENCY MAINTENANCE AND THE AVAILABILITY OF SEVERAL ROUTINES TO CALCULATE ONE VARIABLE

There are several different methods available to estimate almost any of the parameters used in airplane design. Different sources will quote different methods, each with its own result. A consistent method for routine execution is needed. When there is more than one equation or routine available to calculate a given parameter, CAAP will select the most advanced one for which all of the input variables have been determined. For example, the airplane drag coefficient can be calculated using equation (1) or with a drag build-up.

$$C_{L_\alpha} = C_{l_\alpha} \frac{A}{A + [2(A+4)/(A+2)]} \qquad (1)$$

where

C_{L_α} = lift curve slope for a finite lifting surface
C_{l_α} = section lift curve slope
A = aspect ratio

If the components of the plane have been defined, the latter, more advanced drag estimation method will be used. If the components have not yet been defined, the former, simpler method will be used. Importantly, once the components have been defined, the LHS of the drag build-up rule will be satisfied and CAAP will *recalculate* the drag based on the more advanced drag build-up. All calculations are based on the most advanced routine available, due to the rule based programming implementation chosen for CAAP.

Rule "advancedness" will be represented by a priority associated with each rule. This priority is stored in a "rulepriority" function in CAAP's core program. It is presently used to ensure that the Inference Engine sees more "advanced" rules more quickly than it sees more primitive rules. Rulepriority is used by the RWU to set rule saliences during system initialization. This procedure improves program efficiency by decreasing the likelihood that a particular variable will be calculated many times by successively more advanced routines when a very advanced one could have done the job originally. Rule prioritization also allows the user to be confident that the crude initial estimates used in the Initial Sizing Toolbox will not be used in the final configuration. As soon as the airplane begins to take shape, the Initial Sizing Toolbox's estimates will be replaced with more advanced values[4].

If two methods are similarly "advanced", one method will be chosen over the other arbitrarily, but not randomly. If the designer has a preference as to which method is used by CAAP, he or she can specify this to the package. The "rulepriority" function alleviates the need for addressing the situation when two expert recommendations agree. Either they will have different priorities, or their location on the agenda will determine which is fired.

CONSISTENCY MAINTENANCE AND PARAMETER MODIFICATIONS

During the design process, configurations are created and analyzed. If the analysis shows a given configuration to be inadequate in some way, the rules within the Expert run-time module will modify one of the design parameters of the given configuration, in effect creating a new configuration. Until the effects of this single modified parameter have been propagated throughout the airplane, the configuration will be inconsistent. In another scenario, an advanced routine might recalculate a design parameter previously calculated by a more primitive routine. Again, until the effects of this change have been propagated throughout the system, an inconsistent configuration will exist. The solution to this problem follows.

---

[4]This does not necessarily have to occur, if one is not careful. It would be possible for the airplane to be presented as a final product without enough of it having been calculated to replace the Initial Sizing Toolbox's estimates. This would be an absurd situation, and it would result in problems. CAAP will not present a plane to the user unless a minimum set of parameters have been calculated to a sufficient level of "advancedness". This way, no Initial Sizing Toolbox estimates will make their way to the user.

Consistency maintenance will be accomplished in two ways. When a rule within the Expert run-time module modifies an airplane design parameter, it will have to do so in a "responsible" manner. For example, suppose the Expert rule, for an "expert" reason, decides that the aspect ratio of the wing needs to be changed. If it simply changes the aspect ratio, the span and/or wing area will be inconsistent. Therefore, the Expert rule will have to *also* change the span or the wing area. The rule could, for example, adjust the aspect ratio while keeping the wing area constant. In other words, the Expert rule will have to look at the input variables that determine the value of the design parameter and modify them so that they are consistent with the new value of the changed variable.

The second consistency maintenance procedure will be based on computational paths. Figure 4 presents a diagram of a hypothetical set of computational paths. Each box on the diagram represents a variable. The directed connections represent the dependency of a variable on the value of other variables.

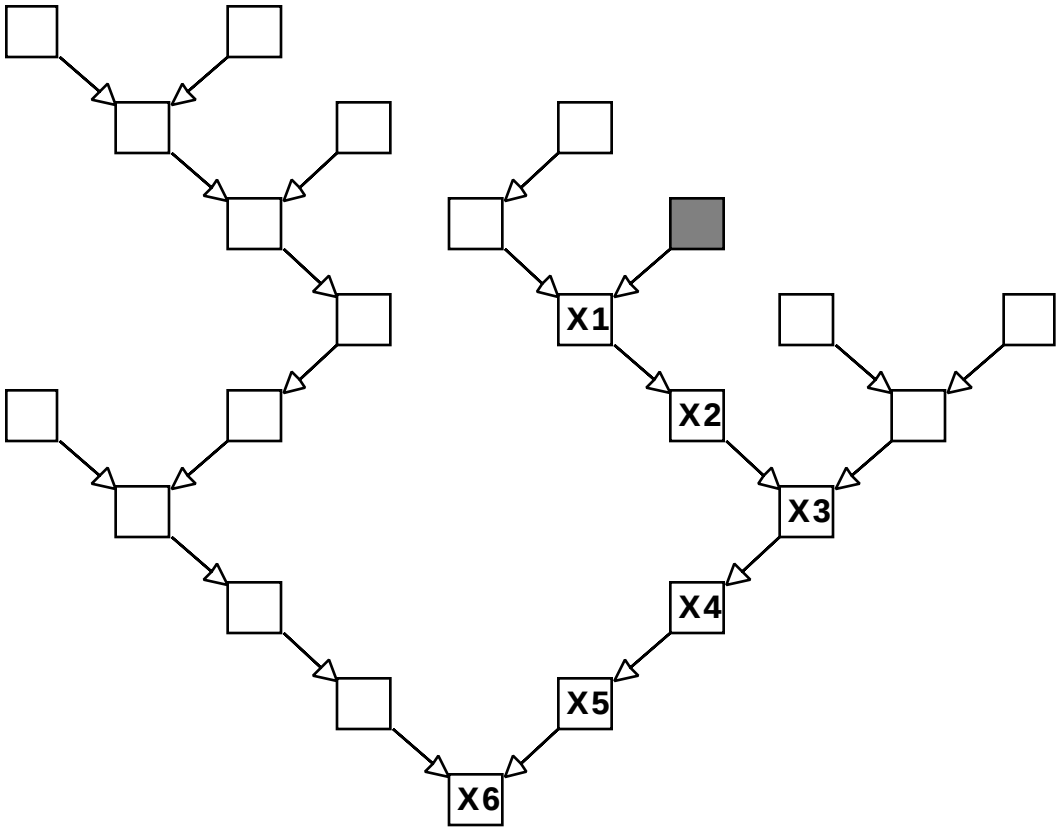


Figure 4. Consistency Maintenance Example

Suppose that the variable in the shaded box has just been redefined, perhaps by a rule from the Expert run-time module or by an advanced estimation routine. The value of every box "downstream" of the shaded box is now inconsistent. The "downstream" variables are represented by the presence of an "X" in the variable box. The "downstream" variables need to be recalculated as if they had never been determined in the first place. Each rule will have access to the list of variables which depend on the variable in the shaded box (i.e. X_1 in this example). This list is stored in instances of the RULE_INFORMATION class, introduced in the *Rule*

Writing Utility section. The rule will erase, or undefine[5], all of the variables that depend on the changed variable (i.e. it will undefine X_1). The Toolbox will then undefine all of the variables that depended on those variables, and so on until the there are no more dependent variables to undefine (i.e. X_2, X_3, ... , X_6). This systematic undefining is called the "downstream" erasure procedure. It has been coded as part of the "assign" routine that is used for all variable assignments. Every rule must use the "assign" routine. After a "downstream" erasure, the other rules in CAAP will automatically recalculate the undefined "downstream" variables. This will occur since the LHS of CAAP rules is satisfied when the input variables for the rule are undefined.

A problem with the method of consistency maintenance presented in Figure 4 will arise if any loops exist within the computational paths. A discussion of this problem is beyond the scope of this paper, and the problem has only been partially solved. A full solution to the "Loop Problem" is one of the major remaining issues facing CAAP.

PRACTICAL LIMITATIONS

The future of CAAP will focus on three different areas: the core program, the Toolboxes, and the user interface. The essentials of the core program have been entirely written. Some extra functionality has also been added to the program. Nonetheless, there is always room for improvement and CAAP is by no means complete. Among the pieces of code not yet written is a numerical optimizer. Such code could provide CAAP with a way to make "expert" recommendations when no rules from the Expert run-time module apply to a given configuration. If no rules exist to help, CAAP could turn to numerical optimization methods in order to determine what changes to make to a configuration in order to make it meet all user and FAR requirements. A simultaneous equation solver could significantly facilitate solving the airplane design problem.

The Toolbox files need to receive a significant amount of data. Proof of study Toolbox files have been implemented and successfully tested, but there remains a lot of data to input in order to fully design an airplane. The graphical user interface ran into difficulties associated with system level Macintosh programming. Finding an alternative to friendly user interactions will be a priority for CAAP in the future.

The first category of plane that CAAP should be able to completely design will be the light, general aviation, non-acrobatic, single engine aircraft. The graphics for displaying the airplane are next on the implementation list. Eventually, trend studies and increased user involvement in the design process could be added. For example, if the user wished CAAP to produce several final designs instead of one, this could be done. If the user wished to watch CAAP fire one rule at a time, this could be done. A utility could be added to allow users to see which rules are firing at any given time. This would provide the user with a better "feel" for how the package is going about designing their airplane.

CONCLUSION

A firm theoretical foundation has been developed for CAAP. The problem of designing an airplane has been laid out and implemented using rule based, object oriented, and procedural programming paradigms. Rule based programming enables CAAP to capture expert knowledge

[5]CLIPS 6.0 no longer supports undefined slot values. It is necessary to have such reserved values for airplane variables that may take on a range of values. In order to satisfy the LHS's of any of the rules, the LHS's must contain tests for variables to see if they have not yet been calculated, that is that they are undefined. A typical undefined value is -1e-30 for a floating point variable.

and to mimic the *potentially iterative* nature of preliminary airplane design. Object oriented programming handles the voluminous, complex, and hierarchically arranged data produced during airplane design. Procedural programming is used to implement the actual analysis routines necessary for engineering design. CAAP has realized core program implementation and proof-of-concept Toolbox file creation and test. CAAP can begin designing airplanes and awaits the addition of more data in order to be able to complete the design process. CAAP is still in the developmental phase.

REFERENCES

1. Newman, D., and K. Stanzione. *Aircraft Configuration Design Code Proof-Of-Concept: Design of the Crewstation Subsystem*. Proc. of the AIAA Aircraft Design Systems and Operations Meeting. 23-25 Sept. 1991. Baltimore: AIAA paper No. 91-3097, 1991.

2. Fulton, R. E., and Yeh Chao-pin. *Managing Engineering Design Information*. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4452, 1988.

3. Roskam, Jan, and Seyyed Malaek. "Automated Aircraft Configuration Design and Analysis." *SAE Technical Paper Series No. 891072* (1989): General Aviation Aircraft Meeting & Exposition (Wichita, KS), 1989.

4. Kroo, I., and M. Takai. *A Quasi-Procedural, Knowledge-Based System for Aircraft Design*. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4428, 1988.

RULE BASED DESIGN OF CONCEPTUAL MODELS FOR FORMATIVE EVALUATION*

Loretta A. Moore+, Kai Chang+, Joseph P. Hale++, Terri Bester+,
Thomas Rix+, and Yaowen Wang+

+Computer Science and Engineering
Auburn University
Auburn, AL 36849
(205) 844 - 6330
moore@eng.auburn.edu

++Mission Operations Laboratory
NASA Marshall Space Flight Center
MSFC, AL 35812
(205) 544-2193
joe.hale@msfc.nasa.gov

ABSTRACT

A Human-Computer Interface (HCI) Prototyping Environment with embedded evaluation capability has been investigated. This environment will be valuable in developing and refining HCI standards and evaluating program/project interface development, especially Space Station Freedom on-board displays for payload operations. This environment, which allows for rapid prototyping and evaluation of graphical interfaces, includes the following four components: (1) a HCI development tool, (2) a low fidelity simulator development tool, (3) a dynamic, interactive interface between the HCI and the simulator, and (4) an embedded evaluator that evaluates the adequacy of a HCI based on a user's performance. The embedded evaluation tool collects data while the user is interacting with the system and evaluates the adequacy of an interface based on a user's performance. This paper describes the design of conceptual models for the embedded evaluation system using a rule-based approach.

INTRODUCTION

Formative evaluation is conducted through usability studies. Given a functional prototype and tasks that can be accomplished on that prototype, the designer observes how users interact with the prototype to accomplish those tasks in order to identify improvements for the next design iteration. Evaluation of the interaction is measured in terms of specific parameters including: time to learn to use the system, speed of task performance, rates and types of errors made by users, retention over time, and subjective satisfaction [4]. Analysis of this information will assist in redesign of the system.

The conceptual model of a designer is a description of the system and how the user should interact with it in terms of completing a set of tasks [2]. The user's mental model is a model formed by the user of how the system works, and it guides the user's actions [1]. Most interaction problems occur when the user has an inaccurate model of the system or when the user's model of a system does not correspond with the designer's conceptual model of the system. The evaluation

* This research is supported in part by the Mission Operations Laboratory, NASA, Marshall Space Flight Center, MSFC, AL 35812 under Contract NAS8-39131, Delivery Order No. 25. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of NASA.

approach which will be discussed in this paper evaluates the user's mental model of the system against the designer's conceptual model.

A rule-based evaluation approach, implemented using CLIPS, is used to develop the conceptual model. The model outlines the specific actions that the user must take in order to complete a task. Evaluation criteria which are embedded in the rules include the existence of certain actions, the sequencing of actions, and the time in which actions should be completed. Throughout the evaluation process, user actions are continuously associated with a set of possibly changing goals. Once a goal has been identified, the user's action in response to that goal are evaluated to determine if a user has performed a task correctly. Tasks may be performed at three levels: expert, intermediate, and novice.

The dynamic relationship between the evaluation tool and the user environment allows the simulation director to constantly introduce new goals that need to be responded to. This paper will discuss the approach of rule-based conceptual modeling and will provide an example of how this approach is used in the evaluation of a graphical interface of an automobile.

ARCHITECTURE OF THE HCI PROTOTYPING ENVIRONMENT

The Human-Computer Interface Prototyping Environment with Embedded Evaluation capability is designed to allow a developer to create a rapid prototype of a system and to specify correct procedures for operating the system [3]. The first component of the architecture is the Graphical User Interface (GUI) development tool. This tool allows the designer to graphically create the interface of the system and specify a data source for each object within the display. The simulator tool provides the capability to create a low-fidelity simulation of the system to drive the interface. The embedded evaluation tool allows the designer to specify which actions need to be taken to complete a task, what actions should be taken in response to certain events (e.g., malfunctions), and the time frames in which these actions should be taken. Each of these components is a separate process which communicates with its peers through the network server. Figure 1 shows the architecture of the HCI Prototyping Environment.

[Figure Deleted]

Figure 1. Architecture of the HCI Prototyping Environment

During execution of the system, the interface objects send and receive data and commands to the simulator by way of the data server and the simulator provides realistic feedback to the interface based on user inputs. The server sends the embedded evaluation tool the actions which the user has taken, all events and activities which have occurred, and the times associated with these items. The embedded evaluation tool analyzes the actions which have been performed by the user, that is, the user's model of the system, against the predefined conceptual model of the designer. The system identifies which tasks were completed correctly, or not, and provides data to the designer as to the points in the interaction in which the user's model of the system did not correspond to the designer's conceptual model of the system.

In order to evaluate the architecture, an automobile system was prototyped in the environment. An automobile was chosen because it has sufficient complexity and subsystems' interdependencies to provide a moderate level of operational workload. Further, potential subjects in the empirical studies would have a working understanding of an automobile's functionality, thus minimizing pre-experiment training requirements. The conceptual model which will be described in this paper is that for the automobile prototype. Before describing the rule based conceptual model, we will first discuss changes made to integrate CLIPS into the HCI Prototyping Environment.

THE GRAPHICAL USER INTERFACE EVALUATION TOOL (GUIET)

The goal of GUIET is to provide for dynamic evaluation of user actions within the HCI Prototyping Environment. Using GUIET, the process of formative evaluation has more flexibility and takes less time for analysis. The main feature is that the evaluation of most of the participant's actions are automated. The evaluation is performed at runtime by an expert system. The knowledge base of the system contains the designer's conceptual model, of how he/she thinks the user should interact with the prototyped system. Because the knowledge base is not hard coded into the application, it can be dynamically changed according to the needs of the evaluator. This provides the flexibility to evaluate different interfaces with the same evaluation criteria or one interface with different evaluation criteria. This design saves time because the data is automatically collected and analyzed based on the rule based conceptual model. If a new interface is prototyped, the only change that needs to be made with GUIET is changing the knowledge base.

In addition to the rule-based modeling design, GUIET provides a graphical interface and communication capabilities to CLIPS. In order to be integrated with the existing architecture, GUIET needs to receive information from both the interface and the simulator. The server sends the messages that are passed between the GUI tool and the simulator tool to GUIET. This is done using datagram sockets for the interprocess communication. The messages are in the form:

```
(newfact 193.0 I>S Message SetVariable gear 2)
```

where newfact is a string used for pattern matching for CLIPS rules, 193.0 is the time stamp, I>S states that communication is from the interface to the simulator, Message is the type of communication, SetVariable represents that the value of a variable is being set by the user, gear is the variable name, and 2 is the variable value.

Although this is the natural way to assert facts into CLIPS, the state for the car is stored not as a set of facts, but as one fact with many attribute/value pairs. Before the new fact is evaluated it is translated into the form of a fact. This translation is done by a set of translation rules. An example of a translation rule is:

```
; GEARS

(defrule trans_gear
   ?newfact <- (newfact ?time ?direction ?type ?action gear ?value)
   ?state <- (car_state)
   =>
   (modify ?state (gear ?value))
   (retract ?new_fact)
   (bind ?*current_tag* (+ ?*current_tag* 1))
   (assert (action (type gear) (value ?value) (time ?time)
           (clock_tag ?*current_tag*) ))
```

Another set of rules perform the evaluation of the user's actions, which are described in the next section. The last section describes a graphical interface which has been created for CLIPS.

CONCEPTUAL MODEL FOR THE AUTOMOBILE PROTOTYPE

The tasks which the user are asked to perform with the prototype can be divided into two categories: driving the car (i.e., using the controls) and responding to events (e.g., environmental and maintenance). The tasks measured include:

- Starting the car
- Driving forward (including changing gears)
- Driving backward
- Turning
- Stopping (at stop signs, lights, etc.)
- Parking the car
- Increasing and decreasing speed [Responding to speed limit changes]
- Driving uphill and downhill [Responding to hill events]
- Performing maintenance [Responding to maintenance events]
- Responding to environmental conditions

The events which can occur while the user is driving include environmental condition events (e.g., rain, snow, fog, and clear weather), time of day events (e.g., day and night), terrain changes (uphill and downhill), speed limit changes, and maintenance problems (e.g., gas, oil, battery, alternator, and engine). In addition to the events, the participant is given a set of instructions that must be followed. These are in the form of driving directions (e.g., drive 5 miles north and park the car).

Driving the car consists of manipulating graphical objects on the screen. For each of the tasks described above, the designer has determined a set of correct actions that must be made to complete the task. For example, the actions which must be taken for starting the car include:

1. Lock the seatbelt
2. Release emergency brake
3. Depress the brake
4. Depress the clutch
5. Put the gear in neutral
6. Turn the key on

Task correctness is evaluated based mainly on three evaluation criteria: the existence of certain actions, the sequencing of actions, and the time associated with the completions of the actions or task. An integer clock counter is used to indicate the action or event sequence. In the beginning of evaluation, the clock is reset to zero. Every subsequent action taken by the driver would increment the clock by one. Action sequence is important for many driving maneuvers. For example, clutch must be engaged before shifting gears. The evaluation process evaluates the correctness and effectiveness of a driver's interactions with the graphical user interface. User performance can be classified into three levels for most tasks - expert, intermediate, and novice. There may also be no response to a task. A counter is designated for each performance level. Every time a sequence of user actions is classified at a particular level, the associated counter will be incremented by one. The purpose of the evaluation is not to classify or evaluate users, but to evaluate the interface. The classification of users into categories is done to identify the level at which the users are interacting with the system. The goal is to have most if not all interactions at what the designer would consider the expert level. If users are not interacting at this level, it is the interface which must be enhanced to improve user performance.

In the CLIPS implementation a fact template is used to represent the car states and driving scenarios. The following shows the template for car-states:

```
(deftemplate car_states "Variables used in the car interface"
   (slot turnsignal)
   (slot brake)
   (slot emer_brake_on)
   (slot clutch)
   (slot key)
```

```
   (slot gear)
   (slot throttle)
   (slot speed)
   (slot seatbelt)
   (slot wipers)
   (slot lights)
   (slot fog_lights)
   (slot oil)
   (slot gasoline)
   (slot engine_temp)
   (slot battery)
   (slot alternator_ok)
   (slot rpm)
   (slot terrain)
   (slot day)
   (slot weather)
   (slot speed_limit)
)  ;car template
```

Associated with each action taken by the user, a template is defined as:

```
(deftemplate action "Action taken by the user on the interface"
   (slot type)       ;action type
   (slot value)      ;action value
   (slot time)       ;action time
   (slot clock_tag)  ;action sequence
)  ;action template
```

An evaluation rule is designed for each performance level. After a sequence of actions is completed, it will be evaluated based on the rules for the three performance levels. However, only one of the rules would succeed. The rules are organized in a way that the expert level would be tried first, then the intermediate level, and then the novice level. Once a rule has been successfully fired, this sequence of actions will be discarded. The prioritization of these rule is achieved through the *salience* values of CLIPS.

The following examples describe a portion of the evaluation process for *starting the engine*.

```
; Rules for Starting the Engine
(defrule expert_start_engine "Expert level starting engine"
   ; salience value highest among the four performance evaluation rules
   (declare (salience 3))

   ; This part of the rule ensures that all of the actions have occurred.
   ; the first action putting the seatbelt on
   ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
   ; next action is about the emergency brake
   ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
   ; next action is brake on
   ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
   ; next action is clutch depressed
   ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
   ; next action is gear neutral
   ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
   ; next action is key on
   ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
   ?f7 <- (noresponse start_engine)
```

```
   ; To ensure that the actions occurred in the proper order the following
   ; test is performed. The proper sequence is lock seatbelt, release
   ; emergency brake, depress brake, depress clutch, select neutral gear,
   ; turn key, release brakes and release clutch.

   (test (= (+ ?t 1) ?t1))
   (test (= (+ ?t 2) ?t2))
   (test (= (+ ?t 3) ?t3))
   (test (= (+ ?t 4) ?t4))
   (test (= (+ ?t 5) ?t5))
=>
   ; The following assertion aids in keeping track of whether a response
   ; was made or not.
   (assert (response start_engine))

   ; effects of actions considered, therefore removed from the fact base
   (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

   ; increment expert level count
   (bind ?*expert_count* (+ ?*expert_count* 1))

   (printout evaluation "TASK: starting the engine" crlf)
   (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) "
seconds" crlf)
   (printout evaluation "# ERRORS: 0" crlf crlf)

)  ; expert_start_engine

(defrule intermediate_start_engine "Intermediate level starting engine"

   ; salience value smaller than expert level, but higher than novice level
   (declare (salience 2))

   ; This part of the rule ensures that all of the actions have occurred.

   ; the first action putting the seatbelt on
   ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
   ; next action is about the emergency brake
   ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
   ; next action is brake on
   ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
   ; next action is clutch depressed
   ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
   ; next action is gear neutral
   ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
   ; next action is key on
   ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
   ?f7 <- (noresponse start_engine)

   ; The following test is to see what sequence the events occurred in.
   ; The proper sequence is: key turned after the clutch is off, clutch off
   ; after the brake is on, and seatbelt is on before the key is turned.
   ; Additional actions may be done in between the needed actions.
   ; For example, turning on the fog lights.

   (test (> ?t1 ?t))
   (test (> ?t2 ?t1))
   (test (> ?t3 ?t2))
```

```
   (test (> ?t4 ?t3))
   (test (> ?t5 ?t4))
=>
   ; The following assertion aids in keeping track of whether a response
   ; was made or not
   (assert (response start_engine))

   ; effects of actions considered, therefore removed from the fact base
   (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

   ; increment intermediate level count
   (bind ?*intermediate_count* (+ ?*intermediate_count* 1))

   (printout evaluation "TASK: starting the engine" crlf)
   (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) "
seconds" crlf)
   (printout evaluation "# ERRORS: 1 or more" crlf)
   (printout evaluation "EXPLANATION: Extra events occurred in the
sequence." crlf )

)  ;intermediate_start_engine

(defrule novice_start_engine "Novice level starting engine"

   ; salience value smaller than intermediate level, but higher than no
response level
   (declare (salience 1))
   ; This part of the rule ensures that all of the actions have occurred.

   ; the first action putting the seatbelt on
   ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
   ; next action is about the emergency brake
   ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
   ; next action is brake on
   ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
   ; next action is clutch depressed
   ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
   ; next action is gear neutral
   ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
   ; next action is key on
   ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
   ?f7 <- (noresponse start_engine)

   ; The events do not have to occur in any particular order. They just
   ; all have to occur.
=>
   ; The following assertion aids in keeping track of whether a response
   ; was made or not
   (assert (response start_engine))

   ; effects of actions considered, therefore removed from the fact base
   (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

   ; increment novice level count
   (bind ?*novice_count* (+ ?*novice_count* 1))

   (printout evaluation "TASK: starting the engine" crlf)
```

```
   (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) "
seconds" crlf)
   (printout evaluation "# ERRORS: 1 or more" crlf)
   (printout evaluation "EXPLANATION: Events occurred out of sequence."
crlf)

)  ;novice_start_engine

(defrule no_response_start_engine "No response to starting engine"
   ; No salience value is assigned therefore it
   ; has the default value of 0

   ; Check to see if there was an attempt to start the engine
   (noresponse start_engine)

   ; Check to see if the time limit has exceeded for starting the engine
   (test (>  (integer (time)) (+ ?*timeout* ?*start_time*)))
=>
   ; increment no response count
   (bind ?*no_response_count* (+ ?*no_response_count* 1))

   (printout evaluation "No response to starting the engine" crlf)
   (printout evaluation "TIME:  "  (- (integer (time)) ?*start_time*)
"seconds" crlf crlf)

)  ;no_response_start_engine
```

Rules for different tasks may contain different evaluation criteria. It depends on the designer's conceptual model of how he/she feels the task needs to be completed.

GRAPHICAL INTERFACE FOR CLIPS

A graphical interface was created for CLIPS using the Motif toolkit. The purpose of this interface is to provide the human evaluator a graphical means by which to use CLIPS. The main window of GUIET is composed of two areas. The top area is used for CLIPS standard input and output, and the bottom area displays any error or warning messages from CLIPS. These areas receive their input from a series of CLIPS I/O router functions. The menu bar for this window provides the following options: System, Network, CLIPS, Rules, and Facts. The system pulldown provides functions for quitting the application and for bringing up a window for entering participant information. The network pulldown allows the evaluator to send a selection of messages to the server (e.g., connect and disconnect). The CLIPS pulldown supports functions that affect the entire expert system, such as loading the evaluation rule base, resetting the expert system to its initial settings, running, clearing, or accepting input. Debugging functions that relate to the rules, such as watching/unwatching the rules fire or viewing/editing existing rules, can be accessed through the rules pulldown. Debugging functions that relate to the facts in the expert system are provided under the facts pulldown.

The user information window accessed under the system pulldown allows the evaluator to enter data relevant to the current participant and system being evaluated. The top section of this window displays the expert system's evaluations. There is an I/O router which handles the display for the evaluation window and is accessed through the printout statement within the rules. The bottom area is a text area in which the evaluator can enter comments or observations about the participant's session. The window provides options for saving, printing, and cancelling information.

CONCLUSION

The rule-based design of conceptual models enables the iterative process of design and evaluation to proceed more efficiently. A designer can specify user performance criteria prior to evaluation, and the system can automatically evaluate the human computer interaction based on the criteria previously specified. In order to evaluate the system which has been designed, a study is being planned which will evaluate user performance (using the rule based system developed) using a good interface and a bad interface. The hypothesis is that the good interface will produce more user responses at the expert level, and the bad interface will produce less acceptable responses.

REFERENCES

1. Eberts, Ray, *User Interface Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

2. Mayhew, Deborah, *Principles and Guidelines in Software User Interface Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

3. Moore, Loretta, "Assessment of a Human Computer Interface Prototyping Environment," Final Report, Delivery Order No. 16, Basic NASA Contract No. NAS8-39131, NASA Marshall Space Fight Center, Huntsville, AL, 1993.

4. Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1992.

AUTOMATED RULE-BASE CREATION VIA CLIPS-INDUCE

Patrick M. Murphy
Department of Information & Computer Science
University of California, Irvine, CA~92717
pmurphy@ics.uci.edu
(714) 725-2111

ABSTRACT

Many CLIPS rule-bases contain one or more rule groups that perform classification. In this paper we describe CLIPS-Induce, an automated system for the creation of a CLIPS classification rule-base from a set of test cases. CLIPS-Induce consists of two components, a decision tree induction component and a CLIPS production extraction component. ID3 [1], a popular decision tree induction algorithm, is used to induce a decision tree from the test cases. CLIPS production extraction is accomplished through a top-down traversal of the decision tree. Nodes of the tree are used to construct query rules, and branches of the tree are used to construct classification rules. The learned CLIPS productions may easily be incorporated into a large CLIPS system that perform tasks such as accessing a database or displaying information.

INTRODUCTION

Many CLIPS rule-bases contain one or more rule groups that perform classification. In this paper we describe CLIPS-Induce, an automated system for the creation of a CLIPS classification rule-base from a set of test cases. The rule-base created by CLIPS-Induce consists of two sets of rules, a set of user query rules to ask the user for any missing information necessary to perform classification, and a set of classification rules that is used to make the classification.

In the remainder of the paper, a detailed description of CLIPS-Induce and ID3 will be presented, followed by an analysis of CLIPS-Induce and list of potential extensions.

DESCRIPTION

In this section a description of CLIPS-Induce will be given, along with an example of its usage on a real-world problem, the Space Shuttle Landing Control problem. The goal of this classification problem is to determine whether the space shuttle should be landed manually or automatically.

CLIPS-Induce takes as input a set of test cases and returns two sets of CLIPS rules that perform user querying and classification. Each case is described in terms of a set of feature-value pairs. The same set of features is used for each case. An example case for the shuttle problem is given in Table 1.

- Landing = manual
- Stability = stab
- Error = mm
- Sign = nn
- Wind = tail

- Magnitude = OutOfRange
- Visibility = yes

Table 1. Example case from shuttle problem.

One feature is identified as the feature to be predicted given the values of the other features. For the shuttle problem, the feature *Landing* is to be predicted in terms of the features *Stability, Error, Sign, Wind, Magnitude and Visibility.*

[Figure Deleted]

Figure 1. Decision tree constructed by ID3 using the shuttle cases.

[Figure Deleted]

Figure 2. CLIPS-Induce Architecture

A decision tree is constructed from the set of cases using the decision tree construction algorithm ID3. The tree constructed from the shuttle cases is shown in Figure 1. The decision tree is then used to construct the user querying and classification rule sets. The basic organization for CLIPS-Induce to presented in Figure 2.

Decision Tree Construction

A decision tree predicts the value of one feature in terms of the values of other features. The process by which a prediction is made using a decision trees is described below.

Using the decision tree in Figure 1, the value of the feature *Landing* will be predicted for the example case shown in Table 1. Starting at the top (root) of the decision tree, the value of the feature *Visibility* is checked. Because the value is *Yes* the node at the end of the branch labeled *Yes* is next tested. Since the value for the feature *Stability* is *stab*, the *Error* node is next checked. Traversal of the tree continues down the *not(ss)* branch (because the value of Error is not *ss*), across the *mm* branch and finally down the *nn* branch to the leaf labeled *Manual*. The value for the *Landing* feature, predicted by the decision tree for this case is *Manual*.

ID3 is a decision tree construction algorithm that builds a decision tree consistent with a set of cases. A high-level description of the algorithm is shown in Table 2. The tree is constructed in a recursive top-down manner. At each step in the tree's construction, the algorithm works on the set of cases associated with a node in the partial tree. If the cases at the node all have the same value for the feature to be predicted, the node is made into a leaf. Otherwise, a set of tests is evaluated to determine which test best partitions the set of cases down each branch. The metric used to evaluate the partition made by a particular test is know as information gain (for a more complete description of information gain and ID3, see [1]). Once a test is selected for a node, the cases are partitioned down each branch, and the algorithm is recursively called on the cases at the end of each branch.

```
function generate_dtree(cases)
   if stop_splitting(cases)
      return leaf_class(cases);
   else
      best_cost := eval_examples_cost(cases);
      for all tests()
         cost := eval_cost(cases,test);
```

```
      if cost < best_cost then
         best_cost := cost;
         best_test := test;
   for all case_partitions(cases,best_test)
      branches := branches  {generate-dtree(case_partition)};
   return (best_test,branches);
```

Table 2. ID3 Decision Tree Construction Algorithm.

There are three types of tests that are used by CLIPS-Induce to construct decision trees:

1. Two branch *feature = value* test: One branch for *feature = value* and a second branch for *feature ≠ value*.

2. Multi-branch *feature = value* test: A branch for each of the values that *feature* has.

3. Two branch *feature > value* test: One branch for *feature > value* and a second branch for *feature ≤ value*.

The first and second test types are used for nominal-valued features, e.g. color. Whereas the third test type is used for features with real or ordered values, e.g. age or size.

Rule Generation

The first step in rule generation is to generate the user query rules. The purpose of each user query rule is to ask the user for the value of a particular feature. The user-defined function used to ask the actual questions is shown below.

```
(deffunction ask-question (?question)
   (format t "%s " ?question)
   (read))
```

The query rules are generated such that they only fire when the value for a particular feature is needed and not already available. If, for example, the values for certain features were asserted before execution of the classification rules began, query rules for those features would never fire. Typically, user query rules and classification rules fire in an interleaved manner.

User query rules are generated via a pre-order traversal of the tree. During the traversal, each internal node of the tree is associated with a unique identifier that is used to identify the act of having visited that node during rule execution. An example user query rule, for the *Sign* node in Figure 1, is shown below.

```
(defrule sign-query-g773
   (node node8)
   (not (feature sign ?value))
   =>
   (bind ?answer (ask-question "What is the value of feature sign?"))
   (assert (feature sign ?answer)))
```

The second step in rule generation is to generate the classification rules. The purpose of the classification rules is to traverse the decision tree along a path from the root of the tree to a leaf. Upon reaching the leaf, the value for the feature to be predicted is asserted to the fact-list. Whereas query rules are associated with internal nodes in the decision tree, classification rules are associated with branches in the tree. The two rules for the branches from the *Sign* node in Figure 1, are shown in Table 3.

The *Sign* node is identified as *node8*. If the value for feature *Sign* is *pp*, than *(node node9)* will be asserted by the first rule. *Node9* is associated with the *Magnitude* node. If the value for *Sign* were instead *nn*, the value *manual* for the predicted feature *Landing* would be asserted by the second rule. In the later case, because no new *(node ...)* fact is asserted, execution of the user query and classification rules halts.

```
(defrule node8-sign-pp
   ?n <- (node node8)
   (feature sign pp)
   =>
   (retract ?n)
   (assert (node node9)))

(defrule node8-sign-nn
   ?n <- (node node8)
   (feature sign nn)
   =>
   (retract ?n)
   (assert (feature landing manual)))
```

Table 3. Example classification rules.

ANALYSIS

The first issue to be concerned with in using the CLIPS-Induce, is the time savings relative to generating the rules by hand. For the shuttle problem, the 25 rules were generated from a set of 277 cases in only a few seconds. For another problem that deals with predicting lymph node cancer, 87 rules were generated in less than a minute. Other problems have been observed to generate rule-bases with as many as 500 rules in very reasonable amounts of time. Given that a set of test cases is available, CLIPS-Induce can save a great deal of time.

The second issue concerns the accuracy of the induced rules on new cases. This concern has been addressed by the area of machine learning where a great deal of research has been done on the induction of decision trees from cases. Specifically, ID3 has been empirically shown to do well at generating decision trees that are accurate on unseen cases. For example, Figure 3 shows a learning curve for the shuttle problem. Learning curves show the accuracy of a model (a decision tree) on unseen cases, as a function of the number of cases used to generate the model. For the shuttle problem, when only 10% of the 277 cases were used to generate the decision trees, the accuracy on the remaining 90% of the cases is approximately 92%. As the proportion of cases increases, the accuracy of the constructed decision trees increases.

The third and final issue concerns the availability of a sufficient numbers of cases needed to induce an accurate set of rules (from Figure 3, the fewer the number of cases, the less accurate is the induced decision tree). In answer to this concern, even if there are only a small number of cases for a problem, the rule-base generated by CLIPS-Induce can be used as a starting point for a domain expert.

[Figure Deleted]

Figure 3. Average accuracy of decision trees as a function of the proportion of the 277 cases used to construct the decision trees. The accuracy of each decision tree is based on the cases not used to construct the tree.

EXTENSIONS

One of the ways that CLIPS-Induce could be extended would be to take advantage of ID3's approach for dealing with missing feature values. Currently, the rule-bases, generated by CLIPS-Induce, halt when the user cannot enter a value for a required feature. The only drawback to extending CLIPS-Induce in this manner, is the increased complexity and reduced understandability of the generated rules.

Another enhancement to CLIPS-Induce would be to use a more sophisticated *ask-question* function. User-query rules could be generated that also pass the set of allowable values or value type to the *ask-question* function. The extra argument could provide constraints on the allowable responses made by the user.

The third extension to CLIPS-Induce would be to allow interactive creation of decision trees. It is often the case that an expert in the field has knowledge that could help in forming a more accurate and more understandable decision tree.

CONCLUSION

In this paper, CLIPS-Induce, a Common Lisp application that induces a CLIPS classification rule-base from a set of test cases, is described. Given a set of test cases, described in terms of a fixed set of features, a decision tree is constructed using the decision tree construction algorithm, ID3. From the decision tree, two sets of rules are extracted. One set of rules, the user query rules, ask the user for the values of features needed to make a classification. The other set of rules, the classification rules, simulate a traversal of the decision tree in order to make the prediction that the decision tree would make. The rule-base formed by CLIPS-Induce can easily be embedded in rule-bases that need classification rule groups.

REFERENCES

1. Quinlan, J.R., “Induction of Decision Trees,” *Machine Learning*, Boston, Massachusetts, 1(1), 1986, 81-106.

AUTOMATED REVISION OF CLIPS RULE-BASES

Patrick M. Murphy,
pmurphy@ics.uci.edu

Michael J. Pazzani,
pazzani@ics.uci.edu

Department of Information & Computer Science
University of California, Irvine, CA 92717

ABSTRACT

This paper describes CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. CLIPS-R may be used for a variety of knowledge-base revision tasks, such as refining a prototype system, adapting an existing system to slightly different operating conditions, or improving an operational system that makes occasional errors. We present a description of how CLIPS-R revises rule-bases, and an evaluation of the system on three rule-bases.

INTRODUCTION

Considerable progress has been made in the last few years in the subfield of machine learning known as theory revision, e.g. [1,2,3]. The general goal of this area is to create learning models that can automatically update the knowledge base of a system to be more accurate on a set of test cases. Unfortunately, this progress has not yet been put into common practice. An important reason for the absence of technology transition is that only a restricted form of knowledge bases have been addressed. In particular, only the revision of logical knowledge bases that perform classification tasks with backward chaining rules [4] has been explored. However, nearly all deployed knowledge-based systems make use of forward-chaining production rules with side effects. For example, two of the knowledge-based systems reported on at the 1993 Innovative Applications of Artificial Intelligence use CLIPS. The remainder of the knowledge-based systems use ART, a commercial expert system that has many of the same features as CLIPS.

There are a variety of practical reasons why the production rule formalism is preferred to the logical rule formalism in deployed expert systems. First, production rules are suitable for a variety of reasoning tasks, such as planning, design and scheduling in addition to classification tasks that are addressed by logical rules. Second, most deployed knowledge-based systems must perform a variety of computational activities such as interacting with external databases or printing reports in addition to the “reasoning” tasks. The production system formalism allows such procedural tasks to be easily combined with the reasoning tasks. Third, the production rule systems tend to be computationally more efficient. The production systems allow the knowledge engineer to have more influence over the flow of control in the systems allowing the performance to be fine tuned. Whereas in a logical system, the rules indicate what inferences are valid, in a production system, the rules indicate both which inferences are valid and which inferences should be made at a particular point.

The revision of CLIPS rule-bases presents a number of challenging problems that have not been addressed in previous research on theory revision. In particular, rules can retract facts from working memory, display information, and request user input. New opportunities to take advantage of additional sources of information also accompany these new problems. For example, a user might provide information that a certain item that was displayed should not have been, or that information is displayed in the wrong order.

In the remainder of this paper, we give an overview description of CLIPS-R, a system for revising CLIPS rule-bases and an evaluation of CLIPS-R on three rule-bases.

DESCRIPTION

CLIPS-R has an iterative refinement control structure (see Figure 1). The system takes as input a rule-base and a set of instances that define constraints on the correct execution of the rules in the rule-base. While there are unsatisfied constraints CLIPS-R heuristically identifies a subset of similar instances as problem instances (instances with many unsatisfied constraints). Using the problem instances, a set of potential repairs to the rule-base are heuristically identified. Each of these repairs is used to temporarily modify the rule-base, with each modified rule-base evaluated over all instances. The repair that improves the rule-base most is used to permanently modify the rule-base. If no repair can increase the evaluation of the rule-base, than the addition of a new rule through rule induction is attempted. If rule induction cannot generate a rule that can improve the evaluation of the rule-base, the revision process halts and the latest rule-base is returned. The process of modification and rule induction continues until all constraints are satisfied or until no progress is made.

[Figure Deleted]

Figure 1. System Organization.

Instances

Each instance has two components: initial state information and constraints on the execution of the rule-base given the initial state. The initial state information consists of a set of initial facts to be loaded into the fact-list before execution of the rule-base, and a set of bindings that relate input function calls to their return values. From the automobile diagnosis rule-base, the function *ask-question* with argument “What is the surface state of the points?” may return “burned” for one instance and “contaminated” for another instance. The set of constraints on the execution of the rule-base includes constraints on the contents of the final fact-list (the final fact-list is the fact-list when execution of the rule-base halts), and constraints on the ordering of observable actions such as displaying data or asking the user questions.

Rule-Base Evaluation

The metric used to evaluate of the rule-base (relative to a set of instances) is the mean error rate across all instances. The error rate for an instance is the percentage of constraints unsatisfied by the execution of the rule-base. An instance is executed in the following manner.

- Reset the rule-base.
- Assert any initial facts.
- Associate bindings with user-defined functions.
- Execute the rule-base until either the agenda is empty or until a user-defined rule execution limit is reached.

During execution of the rule-base for a particular instance, trace information is recorded that is used to determine how many of the constraints associated with the instance are unsatisfied.

Repair Operators

The set of potential repairs to a rule-base are, LHS specialization and generalization (the addition and deletion of conditional elements to the LHS of a rule), action promotion and demotion (the decrease and increase of the embeddedness of an action within if-than-else function, salience modification, assert and retract addition and deletion, observable action modification, rule deletion and rule induction.

Repair Identification

Below is a brief example of how repairs for a particular problem instance are identified. For a more thorough description of these and other aspects of CLIPS-R, see [5]. Assume a problem instance is identified that has an error because the final fact-list contains the extra fact *(repair "add gas")*. The heuristics that suggested repairs for an
extra fact are described below.

- The action within the rule that asserted this fact should be deleted.
- Add a retract for the fact to a previously fired rule.
- For each unfired rule in the rule-base that has a retract that could retract this fact, identify repairs that would allow that rule to fire.
- Identify repairs that could cause the rule that asserted the fact to not fire.

EMPIRICAL ANALYSIS

A series of experiments were designed to analyze various characteristics of CLIPS-R. With the first rule-base, automobile diagnosis (distributed with CLIPS), we perform experiments to determine how well CLIPS-R does at increasing the accuracy of randomly mutated rule-bases. For the second domain, we deal with the nematode identification rule-base. With this rule-base, we show how CLIPS-R can be used to extend a rule-base to handle new cases. For the final rule-base, student loan [2], a problem translated from PROLOG to CLIPS, we chow that CLIPS-R is competitive with an existing revision system, FOCL-FRONTIER [6], that is designed to revise the Horn clause rule-bases.

Automobile Diagnosis Rule-Base

The auto diagnosis rule-base is a rule-base of 15 rules. It is an expert system that prints out an introductory message, asks a series of questions of the user, and prints out a concluding message including the predicted diagnosis.

Cases were generated from 258 combinations of responses to the user query function. Each instance consisted of a set of answers for each invocation of the query function as initial state information, a single constraint on the final fact-list and an ordering constraint for the sequence of printout actions. The target constraint for each instance was a positive constraint for a repair fact, e.g. *(repair “Replace the points").*

Execution of an instance for the auto diagnosis rule-base consisted of clearing the fact-list, setting the bindings that determine the return values for each function call instance (to simulate user input for the user query function) and executing the rule-base to completion. The bindings that associate function calls to their return values allowed an otherwise interactive rule-base to be

run in batch mode. This is necessary because no user would be willing to answer the same questions for 50 instances on different variations of the rule-base.

The experiments performed using the auto diagnosis rule-base were designed to determine how well CLIPS-R could do at revising mutated versions of the correct rule-base. Mutations consisted of extra, missing or incorrect conditional elements or actions and incorrect rule salience values. Two sets of 20 mutated rule-bases were randomly generated with one set of rule-bases having only a single mutation and the other set having three mutations per rule-base. Each mutated rule-base was revised using a random set of 50 training instances. The remaining instances were used for testing. Figure 2 contains a scatter plot showing the initial error of each mutated rule-base and the final error after revision of the rule-base.

[Figure Deleted]

Figure 2. Target Error After Revision as a function of Target Error Before

An analysis of the scatter plots in Figure 2 shows that, for the most part, CLIPS-R is able to reduce the error rates of the mutated rule-bases (points below the diagonal indicate a decrease in error). For one mutation, the average rule-base error was 11.2% before learning and 0.7% after learning. With three mutations, the error before learning was 28.0% and after learning it was 7.2%.

Nematode Identification Rule-Base

The second rule-base, nematode identification (a nematode is a class or phylum of worm), has 93 rules. The intent in presenting this rule-base, is show an example of how CLIPS-R can be used to extend a classification rule-base to handle new cases. The basic requirements for this problem are a rule-base, a set of cases that the rule-base correctly classifies, and a set of cases that are not correctly classified by the rule-base.

For the nematode rule-base, because no cases were provided with the original rule-base, a set of 50 cases were generated by interactively running the rule-base over different responses to the user query function. In order to simulate a rule-base that is in need of extension, two errors were introduced into the rule-base. Specifically, one rule was deleted (a rule that would normally assert the classification for two of the 50 cases), and a second rule was changed, so that it fired and asserted an incorrect classification for the two cases no longer classified by the deleted rule, see Table 1. The two cases that are misclassified by the mutated rule-base, are the cases that CLIPS-R needs to extend the rule-base to cover.

```
(defrule Pratylenchus
   ?f1 <- (esophagus-glands-overlap-intestine ventrally)
   ?f2 <- (ovary 1)
   =>
   (retract ?f1)
   (retract ?f2)
   (assert (nematode pratylenchus))
   (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                        "2. ovary 1."
                        "3. head-shape low and flat.")))
```

(a) Deleted rule.

```
(defrule Hirshmanniella
   ?f1 <- (esophagus-glands-overlap-intestine ventrally)
   ;;; ?f2 <- (ovary 2)
```

```
   =>
   (retract ?f1)
   ;;; (retract ?f2)
   (assert (nematode hirshmanniella))
   (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                        "2. ovary 2."
                     "3. head-shape low and flat.")))
```

(b) Rule with deleted conditional element *(ovary 2)* and retract.

Table 1. Mutated Rules.

When provided with the set of 50 cases and the mutated rule-base, CLIPS-R extends the rule-base to handle the new cases as follows. First, the two misclassified cases are identified by CLIPS-R as the problem cases. Second a set of repairs are identified and evaluated over all 50 cases. After completing the evaluations the repair that specialized the rule *Hirshmanniella* to include *(ovary 2)* as a conditional element is selected as the best repair because it is the repair that most decreased the error rate over all 50 cases. Upon permanently adding *(ovary 2)* to the rule *Hirshmanniella*, the two cases, previously misclassified, are unclassified by the revised rule-base. After completion of a second set of repair evaluations with no success at reducing error rate, rule induction is successfully used to fix the unclassified cases, see Table 2.

```
(defrule G123091
   (esophagus-glands-overlap-intestine ventrally)
   (ovary 1)
   =>
   (assert (nema-id pratylenchus)))
```

(a) New rule.

```
(defrule Hirshmanniella
   ?f1 <- (esophagus-glands-overlap-intestine ventrally)
   (ovary 2)
   =>
   (retract ?f1)
   (assert (nematode hirshmanniella))
   (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                        "2. ovary 2."
                        "3. head-shape low and flat.")))
```

(b) Revised rule with conditional element *(ovary 2)* added.

Table 2. Revised Rules.

Note the difference between the original rules shown in Table 1 and the revisions of the mutated rules shown in Table 2. The revised *Hirshmanniella* rule differs from the original rule by the absence of a retract for the fact matching the *(ovary 2)* conditional element. The set of 50 test cases were insufficient to recognize that a retract was missing. A similar problem is true for the induced rule *G123091*. This rule was was added by CLIPS-R to take the place of the the deleted rule *Pratylenchus*. While this rule asserts a classification that is correct with respect to the test cases, *(nema-id pratylenchus*), it is not quite the same assertion made by the deleted rule, *(nematode pratylenchus)* (if (*nematode pratylenchus*) had been asserted by *G123091*, it would later be replaced by the fact *(nema-id pratylenchus)*). In short, the results of this experiment highlight the need for a comprehensive library of test cases.

Student Loan Rule-Base

In the original form, the student loan domain consists of a set of nine rules (represented as Horn clauses) and a set of 1000 cases. The rule-base contains four errors (an extra literal, a missing literal, an extra clause and a missing clause). The initial theory has an error of 21.6%. In order to use this rule-base with CLIPS-R, the nine Horn clause rules were converted into nine production rules, each with a single assert action. Multiple clauses in the Horn clause rules were converted to a disjunction of conjuncts within a CLIPS production rule.

Execution of a case for the student loan rule-base consisted of asserting into an empty fact-list a set of facts specific to the case and then executing the rule-base to completion. All results for the following experiments are averages of 20 runs. All cases not used for training are used for testing.

Num. Cases	FOCL % Error	CLIPS-R % Error
25	11.8	12.6
50	5.8	3.0
75	2.8	2.1

Table 3. A Comparison of FOCL-FRONTIER and CLIPS-R.

The experiment performed was to determine how well CLIPS-R performed at revising the rule-base relative to FOCL-FRONTIER. Table 3 shows that the error rate is competitive with that of FOCL-FRONTIER on this problem. Only with 50 training examples is the difference in error significant (p $<$ .05).

FUTURE WORK

CLIPS-R is still in its infancy and we expect many of the details of the individual operators and heuristics to change as this work matures. Future directions include solutions to the issues that arose when revising the nematode rule, e.g. a better language for representing constraints on the correct execution of the rule-base, and the use of rule clustering, rule-models and rule-groups to guide the revision and induction of rules. Additional research could include a greater understanding of the distributions of rule-base coding styles, automated rule-base understanding systems, and revision strategies that simulate the methods by which humans manually revise rule-bases.

CONCLUSION

We have described CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. Novel aspects of CLIPS-R include the ability to handle forward chaining theories with “nonlogical” operations such as rule saliences and the retraction of information from working memory. The system is organized with an iterative refinement control structure that identifies a set of similar problematic instances, identifies repairs that can fix the errors associated with the instances, and than evaluates each repair to identify the repair that best improves the rule-base. CLIPS-R can take advantage of a variety of user specified constraints on the correct processing of instances such as ordering constraints on the displaying of information, and the contents of the final fact-list. In addition, CLIPS-R can operate as well as existing systems when the only constraint on processing an instance is the correct classification of the instance.

ACKNOWLEDGMENTS

The research reported here was supported in part by NSF Grant IRI-9310413, ARPA Grant F49620-92-J-0430, and AFOSR AASERT grant F49620-93-1-0569.

REFERENCES

1. Ourston, D. & Mooney, R., “Changing the Rules: A Comprehensive Approach to Theory Refinement,” Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI-90, 1990, 815-820.

2. Pazzani, M.J. & Brunk, C., “Detecting and Correcting Errors in Rule-Based Expert Systems: An Integration of Empirical and Explanation-Based Learning,” In Knowledge Acquisition, 3, 1991, 157-173.

3. Wogulis, J. & Pazzani, M.J., “A Methodology for Evaluating Theory Revision Systems: Results with AUDREY II,” Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI93, 1993, 1128-1134.

4. Clancey, W., ``Classification problem solving, “Proceedings of the National Conference on Artificial Intelligence,” 1984, 49-55.

5. Murphy, P.M. & Pazzani, M.J., “Revision of Production System Rule-Bases,” Machine Learning: Proceedings of the Eleventh International Conference, 1994, 199-207.

6. Pazzani, M.J. & Brunk, C., “Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning,” Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI93, 1993, 328-334.

DAI-CLIPS: DISTRIBUTED, ASYNCHRONOUS, INTERACTING CLIPS

Denis Gagné & Alain Garant
Groupe de Recherche en Intelligence Artificielle Distribuée
Collège militaire royal de Saint-Jean
Richelain, (Québec)
Canada, J0J 1R0
dgagne@cmr.ca

ABSTRACT

DAI-CLIPS is a distributed computational environment within which each CLIPS is an active independent computational entity with the ability to communicate freely with other CLIPS. Furthermore, new CLIPS can be created, others can be deleted or modify their expertise, all dynamically in an asynchronous and independent fashion during execution. The participating CLIPS are distributed over a network of heterogeneous processors taking full advantage of the available processing power. We present the general framework encompassing DAI-CLIPS and discuss some of its advantages and potential applications.

INTRODUCTION

Scenarios to be solved by Artificial Intelligence (AI) applications rapidly increase in complexity. If we are to even allude to the enormously difficult endeavor that represents Artificial Intelligence, very flexible, robust and powerful computational systems are going to be needed. These programs, and the processing architecture supporting them, will have to be able to cope with a very wide range of dynamic external demands that are simply unpredictable.

Conditions vary greatly across tasks instances and constantly dictate different accomplishment strategies and usage of disparate sources of expertise. To be effective in such situations, fusion must take place at many levels (information, expertise, processes, …). Maximum degree of openness and flexibility is required of AI systems. Both hardware architecture and software solutions must allow for scalable performance and scalable functionalities. This, in return, will allow the matching of AI systems to the needs of the situation as well as gaining access to the latest technological advances.

We believe that complex problems can best be solved via a pendemonium of smaller agents. Each agent specializes in a different narrow aspect of cognition or field of knowledge [17, 18]. Emphasis is thus placed on data and application parallelism.

The computational environment presented herein integrates the simplicity, elegance and expressiveness of the ACTOR computational model [3, 12] with the exceptional processing power of heterogeneous distributed and parallel architectures [5]. Expressiveness is increased by providing for disjunct or even disparate sources of expertise to cohabitate rather than trying to integrate them. The interfacing or fusion required is achieved via message passing enabling asynchronous exchange of knowledge (facts, rules) among agents. Agents in the present context are effectively extended CLIPS. CLIPS is an expert system tool that was developed by NASA at the Software Technology Branch of the Johnson Space Center [10].

DAI-CLIPS is a distributed computational environment within which each extended CLIPS is an active independent computational entity with the ability to communicate freely with other CLIPS. Furthermore, new CLIPS can be created, others can be deleted or modify their expertise, all dynamically in a totally asynchronous and independent fashion during execution.

The remainder of this text is structured as follows. Section 2 highlights the key characteristic of the Actor computational model that influenced DAI-CLIPS and briefly describes the system layer that constitutes the foundation of DAI-CLIPS. In section 3, we provide a description of the conceptual framework offered by DAI-CLIPS by outlining its global functionalities. In section 4, we provide some details about the architecture and available functions. Section 5 brushes a quick picture of a few potential areas of research and development that could benefit from such computational environment. Finally, sections 6 and 7 provides a discussion and our conclusions on the subject.

THE ACTOR MODEL & CLAP

In this section we highlight some of the main characteristics of the Actor model that influenced and characterizes DAI-CLIPS. We then present a brief description of CLAP [5, 6, 7], a system layer based on the Actor Model, that constitute the foundation of DAI-CLIPS.

The Actor Model

A detailed description of the Actor Model can be found in [2, 13]. We will only discuss here a few of the more salient characteristics of the model:

- **Distributed**. The Actor model consists of numerous independent computational entities (actors). The actors process information concurrently, which permits the overall system to handle the simultaneous arrival of information from different outside sources.

- **Asynchronous**. New information may arrive at any time, requiring actors to operate asynchronously. Also, actors can be physically separated where distance prohibits them from acting synchronously. Each actor has a mailbox (buffer) where messages can be stored while waiting to be processed.

- **Interactive**. The Actor model is characterized by a continuous information exchange through message passing, subject to unanticipated communication from the outside.

Thus, an actor is basically an active independent computational entity communicating freely with other actors.

There are at least two different ways to look at the Actor model. Viewed as a system, it is comprised of two parts: the actors and a system layer (operating/management system). From such a point of view the actors are the only available computational entities. The system layer is responsible to manage, create and destroy actors as required or requested. The system layer is also responsible for ensuring message passing among actors.

Viewed as a computational entity, an actor also comprises two parts: a script, that defines the behaviors of the actor upon receipt of a message; and a finite set of acquaintances which are the other actors known to the actor.

CLAP

The above viewpoint duality is preserved in CLAP[1]. CLAP is an implementation of an extension of the actor model that can execute on a distributed heterogeneous network of processors. The present version of CLAP can execute over a network of SUN SPARC workstations and Alex

[1] **C**++ **L**ibrary for **A**ctor **P**rogramming

Informatique AVX parallel machines which are transputer based distributed memory machines [5]. A port to HP and SGI workstations is in progress.

CLAP is an object-oriented programming environment that implements the following concepts of the Actor model: the notion of actor, behaviors, mailbox, and parallelism at the actor level. Further, CLAP offers the extension to the model of intra-actor parallelism.

Generally, CLAP applications will consist of many programs distributed over available processors executing as a task under the control of the CLAP run time environment. In CLAP, each actor is a member of a given task. It is up to the programmer to determine how many actors there will be for any given task (although, a large number of actors in a single task could mean the loss of potential parallelism in the application.) A scheduler controls the execution of processes inside the tasks. Each task possesses a message server that handles message reception for the actors in the task. Inter-processor message transmissions are handled via RPC servers. XDR filters and type information are utilized for the encoding and decoding of these messages. The CLAP environment is implemented in C++.

THE CONCEPTUAL FRAMEWORK

DAI-CLIPS is a distributed computational environment within which each CLIPS has been extended to become an active independent computational entity with the ability to communicate freely with other extended CLIPS. Furthermore, new extended CLIPS can be created, others can be deleted or modify their expertise, all dynamically in a totally asynchronous and independent fashion during execution.

The *desirata* behind DAI-CLIPS is to produce a flexible development tool that captures the essence of the "aggregate of micro agents'" thesis supported by many in the study of Computational Intelligence and Cybernetic [12, 17, 18]. The underlying thesis being to have "micro agents," in our case complete CLIPS, specialized in different very narrow aspects of cognition or fields of knowledge[2]. As they go about their tasks, these micro-agents confer with each other and form coalitions producing collated, revised enhanced views of the raw data they take in. These coalitions and their mechanism implement various cognitive processes leading to the successful resolution of the problem.

D.A.I.

The three highlighted characteristics of the Actor model in the previous section, namely distributed, asynchronous and interactive, are at the basis of the conceptual framework for DAI-CLIPS. The augmented CLIPS participating in the environment are completely encapsulated and independent allowing their distribution at both the software and hardware level. Meaning that not only can the CLIPS execute in parallel but they can also be physically distributed over the network of available processors. Our present version of DAI-CLIPS can have participating CLIPS distributed over a network of SPARC workstations and/or the nodes of a transputer based distributed memory parallel machine. The interaction among the CLIPS is asynchronous (synchronicity can be imposed when required). The interchange of knowledge between these extended CLIPS can involve exchanging facts, rules, and any other CLIPS data object or functionality.

---

[2] Expert Systems excel under these domain constraints.

Cooperation

The DAI-CLIPS environment is conducive of cooperation among a set of independent CLIPS. We regard as cooperation any exchange of knowledge among CLIPS whether productive or not.

There is *a priori* no pre-defined notion of an organizational structure among the CLIPS in DAI-CLIPS. Any desired type of organization (*e.g.* hierarchy, free market, assembly line, task force, *etc*.) can be achieved by providing each CLIPS the appropriate knowledge of the structure and the mechanism or protocol to achieve it.

The broad definition of cooperation and the inexistence of pre-defined organizational structures in DAI-CLIPS were conscious initial choices. We wanted to maintain the highest flexibility possible for the environment in this first incarnation. We are contemplating the introduction of mechanisms to DAI-CLIPS to ease the elaboration of specific types of organizations based on the premise of groups or aggregates. The aim of these efforts is to capture the recursive notion of *agency*.

Dynamic Creation

A powerful capability of DAI-CLIPS is the possibility of dynamically generating or destroying participating CLIPS at run time. The generation of new CLIPS can involve introducing a new expertise or simply cloning an existing participant. When generating a new CLIPS, one can specify which expertise the CLIPS is to possess by indicating the appropriate knowledge base(s) to be loaded in the CLIPS at creation. This functionality has enormous potentials that we have yet to completely explore.

THE ARCHITECTURE

The general framework encompassing DAI-CLIPS can be viewed as four layers: the hardware layer, the system layer, the agent layer, and the application layer (see figure 1). The hardware layer consists of a set of nodes (available processors) on the network (SPARCs and transputers). The system layer (CLAP) is responsible to manage, create and destroy the processes required or requested from the above agent layer as well as managing inter-agent communications. The agent layer provides a series of functionalities to implement various cognitive processes via coalitions and organization mechanisms for a series of specialized micro-agents (DAI-CLIPS). Finally, the application layer provides interfacing services and captures and implements the user's conceptualization of the targeted domain. Such conceptualization usually involves the universe of discourse or set of objects presumed in the domain, a set of functions on the universe of discourse, and a set of relations on the universe of discourse [9].

[Figure Deleted]

Figure 1. The conceptual framework.

Within this general framework, an application designer can directly access and manipulate any of the four layers of the environment. This provides the designer with the flexibility of manipulating objects at the level of abstraction he is more at ease with. For example, a more advanced application designer could seek efficiency in his particular application by manipulating objects all the way down to the system layer level, where someone else may be quite content of the functionalities provided at the top layer.

Design

In the present version of the environment, each augmented CLIPS is associated with a CLAP actor. These actors are loaded on different available processing nodes according to the load of the nodes. To each augmented CLIPS is connected an interface which provides access to the individual standard command loops of the CLIPS. An initial knowledge base is loaded in each CLIPS (see figure 2). Note that it is possible for two CLIPS to be uploaded with the same initial knowledge base or for a CLIPS to upload a supplementary knowledge base at run time.

[Figure Deleted]

Figure 2. Surrounding environment of DAI-CLIPS.

Implementation

In this section we enumerate some of the functions specific to DAI-CLIPS and describe their respective use and functionality. The list is not exhaustive[3], rather the intent here is to present some of the main functions which can be used directly by the user.

- **create-agent**

  Purpose: Creates a named agent without expertise.

  Synopsis: (create-agent <string-or-symbol-agent-name>)

  Behavior: The agent <string-or-symbol-agent-name> is created.

- **destroy-agent**

  Purpose: Destroys a named agent.

  Synopsis: destroy-agent <string-or-symbol-agent-name>)

  Behavior: The agent <string-or-symbol-agent-name> is destroyed.

- **send-fact-agent**

  Purpose: Asserts a run-time fact in a named agent.

  Synopsis: send-fact-agent <string-or-symbol-agent-name> <string>)

  Behavior: The fact <string> is asserted in the agent <string-or-symbol-agent-name> who then executes.

- **send-deffact-agent**

  Purpose: Defines a persistent fact in a named agent.

  Synopsis: (send-deffact-agent <string-or-symbol-agent-name> <symbol-deffact-name>)

[3] Due to restricted space in this article.

Behavior: The fact <symbol-deffact-name> is permanently asserted in the agent <string-or-symbol-agent-name> who then executes.

- **send-defglobal-agent**

  Purpose: Defines a global variable in a named agent.

  Synopsis: (send-defglobal-agent <string-or-symbol-agent-name> <symbol-defglobal-name>)

  Behavior: The global variable <symbol-defglobal-name> is defined in the agent <string-or-symbol-agent-name> who then executes.

- **send-defrule-agent**

  Purpose: Defines a rule in a named agent.

  Synopsis: (send-defrule-agent <string-or-symbol-agent-name> <symbol-defrule-name>)

  Behavior: The rule <symbol-defrule-name> is added in the expertise of agent <string-or-symbol-agent-name> who then executes.

- **send-deftemplate-agent**

  Purpose: Defines a template in a named agent.

  Synopsis: (send-deftemplate-agent <string-or-symbol-agent-name> <symbol-deftemplate-name>)

  Behavior: The template <symbol-deftemplate-name> is added in the expertise of agent <string-or-symbol-agent-name> who then executes.

- **load-for-agent**

  Purpose: Send a message to a named agent ordering him to load a specific expertise from a named file.

  Synopsis: (load-for-agent <string-or-symbol-agent-name> <file-name>)

  Behavior: The agent <string-or-symbol-agent-name> possesses the expertise specified in <file-name>.}

POTENTIAL AREAS OF APPLICATIONS

DAI-CLIPS provides an environment with a varying number of autonomous knowledge based systems (expert-systems) that can exchange knowledge asynchronously. Such organizations of interconnected and independent computational systems are what Hewit calls **Open Systems**[4] [14]. We thus refer to DAI-CLIPS as an *Open Knowledge Based Environment* or *Open KBE* for short. The potential areas of research and development that could benefit from such a computational environment are considerable.

---

[4] The term “open system” being an overloaded term, we specifically refer to Hewit's definition of open systems within the context of this article.

Distributed Artificial Intelligence

The first such area that comes to mind is that of Distributed Artificial Intelligence (DAI) [3, 8, 15]. The facilities available in DAI-CLIPS to support interaction between "intelligent" agents make it a flexible tool for DAI applications and research. In comparison with some existing DAI test beds, DAI-CLIPS: does not impose a specific control architecture such as the blackboard in GBB [4]; does not restrict agents to a specific set of operators as in TRUCKWORLD; and is not a domain specific simulator as in PHOENIX [11]. The most closely related work is SOCIAL CLIPS [1]. The major difference with SOCIAL CLIPS is DAI-CLIPS' dynamic creation and destruction of participating CLIPS at run time.

There are *a priori* no predefined domain of application for DAI-CLIPS. A designer is free to specialize his agents in the domain of his/her choice. Further, the agents can be heterogeneous in their speciality (expertise) within a single application. The only imposed commonality in DAI-CLIPS is the use of the extended CLIPS shell. The added power provided by the dynamic creation/destruction of agents within DAI-CLIPS is the source of the potential area of application proposed in the next section.

Evolutionary Computing

The principle behind evolutionary computing is that of population control [16]. That is ensuring that the population is not allowed to grow indefinitely by selectively curtailing it. This population control is carried out by creative and destructive processes guided by natural selection principles. The destructive process examines the current generation (population) and curtails it by destroying its weakest members (usually those with the lowest values from some predefined fitness measure). The creative process introduces a new generation created from the survivors of the destructive process. The expected result is that of a better fit or optimum population.

Given DAI-CLIPS capability of dynamically creating and destroying participating CLIPS at run-time, one can begin to explore the potential of coarse grain evolutionary computing. That is, applying evolutionary computing principles to a population of "agents" or expert systems in order to optain a population of expert systems that selectively better perform on a global task in accordance with some selected fitness measure. The creative and destructive processes could be carried out by two independent agents. One agent evaluating the agents of the population and destroying those that do not perform as expected (destructive process), another, either bringing together the expertise of two fit agents into a newly created expert or simply cloning a fit agent (creative process). We will refer to this approach as a *disembodied genetic mechanism*. Alternatively, the agents of a population could themselves possess "genetic knowledge" that would lead to self-evaluation. Based on the knowledge of its own fitness, an agent could then decide to terminate operations or to seek an appropriate agent for procreation (via mutation, crossover, *etc*.). This *embodied genetic mechanism* could take place based on some pre-determined evolution cycle. Note that both the embodied and disembodied genetic mechanisms can take place continuously and in totally asynchronous fashion.

DISCUSSION

Open KBEs such as the one presented herein have considerable advantages:

- they allow independent systems to cooperate in solving problems;
- they allow disparate participant systems to share expertise;

- they allow for the presence of incoherent information among participant systems, no need for global consistency;
- they provide for participant systems to work in parallel on common problems;
- participant systems can be distributed physically to make ultimate use of the available processing power;
- asynchronous communication ensures very remote chances of deadlock;
- fault tolerance is easy to implement via system redundancy;
- participant systems can be developed and implemented independently and modularly;
- participant system are reusable in other applications;

and many others.

By choice, DAI-CLIPS has one limitation with respect to Open KBE: the participant systems are limited to CLIPS based systems. In fact, the general framework encompassing DAI-CLIPS can easily be extended to allow heterogeneous applications (*e.g.* other ES shells, Data Bases, Procedural applications) to participate through the use of a common formal language for the interchange of knowledge among disparate computer programs such as *Knowledge Interface Format* (KIF) and the use of a common message format and message-handling protocol such as the *Knowledge Query and Manipulation Language* (KQML). The use and adherance to these two upcoming standards from the DARPA Knowledge Sharing Initiative can assure that any incompatibility in the paticipant systems' underlying models for representing data, knowledge and commands can be ironed out to attain the desired higher level of openness.

DAI-CLIPS and its encompassing environment will be the source of more research and enhancements. We are presently putting the final touch to a second version of DAI-CLIPS that implements the notion of multiple behaviors from the Actor model. That is, the capability of an agent to change its behavior in order to process the next message. Effectively, a CLIPS shell will possess different expertise and will “context switch” to make use of the appropriate knowledge to process the received message.

CONCLUSION

We introduced DAI-CLIPS, a distributed computational environment within which each CLIPS is an active independent computational entity communicating freely with other CLIPS. Open KBEs such as this one have many advantages, in particular, they allow for scalable performance and scalable functionalities at both the hardware and the software level. The potential applications of such environments are considerable. The unique power of dynamic creation and destruction of DAI-CLIPS could lead to new forms of “intelligent” evolutionary systems.

ACKNOWLEDGMENTS

The authors would like to thank Jocelyn Desbiens and the members of the Centre de Recherche en Informatique Distribuée (CRID) for their constant support in the implementation of DAI-CLIPS. We also want to thank Alain Dubreuil and André Trudel for comments on an earlier version of this article.

BIBLIOGRAPHY

1. Adler, R., “Integrating CLIPS Applications into Heterogeneous Distributed Systems.”, In *Proceedings of the Second CLIPS Conference*, NASA Conference Publication 10085, 1991.

2. Agha, G.A., *Actors: A Model of Concurrent Computation in Distributed Systems*, Cambridge, Massachusetts: MIT Press, 1986.

3. Bond, A. & Gasser, L. (Eds), *Readings in Distributed Artificial Intelligence*., Los Altos, California: Morgan Kaufmann, 1988.

4. Corkill, D., Gallagher, K. & Murray, K., “GBB: A Generic Blackboard Development System.” In *Proceedings of AAAI-86*, 1986.

5. Desbiens, J., Toulouse, M. & Gagné, D., “CLAP: Une implantation du modèle Acteur sur réseau hétérogène”., In *Proceedings of the 1993 DND Workshop on Knowledge Based Systems/Robotics*. Ottawa, Ontario, 1993. (In French)

6. Gagné, D., Nault, G., Garant, A. & Desbiens, J., “Aurora: A Multi-Agent Prototype Modelling Crew Interpersonal Communication Network”., In *Proceedings of the 1993 DND Workshop on Knowledge Based Systems/Robotics*. Ottawa, Ontario, 1993.

7. Gagné, D., Desbiens, J. & Nault, G., “A Multi-Agent System Simulating Crew Interaction in a Military Aircraft”., In *Proceedings of the Second World Congress on Expert Systems*. Estoril, Portugal, 1994.

8. Gasser, L. & Huhns, M.N. (Eds), *Distributed Artificial Intelligence: Volume II*, Los Altos, California: Morgan Kaufmann, 1989.

9. Genesereth, M. & Nilsson, N., *Logical Foundations of Artificial Intelligence*., Morgan Kaufmann, 1987.

10. Giarratano, J. & Riley, G., *Expert Systems: Principles and Programming*., PWS Publishing Company, 1994.

11. Hanks,S., Pollack, M. & Cohen, P., “Benchmarks, Test Beds, Controlled Experimentation and the Design of Agent Architectures.”, In *AI Magazine*, Vol. 14, No. 4, Winter 1993.

12. Hewit, C., Bishop, P. & Steiger, R., “A Universal Modular Actor Formalism for Artificial Intelligence.”, In *Proceedings of the 3rd Joint Conference on Artificial Intelligence (IJCAI73)*., Stanford, California, 1973.

13. Hewit, C., “Viewing Control Structures as Pattern of Passing Messages.”, In *Journal of Artificial Intelligence*., Vol 8, No. 3, 1977.

14. Hewit, C., “The Challenge of Open Systems.”, *BYTE Magazine*, Vol. 10, No. 4, April 1985.

15. Huhns, M.N. (Ed), *Distributed Artificial Intelligence*, Los Altos, California: Morgan Kaufmann, 1987.

16. Koza, J., *Genetic Programming*., The MIT Press, 1993.

17. Minsky, M., *The Society of the Mind*., Simon and Schuster, New York, 1985.

18. Tenney, R. & Sandell, JR., “Strategies for Distributed Decisionmaking.” In Bond & Gasser (Eds) *Readings in Distributed Artificial Intelligence*., Los Altos, California: Morgan Kaufmann, 1989.

PCLIPS: PARALLEL CLIPS

Lawrence O. Hall[1] , Bonnie H. Bennett[2] , and Ivan Tello

Hall & Tello: Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
hall@csee.usf.edu

Bennett: Honeywell Technology Center
3660 Technology Driver MN65-2600
Minneapolis, MN 55418
bennett@src.honeywell.com

ABSTRACT

A parallel version of CLIPS 5.1 has been developed to run on Intel Hypercubes. The user interface is the same as that for CLIPS with some added commands to allow for parallel calls. A complete version of CLIPS runs on each node of the hypercube. The system has been instrumented to display the time spent in the match, recognize, and act cycles on each node. Only rule-level parallelism is supported. Parallel commands enable the assertion and retraction of facts to/from remote nodes working memory.

Parallel CLIPS was used to implement a knowledge-based command, control, communications, and intelligence (C^3I) system to demonstrate the fusion of high-level, disparate sources. We discuss the nature of the information fusion problem, our approach, and implementation. Parallel CLIPS has also been used to run several benchmark parallel knowledge bases such as one to set up a cafeteria. Results shown from running Parallel CLIPS with parallel knowledge base partitions indicate that significant speed increases, including superlinear in some cases, are possible.

INTRODUCTION

Parallel CLIPS (PCLIPS) is a rule-level parallelization of the CLIPS 5.1 expert system tool. The concentration on rule-level parallelism allows the developed system to run effectively on current multiple instruction multiple data (MIMD) machines. PCLIPS has been tested on an Intel Hypercube iPSC-2/386 and I860. Our approach bears similarities in focus to research discussed in [12, 6, 7].

In this paper, we will show an example where the match bottleneck for production systems [1, 3] is eased by utilizing rule-level parallelism. The example involves setting up a cafeteria for different functions and is indicative of the possibilities of performance improvement with PCLIPS [13]. A second example of a battle management expert system provides a perspective to real world applications in PCLIPS.

The rest of the paper consists of a description of PCLIPS, a section describing the knowledge bases (and parallelization approaches) of the examples and speed-up results from running them using PCLIPS, and a summary of experiences with parallel CLIPS.

---

[1] This research partially supported by a grant from Honeywell and a grant from the Software section of the Florida High Technology and Research Council.

[2] Also at Graduate Programs in Software, University of St. Thomas, 2115 Summit Ave PO 4314, St. Paul, MN 55105, bhbennett@stthomas.edu.

THE PCLIPS SYSTEM

Based on experience with an early prototype, the design of the PCLIPS user interface models that of CLIPS as much as possible. A small extension to the syntax is used to allow the user to access working memory on each node, add/retract facts or rules to specific nodes, etc. For example, a load command with four processors allocated now takes the form: (0 1, load “cafe”) and (, load “cafe”). The first command will load files cafe0 to node 0 and cafe1 to node 1, and the second command loads files cafe0, cafe1, cafe2, and cafe3 onto nodes 0, 1, 2 and 3. Other commands operate in the same way with (2, facts) bringing in the facts from node 2 and (3 7, rules) causing the rules from processors 3 and 7 to be displayed.

After rule firing is complete in PCLIPS, the amount of time spent by each node in the match, recognize, and act cycles is displayed. The amounts of time are given as percentages of the overall time, which is also displayed. Sequential timings are obtained from running PCLIPS on one node.

A complete version of CLIPS 5.1 enhanced with three parallel operations, xassert, xretract and mxsend, runs on each of the nodes and the host of an iPSC2 hypercube. The host node automatically configures each of the allocated nodes without user intervention when PCLIPS is invoked. The xassert command simply asserts a fact from one node to a remote node’s working memory. For example, (xassert 3 (example fact)) makes its assertion into the working memory of node 3. The general form is (xassert node_number fact_to_assert). To retract a fact from a remote working memory use (xretract node_number fact_to_retract). Both operations build a message and cause it to be sent by the hypercube operating system. Neither command depends upon a specific message passing hardware or software mechanism.

Long messages can take less time to send than many short messages on Intel Hypercubes [2, 8] so mxsend() provides the user with the capability of asserting and/or retracting multiple facts into/from one processor to another processor. The syntax of the function is as follows: (mxsend node_numbers). Mxsend() needs a sequence of calls in order for it to work as desired. The first step in correctly building a message to be used by mxsend() is to call the function clear_fact(). The syntax for this function is as follows: (clear_fact). This function simply resets the buffer used by mxsend() to the ‘~’ character. This character is necessary for a receiving processor to recognize the received message was sent by using mxsend(). The second step is to actually build the message to be sent. In order to do this, a sequence of calls to the function buildfact() should be performed. The syntax of buildfact() is as follows: (buildfact action fact). There are four possible values for the action variable. They are ‘0’, ‘1’, ‘retract’, and ‘assert’. The ‘0’ flag and ‘retract’ will both cause the building of a message to retract a fact (this is done by inserting a ‘$’ character in the message buffer followed by fact), and ‘1’ and ‘assert’ will both cause the building of a message to assert fact (this is done by inserting a ‘#’ character in the message buffer followed by fact). If the following sequence of calls is performed, (buildfact assert Hello World) (buildfact retract PARCLIPS is fun) (buildfact assert Go Bulls!!!) (buildfact assert Save the Earth) the following string will be created: “~#Hello World$PARCLIPS is fun#Go Bulls!!!#Save the Earth” Finally, the function mxsend() can be called. Mxsend() will send the message built to the specified processors so that the message will be processed by the receiving processors. The call (mxsend 10 11 12), will cause the previously built message to be sent to processors 10, 11, and 12. The proper action is taken by the receiving processors who either assert or retract facts into/from their working memory.

Since PCLIPS is a research prototype, the user is free to use or misuse the parallel calls in any way he/she chooses. No safeguards are currently provided. On the other hand, the interface is simple and the calls straightforward. The question that comes to mind is whether they provide enough power to enable useful speed-ups on MIMD architectures. Our current work shows that

they are suitable for obtaining useful speedups [13], if the knowledge base is parallelized in a careful and appropriate way.

EXAMPLES

In this section we show results from parallelizing a knowledge base and discuss a real application for parallel expert systems. All speedups are reported as the sequential time or time on one node divided by the time to process the same set of initial facts and obtain the same set of final facts in parallel (Sequential Time/Parallel Time).

Before discussing examples, we discuss a few guidelines for parallelizing rule bases that have become clear in the course of developing and testing PCLIPS.

PARALLELIZING KNOWLEDGE BASES

There are several approaches that have been taken to parallelizing knowledge bases [5, 7, 9, 11]. An important aspect is that the parallel results be equivalent to the serial results. Methods of explicit synchronization [11] do not seem feasible until communication times are significantly reduced on parallel machines. Hence, we have pursued serialization through rule base modification. This means that the rules in a parallel knowledge base generated under our paradigm are not necessarily syntactically the same as a set of sequential rules.

There are two approaches to parallelizing the rules of a specific sequential knowledge base. The first, and most usual one, is to partition the rules independent of the types of facts they will most likely be used with. In this approach, bottleneck rules that may need to be distributed to multiple processors must be searched for during a sequential trace of the knowledge based system's operation. Processors must be load balanced with an appropriate number of rules. All parallel actions must be inserted into the right hand side of the rules. All facts will be distributed to all nodes under this paradigm.

The second approach to parallelizing the knowledge base is to parallelize it based upon the rules and the expected type of facts. This approach is only feasible if a rule base may be expected to work with one type or set of facts (with the facts themselves changing) in most cases. This approach involves an analysis of the sequential performance of the knowledge based system with a specific set of facts and then a parallelization of the knowledge base for a set of processors. In the limited testing done in our work, this approach to parallelizing rules provides a greater speed-up.

CAFETERIA

There are 93 rules in our version of the cafeteria knowledge base. The rules are grouped into contexts, where an example context involves setting a table. A rule and fact partitioning of the cafeteria knowledge base was done with the use of xassert and xretract and a speedup of 5.5 times was obtained using eight processors. The speedup obtained without using these functions was 6.47 times also using eight processors. Both speedups are less than linear but notice the decrease in speedup when using xassert and xretract. The decrease in speedup is here attributed to inter-processor communication. The time required to decode a message and assert it into working memory is between 1-2 msec [10]. The time used to obtain the above results includes the time required to transmit facts across to other nodes, retract/assert them into working memory, and do the complete inferencing. A single message of 1K takes 1.1 msec to process [2]. Larger messages, however, take considerably more time to process, as shown by Boman and Roose [2]. Since, for these partitions, the messages sent across the nodes are larger than 1Kbyte (every node concatenates approximately 50 35-byte messages, making messages of 1.7 Kbytes

that are sent using mxsend()), and all nodes transmit their messages to the same node (messages might have to wait on intermediate nodes and hence are blocked until memory on the destination node is available to receive the complete message [10, 2]). It is clear from the above that communication is the reason for the decrease in speedup.

The cafeteria knowledge base was also partitioned using 11 and 13 processors. A speedup of 11.5 was obtained using 11 processors, whereas the 13-processor partition produced a speedup of 22.85 times. A fact-based partitioning method was used to obtain both of these partitions. These speed-ups are clearly super-linear and occur because the match percentage of time is reduced in a non-linear fashion by this partitioning approach [13]. Due to space limitations, we will not explore this phenomenon further but refer the reader to our technical report [4].

Finally, several two-processor partitions of cafeteria were performed partitioning the rules only. A speedup of 2.035 times (65.99% matching, 21.11% acting) with two processors was obtained. In this case, rules were copied to each partition unmodified, causing the assertion of facts that are never used by the partition (since the asserted facts enable the firing of a context present in another partition). Partitioning the facts also, the speedup obtained was 2.06 (67.41% matching, 20.26% acting), which is only slightly higher than the speedup obtained when the facts were left intact. Notice that this result suggests that the number of extra unnecessary facts does not significantly affect the overall parallel execution time. A final two-processor partition was performed by modifying the rules left in each partition so that they assert only the context facts needed in the partition. A speedup of 2.13 times was obtained in this case.

BATTLE MANAGEMENT EXPERT SYSTEM

The information fusion problem for battle management occurs when multiple, disparate sensor sources are feeding an intelligence center. This intelligence center is trying to produce timely, accurate and detailed information about both enemy and friendly forces in order for commanders to make effective battle management decisions. The challenge to the C^3I operation is to integrate information from multiple sources in order to produce a unified, coherent account of the tactical, operational or strategic situation.

There has recently been a vast proliferation of fixed and mobile, land- and air-based sensors using acoustic, infrared, radar and other sensor technologies. The result of this proliferation has made more work for the C^3I operation.

Sensors can vary in a variety of dimensions including:

- Coverage Area
- Temporal Characteristics of Coverage
- Field of View
- Angle of View
- Range
- Resolution
- Update Rate
- Detection Probability
- Modality of Imagery
- Degree of Complexity/Realism of Imagery
- Type of Target Information
- Temporal Characteristics of Reports

Each collection system, then, gives a specialized sampling of conditions to a particular level of detail, in specific locations, at a specific point in time, and with a particular level of accuracy. As

a result, the analyst receives information that may be incompatible, fragmentary, time-disordered, and with gaps, inconsistencies and contradictions.

Honeywell's Combat Information Fusion Testbed (CIFT) has been developed to provide the hardware and software environment that can support development of tools powerful enough to assist intelligence analysts in correlating information from widely disparate sources. The current testbed capabilities were chosen for the context of handling three sensors: an airborne moving target indication (MTI) radar, a standoff signal intelligence (SIGINT) system, and an unmanned aerial vehicle (UAV) with a television camera payload. This correlation capability is fundamental for information fusion. By integrating Honeywell's proprietary real-time blackboard architecture (RTBA) with the proprietary spatial-temporal reasoning technique called topological representation (TR), the testbed has been able to perform the data association task. CIFT was developed and tested against a four-hour European scenario involving troop movement in a 40X60 km area that was observed by an MTI radar, a SIGINT system, and a UAV. We determined the target detections and circular error probabilities and time delay that these three systems would be expected to make. CIFT was found to operate effectively on this data, associating reports from the different sensors that had emanated from the same target.

CIFT was then implemented on the Intel iPSC-860 parallel processor [14] producing Parallel-CIFT (or Parallel-CIFT). This processor has eight parallel nodes. There are three major components of the CIFT system: Geographic/Scenario data, Blackboard Control Structures, Spatial/Temporal Reasoners.

Geographic/Scenario Data: These contain the bit maps of the map overlays and the scenario-specific operational and doctrinal data. The current scenario illustrates a Motorized Rifle Regiment in the Fulda area of eastern Germany mobilizing for a road march. This activity includes SIGINT, AUV (airborne unmanned vehicles with video camera payloads), and MTI (moving target indicator radar) sensor reports to a G2 intelligence workstation. The geographic data includes overlays for cities, primary and secondary roads, dense vegetation, and railroads.

Blackboard Control Structures: CLIPS provides the control and representation structures for the blackboard control architecture. Honeywell wrote data structures and fusion rules in the CLIPS format on a Sun workstation. These components were then parallelized and ported to the iPSC-860. Three demonstrations are available: one uses only one of the nodes on the parallel processor (this simulates a traditional serial computer for bench marking purposes), one uses two parallel nodes, and one uses four parallel nodes.

Spatial/Temporal Reasoning: The spatial/temporal reasoner for this system is built on a four-dimensional reasoner developed from Allen's temporal interval reasoning system. It defines the relations that can exist between time and space events and reasons from these primary relations.

This system represents a demonstration of concept of the Parallel CIFT system, a challenging problem in a challenging domain which effectively uses the Parallel CLIPS tool.

Current research efforts include:

- **Auto allocation of parallel components—**This work requires some basic and applied research. We propose using a nearest neighbor shear sort algorithm to dynamically allocate processing tasks across multiple processors. This will balance the load among the processors and ensure optimal performance.

- **Demonstrate P-CIFT and extensions on Paragon**—This requires three preliminary steps: 1) Port CLIPS (the forward-chaining inference engine, on which P-CIFT is built)

to the Paragon, 2) Port P-CIFT to the Paragon, 3) Extensions developed to P-CIFT. See following points.

- **Addition of object-oriented data base (OODB) capabilities**—This should be easily completed with use of the CLIPS 6.0.

- **Development of a domain specific information fusion shell—**Common elements from a variety of information fusion applications (Honeywell currently has Army and Navy scenarios, with plans to extend into commercial domains, medical imaging, robotics, and electronic libraries specifically, in the next year) will be formalized and generalized for future use on other systems. This organic growth of generic components will assure the applicability, generality and usefulness.

- **Multi-hypothesis reasoning—**This will require integration of techniques for multiple hypothesis generation, maintenance, and testing. Previous related work [15] has demonstrated successful approaches in tasks with similar multiple assignment requirements. New research would be required to examine parallel implementation of these approaches. It is likely that a parallel approach could be much more efficient.

- **Quantification of performance results—**Past work has provided demonstrations of concept, but has provided no performance results.

SUMMARY

In this paper, we have discussed a parallel version of the CLIPS 5.1 expert system tool. The parallel tool has a simple interface that is a direct extension of the usual CLIPS interface for parallel use. The tool makes use of rule-level parallelism and has been tested on Intel Hypercubes. Examples of expert systems that may be parallelized have been shown. The major bottleneck involves developing effective and automated methods of parallelizing knowledge bases.

The cafeteria knowledge base example shows that good speed-up is possible from just rule-level parallelism. In fact, in the cases where both rule and fact partitioning can be done the speed-up is super-linear in this example. It appears the approach of rule-level parallelism holds significant promise for parallel expert system implementation on MIMD distributed memory computers. The Parallel Combat Information Fusion Testbed represents a challenging real-world application of Parallel CLIPS technologies.

REFERENCES

1. Newell, A., Gupta, A., and Forgy, C. “High-Speed Implementations of Rule-Based Systems,” *ACM Transactions On Computer Systems*, Vol. 7(2), 1989, pp. 119-146.

2. Boman, L. and Roose, D. “Communication Benchmarks for the ipsc/2,” *Proceedings of the First European Workshop on Hypercube and Distributed Computers*, Vol. 1, 1989, pp. 93-99.

3. Gupta, A. *Parallelism in Production Systems*, Morgan-Kaufmann Publishers, Inc., Los Altos, CA, 1987.

4. Hall, L.O. and Tello, I. “Parallel Clips and the Potential of Rule-Level Parallelism,” ISL-94-71, Department of CSE, University of South Florida, Tampa, FL, 1994.

5. Kuo, S and Moldovan, D. “Implementation of Multiple Rule Firing Production Systems On Hypercube,” AAAI, Vol. 1, 1991, pp. 304-309.

6. Miranker, D. P., Kuo, C., and Browne, J.C. “Parallelizing Transformations for a Concurrent Rule Execution Language,” TR-89-30, University Of Texas, Austin, 1989.

7. Neiman, D.E. “Parallel ops5 User’s Manual and Technical Report,” Department of Computer Science and Information Science, University of Massachusetts, 1991.

8. Nugent, S.F. “The Ipsc/2 Direct-Connect Communications Technology,” *Communications of the ACM*, Vol. 1, 1988, pp. 51-60.

9. Oflazer, K., “Partitioning in Parallel Processing of Production Systems,” *Proceedings of the 1984 Conference on Parallel Processing*, 1984, pp. 92-100.

10. Prasad, L. “Parallelization of Expert Systems in the Forward Chaining Mode in the Intel Hypercube,” MS Thesis, Department of CSE, University of South Florida, Tampa, FL, 1992.

11. Schmolze, J.G. “Guaranteeing Serializable Results in Synchronous Parallel Production Systems,” *Journal of Parallel and Distributed Computing*, Vol. 13(4), 1991, pp. 348-365.

12. Schmolze, J.G. and Goel, S. “A Parallel Asynchronous Distributed Production System,” *Proceedings of AAAI-90*, 1990, pp. 65-71.

13. Tello, I. “Automatic Partitioning of Expert Systems for Parallel Execution on an Intel Hypercube,” MS Thesis, Department of CSE, University of South Florida, Tampa, FL, 1994.

14. Bennett, B. H. “The Parallel Combat Information Fusion Testbed,” Honeywell High Performance Computing Workshop, Clearwater, FL, December 1992.

15. Bennett, B.H. “A Problem-solving Approach to Localization,” PhD Thesis, University of Minnesota, February 1992.

USING CLIPS TO REPRESENT KNOWLEDGE IN A VR SIMULATION

Mark Engelberg
LinCom Corporation
mle@gothamcity.jsc.nasa.gov

ABSTRACT

Virtual reality (VR) is an exciting use of advanced hardware and software techonologies to achieve an immersive simulation. Until recently, the majority of virtual environments were merely “fly-throughs” in which a user could freely explore a 3-dimensional world or a visualized dataset. Now that the underlying technologies are reaching a level of maturity, programmers are seeking ways to increase the complexity and interactivity of immersive simulations. In most cases, interactivity in a virtual environment can be specified in the form “whenever such-and-such happens to object X, it reacts in the following manner.” CLIPS and COOL provide a simple and elegant framework for representing this knowledge-base in an efficient manner that can be extended incrementally. The complexity of a detailed simulation becomes more manageable when the control flow is governed by CLIPS’ rule-based inference engine as opposed to by traditional procedural mechanisms. Examples in this paper will illustrate an effective way to represent VR information in CLIPS, and to tie this knowledge base to the input and output C routines of a typical virtual environment.

BACKGROUND INFORMATION

Virtual Reality

A virtual experience, or more precisely, a sense of immersion in a computer simulation, can be achieved with the use of specialized input/output devices. The head-mounted display (HMD) is perhaps the interface that most characterizes virtual reality. Two small screens, mounted close to the user's eyes, block out the real world, and provide the user with a three-dimensional view of the computer model. Many HMDs are mounted in helmets which also contain stereo headphones, so as to create the illusion of aural, as well as visual immersion in the virtual environment. Tracking technologies permit the computer to read the position and angle of the user’s head, and the scene is recalculated accordingly (ideally at a rate of thirty times a second or faster). [1]

There are many types of hardware devices which allow a user to interact with a virtual environment. At a minimum, the user must be able to navigate through the envrionment. The ability to perform actions or select objects in the environment is also critical to making a virtual environment truly interactive. One popular input device is the DataGlove which enables the user to specify actions and objects through gestures. Joysticks and several variants are also popular navigational devices.

Training

Virtual reality promises to have a tremendous impact on the way that training is done, particularly in areas where hands-on training is costly or dangerous. Training for surgery, space missions, and combat all fall into this category; these fields have already benefitted from existing simulation technologies [2]. As Joseph Psotka explains, “Virtual reality offers training as experience” [3, p. 96].

Current Obstacles

VR hardware is progressing at an astonishing rate. The price of HMDs and graphics workstations continues to fall as the capabilities of the equipment increase. Recent surveys of the literature have concluded that the biggest obstacle right now in creating complex virtual environments is the software tools. One study said that creating virtual environments was "much too hard, and it took too much handcrafting and special-casing due to low-level tools" [5, p. 6].

VR developers have suffered from a lack of focus on providing interactivity and intelligence in virtual environments. Researchers have been "most concerned with hardware, device drivers and low-level support libraries, and human factors and perception" [5, p. 6]. As a result,

> Additional research is needed to blend multimodal display, multisensory output, multimodal data input, the ability to abstract and expound (intelligent agent), and the ability to incorporate human intelligence to improve simulations of artifical environments. Also lacking are the theoretical and engineering methodologies generally related to software engineering and software engineering environments for computer-aided virtual world design. [4, p. 10]

CLIPS

VR programmers need a high-level interaction language that is object-oriented because "virtual environments have potentially many independent but interacting objects with complex behavior that must be simulated" [5, p. 6]. But unlike typical object-oriented systems, there must be "objects that have time-varying behavior, such as being able to execute Newtonian physics or systems of rules" [5, p. 7].

CLIPS 6.0 can fill this need for a high-level tool to program the interactions of a virtual environment. COOL provides the object-oriented approach to managing the large number of independent objects in complex virtual environments, and CLIPS 6.0 provides the ability to construct rules which rely on pattern-matching on these objects.

CLIPS is a particularly attractive option for VR training applications because many knowledge-bases for training are already implemented using CLIPS. If the knowledge-base about how different objects act and interact in a virtual environment is implemented in the same language as the knowledge-base containing expert knowledge about how to solve tasks within the environment, then needless programming effort can be saved.

LINKING CLIPS WITH A LOW-LEVEL VR LIBRARY

Data Structures

Every VR library must use some sort of data structure in order to store all relevant positional and rotational information of each object in a virtual environment. These structures are often called nodes and nodes are often linked hierarchically in a tree structure known as a scene. In order to write CLIPS rules about virtual objects, it is necessary to load all the scene information into COOL.

The following NODE class has slots for a parent node,b children nodes, x-, y-, and z-coordinates, and rotational angles about the x-, y-, and z-axis.

```
(defclass NODE
   (is-a USER)
   (role concrete)

   (slot node-index (visibility public)
      (create-accessor read-write) (type INTEGER))

   (slot parent (visibility public)
      (create-accessor read-write) (type INSTANCE-NAME))
   (multislot children (visibility public)
      (create-accessor read-write) (type INSTANCE-NAME))

   (slot x (visibility public) (create-accessor read) (type FLOAT))
   (slot y (visibility public) (create-accessor read) (type FLOAT))
   (slot z (visibility public) (create-accessor read) (type FLOAT))
   (slot xrot (visibility public) (create-accessor read) (type FLOAT))
   (slot yrot (visibility public) (create-accessor read) (type FLOAT))
   (slot zrot (visibility public) (create-accessor read) (type FLOAT)))
```

It is a trivial matter to write a converter which generates NODE instances from a scene file. For example, a typical scene might convert to:

```
(definstances tree
   (BODY of NODE)
   (HEAD of NODE))
   ; and so on

(modify-instance [BODY]
   (x 800.000000)
   (y -50.000000)
   (z 280.000000)
   (xrot 180.000000)
   (yrot 0.000000)
   (zrot 180.000000)
   (parent [REF])
   (children [HEAD] [Rpalm] [Chair]))
   ; and so on
```

Linking the Databases

Note that the NODE class has no default write accessors associated with slots x, y, z, xrot, yrot, or zrot. Throughout the VR simulation, the NODE instances must always reflect the values of the node structures stored in the VR library, and vice versa. To do this, the default write accessors are replaced by accessors which call a user-defined function to write the value to the corresponding VR data structure immediately after writing the value to the slot. Similarly, all of the VR functions must be slightly modified so that any change to a scene node is automatically transmitted to the slots in its corresponding NODE instance.

The node-index slot of the NODE class is used to give each instance a unique identifying integer which serves as an index into the C array of VR node structures. This helps establish the one-to-one correspondence between the two databases.

Motion Interaction

A one-to-one correspondence between the database used internally by the VR library and COOL objects permits many types of interactions to be easily programmed from within CLIPS,

particularly those dealing with motion. The following example illustrates how CLIPS can pattern-match and change slots, thus affecting the VR simulation.

```
(defrule rabbit-scared-of-blue-carrot
   (object (name [RABBIT]) (x ?r))
   (object (name [CARROT])
           (x ?c&:(< ?c (+ ?r 5))&:(> ?c (- ?r 5))))
   =>
   (send [RABBIT] put-x (- ?r 30)))
```

Assuming the rabbit and blue carrot can only move along the x-axis, this rule can be paraphrased as “whenever the blue carrot gets within 5 inches of the rabbit, the rabbit runs 30 inches away.”

The Simulation Loop

Most interactivity in virtual environments can be almost entirely specified by rules analogous the above example. A simulation then consists of the following cycle repeated over and over:

1. The low-level VR function which reads the input devices is invoked.

2. The new data is automatically passed to the corresponding nodes in COOL, as described in Section 2.1.

3. CLIPS is run, and any rules which were activated by the new data are executed.

4. The execution of these rules in turn triggers other rules in a domino-like effect until all relevant rules for this cycle have been fired.

5. The VR library renders the new scene from its internal database (which reflects all the new changes caused by CLIPS rules), and outputs this image to the user’s HMD.

While the low-level routines still provide the means for reading the input and generating the output of VR, the heart of the simulation loop is the execution of the CLIPS rule-base. This provides an elegant means for incrementally increasing the complexity of the simulation; best of all, CLIPS’ use of the Rete algorithm means that only relevant rules will be considered on each cycle of execution. This can be a tremendous advantage in complex simulations.

More VR functions

There are some VR interactions that cannot be accomplished solely through motion. Fortunately, CLIPS provides the ability to create user-defined functions. This allows the programmer to invoke external functions from within CLIPS. User-defined CLIPS functions can be provided for most of the useful functions in a VR function library so that the programmer can use these functions on the right-hand side of interaction rules. Some useful VR functions include:

make-invisible: Takes the node-index of an instance as an argument.

change-color: Requires the node-index of an instance and three floats specifying the red, green, and blue characteristics of the desired color.

test-collision: Takes two node-index integers and determines whether their corresponding objects are in contact in the simulation.

play-sound: Takes an integer which corresponds to a soundfile (this correspondence is established in another index file of soundfiles).

The play-sound function takes an integer as an argument, instead of a string or symbol, because there is slightly more overhead in processing and looking up the structures associated with strings, and speed is crucial in a virtual reality application. Similarly, all user-defined functions should receive the integer value stored in an instance's node-index slot, instead of the instance-name, for speed purposes.

It is also possible to create a library of useful deffunctions which do many common calculations completely within CLIPS. For example, a distance function, which takes two instance-names and returns the distance between their corresponding objects, can be written as follows:

```
(deffunction distance (?n1 ?n2)
   (sqrt (+ (** (- (send ?n2 get-x) (send ?n1 get-x)) 2)
            (** (- (send ?n2 get-y) (send ?n1 get-y)) 2)
            (** (- (send ?n2 get-z) (send ?n1 get-z)) 2))))
```

LAYERS UPON LAYERS

Once basic VR functionality is added to CLIPS, virtual environments can be organized in CLIPS according to familiar object-oriented and rule-based design principles. For example, a RABBIT class could be defined, and the example rules could be modified to pattern match on the is-a field instead of the name field. If this were done, any RABBIT instance would automatically derive the appropriate behavior. Once a library of classes is developed and an appropriate knowledge-base to go with it, creating a sophisticated virtual environment is merely a matter of instantiating these classes accordingly.

Consider this final example, illustrating points from this section and the *More VR Functions* Section.

```
(defrule shy-rabbit-behavior
   ?rabbit <- (object (is-a RABBIT) (personality shy)
                      (x ?) (y ?) (z ?))
   ?hand <- (object (is-a HAND) (x ?) (y ?) (z ?))
   =>
   (if (< (distance ?rabbit ?hand) 100) then
      (bind ?rabbit-index (send ?rabbit get-node-index))
      (if (evenp (random)) then
         (change-color ?rabbit-index 1 0 0)  ; rabbit blushes
         else
         (make-invisible ?rabbit-index))))
```

This rule becomes relevant only if there is a shy rabbit and a hand in the simulation. If so, shy-rabbit-behavior is activated whenever the hand or the rabbit moves. If the hand gets close to the rabbit, there is a 50% chance that the rabbit will blush, and a 50% chance that the rabbit will completely vanish.

CONCLUSION

CLIPS has been successfully enabled with virtual reality programming capabilities, using the methodologies described in this paper. The two test users of this approach have found CLIPS to be a simpler, more natural paradigm for programming virtual reality interactions than the standard approach of managing and invoking VR functions directly in the C language. Hopefully, by using high-level languages like CLIPS in the future, more VR programmers will

be freed from their current constraints of worrying about low-level details, and get on with what really matters—creating complex, intelligent, interactive environments.

REFERENCES

1. Ben Delaney. State of the art VR technology circa 1994. New Media, 4(8):44--45, August 1994.

2. Ben Delaney. Virtual reality goes to work. New Media, 4(8):40--48, August 1994.

3. Joseph Psotka. Synthetic environments for training. In Second Annual Synthetic Environments Conference, pages 79–107. Technical Marketing Society of America, March 1994.

4. U.S. Army Training and Doctrine Command and U.S. Army Research Office. Executive Summary—Virtual Reality/Synthetic Environments in Army Training, October 1992.

5. Andries van Dam. VR as a forcing function: Software implications of a new paradigm. In IEEE 1993 Symposium on Research Frontiers in Virtual Reality, pages 5–8, Los Alamitos, California, October 1993. IEEE Computer Society Technical Committee on Computer Graphics, IEEE Computer Society Press.

REFLEXIVE REASONING FOR DISTRIBUTED REAL-TIME SYSTEMS

David Goldstein
Computer Science Department
North Carolina Agricultural & Technical State University
Greeensboro, NC 27411
USA
Voice: (910) 334-7245
Fax: (910)334-7244
goldstn@garfield.ncat.edu

ABSTRACT

This paper discusses the implementation and use of reflexive reasoning in real-time, distributed knowledge-based applications. Recently there has been a great deal of interest in agent-oriented systems. Implementing such systems implies a mechanism for sharing knowledge, goals and other state information among the agents. Our techniques facilitate an agent examining both state information about other agents and the parameters of the knowledge-based system shell implementing its reasoning algorithms. The shell implementing the reasoning is the Distributed Artificial Intelligence Toolkit, which is a derivative of CLIPS.

INTRODUCTION

There has been a great deal of recent interest in multi-agent systems largely due to the increasing cost-effectiveness of utilizing distributed systems; in just the national CLIPS conference six papers appear which seem to discuss developing multi-agent systems. Further, although we strenuously try to avoid incorporating domain-specific knowledge in systems, real-time applications have an obvious need to understand the relationships between their processing requirements, available system resources, deadlines which must be met, and their environment. Hence, our programming methodology has been that individual agents need not necessarily be cognizant of any system information, but rather can communicate their own informational requirements, can sense their state in the system, and modify the internal processing parameters of the system (e.g., for load balancing) as the application demands. We allow the agents to sense and affect their processing environment so that they can intelligently reason about and affect the execution of applications.

IMPLEMENTATION OF REFLEXIVE REASONING

We have previously documented the characteristics of our tool, the Distributed Artificial Intelligence Toolkit [1][2]. The tool provides extensions to CLIPS for fault-tolerant, distributed, real-time reasoning. Many of these characteristics are controllable by individual reasoning agents so that when insufficient processing time is available, for example, processor fault-tolerance may need to sacrificed. The control of such characteristics is provided numerous predicates. Correspondingly, the agents can sense the current settings of the environment through a state description of the inference engine contained in the fact base (and which can be pattern-matched).

Calls to such predicates, as well as numerous ‘C’ functions implemented to provide additional functionality, were used to implement the Agent Manipulation Language (AML). AML (Table 1) provides the functionality to manipulate, assign tasks to, and teach agents. The functions used to implement AML include those providing fault-tolerance, for transmitting facts, template and objects, and those mimicking user-interface functionality; the data assistants of our architecture (Figure 1) actually interface to the user, evoking functionality from the reasoning agents [1].

build_agent(*processor*)	Creates an agent
teach_agent(*agent name, set of rules as text*)	Sends a ruleset to agent for reasoning
agent_unlearn(*agent name, set of rules as text*)	Causes agent to remove ruleset from its processing
agent_learn(*filename*)	Have agent read a rulebase from *filename*
die(*agent name*)	Kill the agent
wait(*agent name*)	Have the agent suspend reasoning
continue(*agent name*)	Have the agent continue reasoning

Table 1. Agent Manipulation Language (AML)

The last big issue is how the environment is sensed and affected at a low level. These processes are accomplished by intercepting and interpreting the individual elements of facts and templates before they are actually asserted. This kind of functionality allows agents to be minimally required to affect other agents; agents can know and affect each other (on the same or other machines) as much or as little as they desire. Formally proprietary information, this is now being divulged because implementing the code for the parsing of such information has been deemed too difficult for students, even graduate students, to maintain.

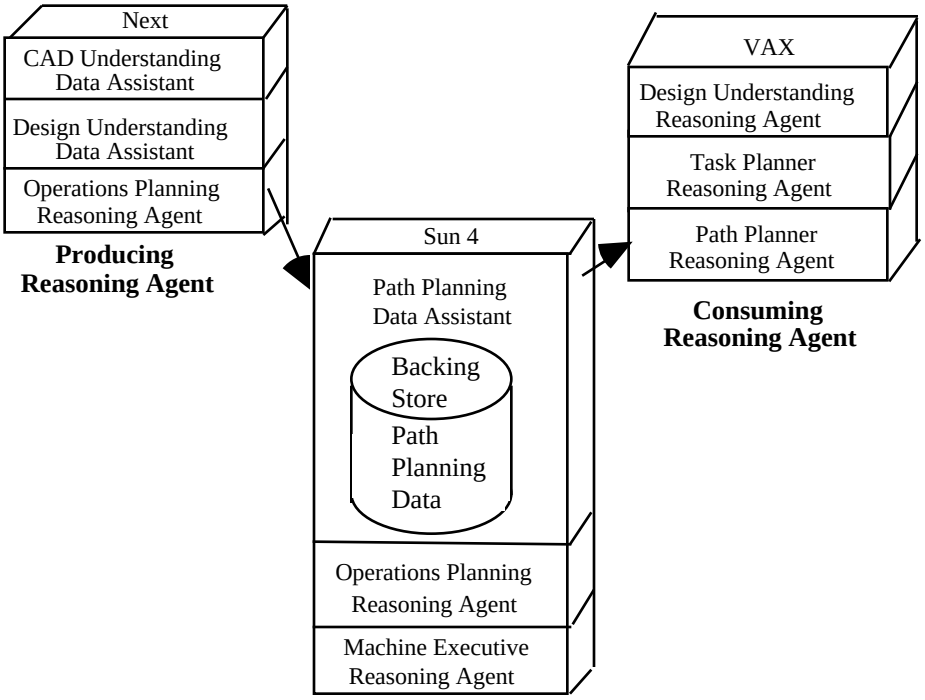


Figure 1. Architecture of the Distributed Artificial Intelligence Toolkit

REFLEXIVE REASONING IN DISTRIBUTED REAL-TIME SYSTEMS

Consider a real-time robotics application. The application consists of path planners, task planners, sensor and effector managers, motion control modules, etc. For the planning modules we typically would want to employ fault-tolerance, but for many of the other modules we would want very fast update rates. Hence, we might initially turn off fault-tolerance for, turn on interruptable reasoning for, and reduce the granularity of reasoning for the machines controllers, sensor managers, motion control modules, etc.

Certain planners and motion control modules would probably would probably require more resources than others. Modules noting short times between actual completion of tasks and the tasks' actual deadlines can evoke operating system functionality, via the data assistants, to determine processors with "excess" computing power. The over burdened agents could then create new agents, advise them to learn a set of rules, and off-load some of their work to newly instantiated agents. Hence, as the system executes, overworked agents could instigate load balancing.

Agents can also use reflexive reasoning in less subtle manners; any agent can request to see the fact and object base of any other agent. Agents can also request to know the goals of interest of other agents (or rather, which agents are interested in what goals). Hence, agents can also reason about the reason about the reasoning being performed by other agents.

CONCLUSION

We have briefly discussed implementing and using reflexive reasoning in distributed, real-time applications. Reflexive reasoning provides reasoning agents in distributed systems to analyze and modify the reasoning processes of other agents. We have found reflexive reasoning an effective tool for facilitating control of real-time, multi-agent systems. Our implementation of reflexive reasoning has been hierarchical, building up an agent manipulation language from predicates describing and affecting the reasoning process. These predicates have been, in turn, implemented from low-level functions written in 'C'.

REFERENCES

1. Goldstein, David, "The Distributed Artificial Intelligence Toolkit," *AI Expert*, Miller-Freeman Publishing, January, 1994, pp 30-34.

2. Goldstein, David, "Extensions to the Parallel Real-time Artificial Intelligence System (PRAIS) for Fault-tolerant Heterogeneous Cycle-stealing Reasoning", in Proceedings of the 2nd Annual CLIPS ('C' Language Integrated Production System) Conference, NASA Johnson, Houston, September 1991.

PALYMSYS™ - AN EXTENDED VERSION OF CLIPS FOR CONSTRUCTING AND REASONING WITH BLACKBOARDS

Travis Bryson and Dan Ballard
Reticular Systems, Inc.
4715 Viewridge Ave. #200
San Diego, CA 92123

ABSTRACT

This paper describes PalymSys™ -- an extended version of the CLIPS language that is designed to facilitate the implementation of blackboard systems. The paper first describes the general characteristics of blackboards and shows how a control blackboard architecture can be used by AI systems to examine their own behavior and adapt to real-time problem-solving situations by striking a balance between domain and control reasoning. The paper then describes the use of PalymSys™ in the development of a situation assessment subsystem for use aboard Army helicopters. This system performs real-time inferencing about the current battlefield situation using multiple domain blackboards as well as a control blackboard. A description of the control and domain blackboards and their implementation is presented. The paper also describes modifications made to the standard CLIPS 6.02 language in PalymSys™ 2.0. These include: 1) A dynamic Dempster-Shafer belief network whose structure is completely specifiable at run-time in the consequent of a PalymSys™ rule, 2) Extension of the *run* command including a continuous run feature that enables the system to run even when the agenda is empty, and 3) A built-in communications link that uses shared memory to communicate with other independent processes.

INTRODUCTION

This paper describes the extensions made to the CLIPS 6.02 language during the design and implementation of a Situation Assessment (SA) expert system for use aboard Army helicopters. An SA system uses data gathered from external environmental sensors, intelligence updates, and pre-mission intelligence to monitor and describe the external environment. An SA system searches for external entities of interest (EEOI), recognizes those EEOIs, and then infers high-level attributes about them. An EEOI is anything that has the potential for affecting the planned rotorcraft mission. EEOIs are primarily (although not necessarily) enemy forces. In order for the system to perform the inferences necessary to develop an assessment of the current situation, it must utilize extensive knowledge about the EEOIs including knowledge about their doctrine, capabilities, probable mission objectives, intentions, plans, and goals. All of these elements combine to form a complete situation description. For a thorough description of the domain problem see [1].

The SA system has been implemented in an extended version of CLIPS called PalymSys™. The SA system implementation makes use of two domain blackboards - current assessment and predicted assessment, as well as a control blackboard for overall control of the system. PalymSys™ provides a reasoning under uncertainty mechanism that handles contradictory and partially contradictory hypotheses and allows multiple hypotheses to coexist. A continuous run option has been added that allows the system to run even when the agenda is empty. Continuous run enables the system to wait for new environmental state data to be provided by the system's sensor subsystems. As new data becomes available, additional reasoning is then performed.

BLACKBOARDS

The SA system uses a blackboard architecture as a paradigm for solving the situation assessment problem. The blackboard architecture approach to problem solving has been a popular model for expert system design since the development of the Hearsay-II speech understanding program in the 1970s. It also serves as a framework for the blackboard control architecture - an extension of the blackboard architecture - which is the method of control used in the SA system. The blackboard model for problem solving consists of three primary elements [2, 3]:

Knowledge Sources: The knowledge necessary to solve the problem is partitioned into separate and independent knowledge sources. The independence of knowledge sources means that major modifications to the system should not be necessary when more rules are added to the system. In CLIPS and PalymSys™, knowledge sources take the form of rules.

Blackboard Data Structure: A global data structure where knowledge that has been brought to bear on the problem is stored. The blackboard represents the current state of the problem solution. The system attempts to combine and extend partial solutions that span a portion of the blackboard into a complete problem solution. Communication between knowledge sources takes place solely via the blackboard. In CLIPS, a blackboard data structure can be represented by objects that encapsulate the knowledge at each level. The knowledge is contributed by the consequent of rules whose antecedent has been satisfied.

Control: Each of the knowledge sources opportunistically contributes to the overall problem solution. Each knowledge source is responsible for knowing the conditions under which it will be able to contribute to the problem solution. In CLIPS, this means deciding which rule or set of rules should fire next given the current state of the blackboards. Our method for achieving this is the use of the control blackboard architecture. The control blackboard is an extension of the traditional blackboard architecture and will be discussed in detail later in this paper.

The SA system uses three concurrently executing blackboards for developing a problem solution. These are a *prediction* blackboard, an *assessment* blackboard, and a *control* blackboard. Each blackboard provides storage for the problem solution state data. The assessment blackboard contains the current situation description and is primarily concerned with the current intentions, capabilities, and commitments of EEOIs. The prediction blackboard contains predictions for EEOI behavior and the predicted situation description. The control blackboard contains the knowledge that manages and prioritizes all of the rules and provides for overall control of system problem-solving behavior.

DESIGNING THE SA ASSESSMENT BLACKBOARD

The blackboard model provides only a general model for problem solving. It falls far short of an engineering specification for actually developing a complete blackboard system in CLIPS. However, this general model does provide significant insight in how to implement complex knowledge-based systems. The first step in designing a blackboard for a given domain problem is to subdivide the problem into discrete subproblems. Each subproblem represents roughly an independent area of expertise. The subproblems are then organized into a hierarchy of levels from least to most abstract. Correctly identifying the problem hierarchically is crucial and will often be the primary factor that determines the effectiveness of the problem-solving system (or whether the problem can be solved at all). Blackboards sometimes have multiple blackboard

panels, each with their own set of levels. That is, the solution space can be segmented into semi-independent partitions.

The knowledge sources used by the system are CLIPS rules that have access to the information on the blackboard. Communication and interaction among rules is solely via the blackboard data structure. Even knowledge sources on the same level must share information through the blackboard. Encoded within each knowledge source are the conditions under which it can contribute to the problem solution.

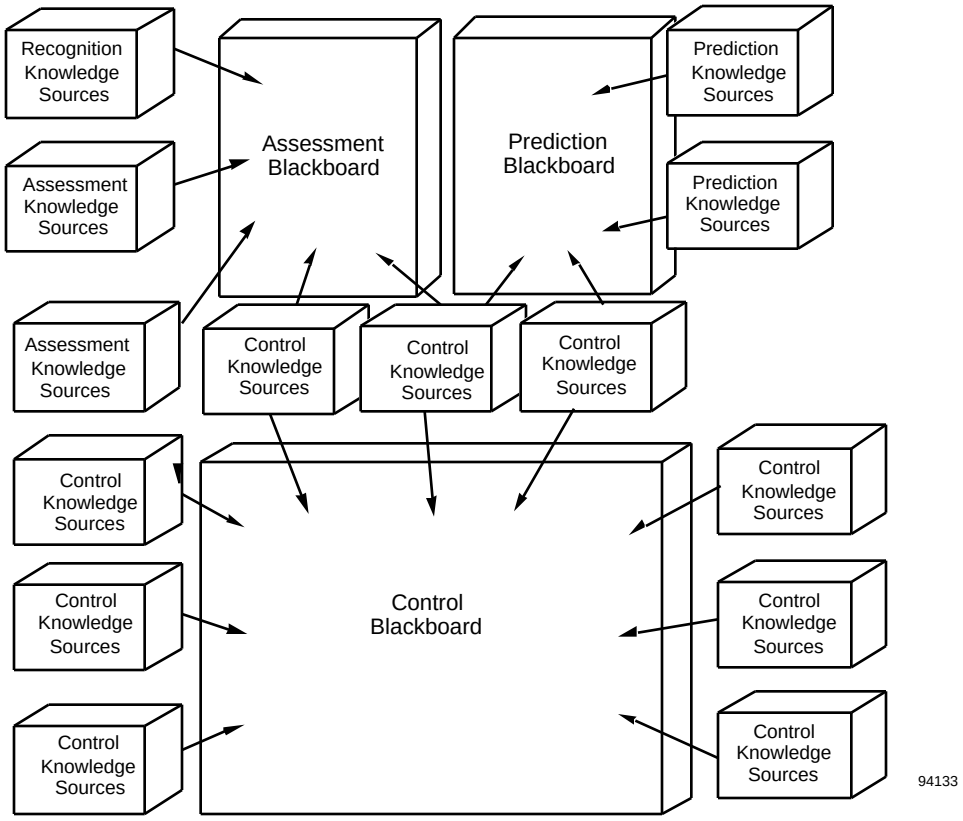


Figure 1.

Figure 2 is an illustration of the assessment blackboard in the SA system. The assessment blackboard is divided into a seven-tiered hierarchy. These levels are concerned with developing an environmental state description, characterizing an EEOI, interpreting EEOI plans, roles, and intents and developing a summary description of the overall situation. Each level of the assessment blackboard is a *part-of* hierarchy that represents a portion of the situation assessment solution for a particular EEOI. There is a gradual abstraction of the problem as higher levels on the blackboard are reached. Information (properties) of objects on one level serve as inputs to a set of rules which, in turn, place new information on the same or adjacent levels. During the problem-solving process, more advanced hypotheses and inferences are placed at higher levels of the blackboard.

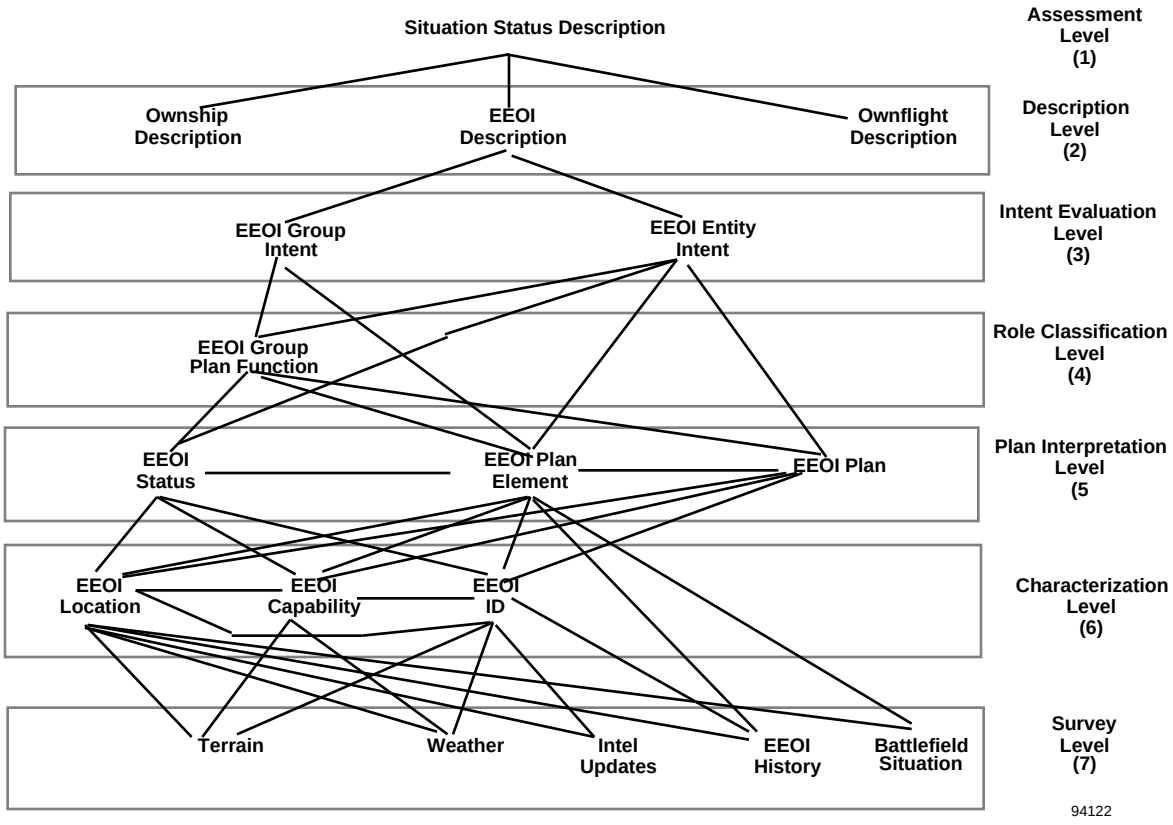


Figure 2.

The blackboard architecture provides a model for the overall problem-solving (inferencing) process. This model is used to structure the problem domain and identifies the knowledge sources needed to solve the problem. While knowledge sources are independent and each contributes to a partial solution, each knowledge source must be designed to fit into the high-level problem-solving blackboard hierarchy created by the system designer.

USING CLIPS OBJECTS AS THE BLACKBOARD

Information on the assessment blackboard is represented as CLIPS objects. Figure 3 shows the object representation for knowledge in one of the SA modules. This figure shows the structure in the global plan function (GPF) module at the Role Classification level of the blackboard hierarchy.

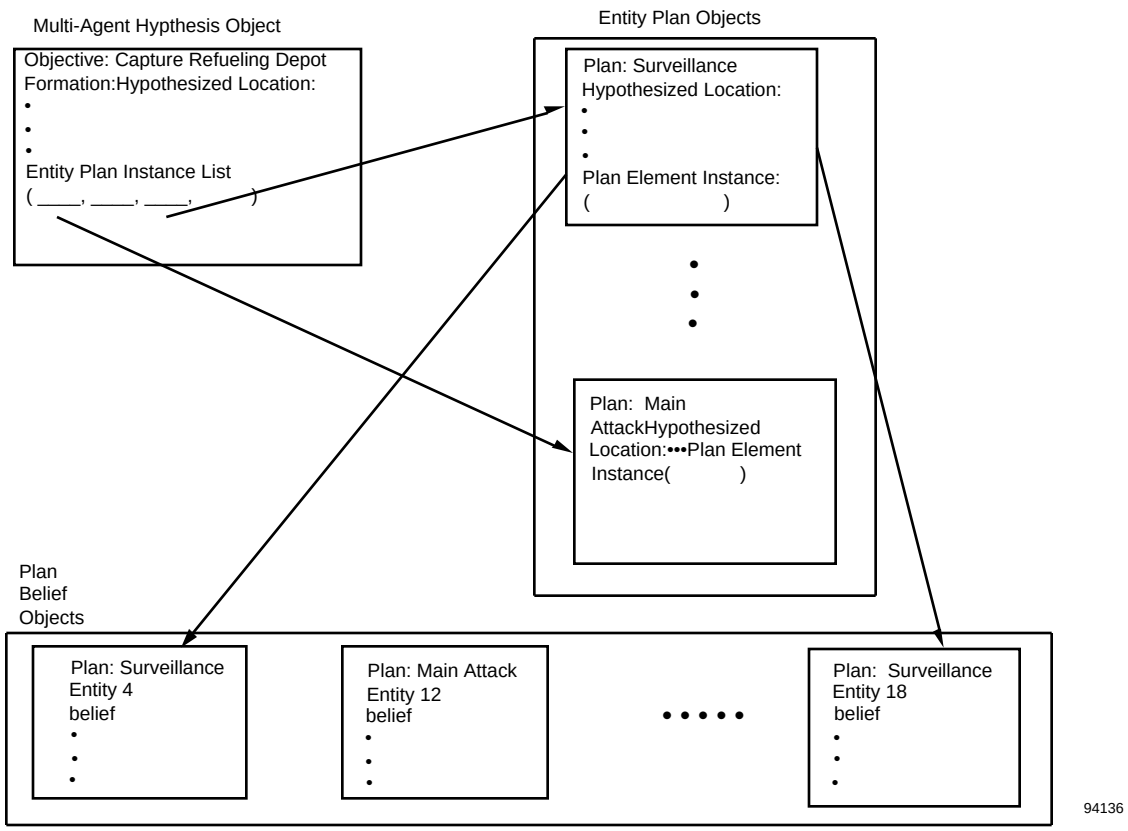


Figure 3.

A multi-agent plan hypothesis object is created by the system to represent the high-level multi-agent plan of a group of EEOIs where each has the same high-level mission objective. Multi-agent hypotheses and their associated entity plans are stored as objects on the blackboard. An entity plan object contains the sequence of plan elements (activities) that a particular EEOI must actually perform within the context of the associated multi-agent hypothesis. For instance, an EEOI's multi-agent plan might be to capture a refueling depot. A number of EEOIs will be needed to achieve this objective including a security force, a main attack force, and surveillance for the attacking force. The EEOI's entity plan might then be *surveillance* for the attacking force. The multi-agent hypothesis object called *capture refueling depot* encapsulates information local to the role classification level like formation information, hypothesized locations, and typical vehicle types. Other objects can access this information only through the multi-agent hypothesis object's defmessage handlers.

The *capture refueling depot* object has a multislot field that contains the list of entity plan instances necessary to carry out its objective. Objects at different blackboard levels which are permanently linked are connected via instance lists. The entity plan instances, in turn, contain the lists of plan elements necessary to carry out the entity-level plan. The plan element lists are encapsulated within the entity plan objects. When the need to do so arises, entity plan objects will search for plan belief objects that correspond to their plan via pattern matching. They search for plan belief objects instead of parsing a pre-defined list because the links in this case are not permanent. The plan belief objects are entity specific and store the degree of belief in which the system believes that a particular EEOI is performing a particular plan. An EEOI may change entity plans or the system may gather evidence that leads it to believe the EEOI is actually performing a different plan. Thus the links between entity plan objects and plan belief objects will change over time.

CLIPS Objects are an ideal data structure for blackboard implementation because they offer encapsulation and easy processing of lists. Recall that the essence of the blackboard approach is that higher levels synthesize the knowledge from the lower levels. The objects at one blackboard level will typically need access to a list of conclusions from the preceding levels. In our approach, objects are always looking down the blackboard, asking other objects for only the information they need. The knowledge at each blackboard level is well encapsulated. Knowledge sources can thus be formulated generically.

THE CONTROL BLACKBOARD ARCHITECTURE

At each point in the problem solving process there are typically a number of knowledge sources that can contribute to the problem solution. Every intelligent system must solve the control problem: i.e., determine which knowledge source should next be brought to bear on the problem solution. In order to solve the control problem it is necessary that control decision making be viewed as a separate problem-solving task. The system must plan problem-solving actions using strategies and heuristics that will help it solve the control problem while balancing efficiency and correctness. The system must become aware of how it solves problems and intelligently guide the problem-solving process.

Explicitly solving the control problem involves providing a body of meta-level (heuristic) knowledge about the domain that is used to guide the control planning process [4]. With such knowledge, the system can reason explicitly about control because the system has access to all of the knowledge that influences control decisions. Meta-level rules then choose domain rules or sets of domain rules that are most appropriate for the current problem-solving situation.

Control knowledge sources interact solely via the control blackboard. The control blackboard is where control knowledge sources post all currently relevant meta-level system knowledge. Partial and complete control plans are stored here. The system also posts high-level control heuristics and problem-solving strategies on the control blackboard.

A well designed control mechanism can make sophisticated meta-level decisions about the problem-solving process. It will seek to make desirable actions more feasible and feasible actions more desirable. It will make plans to seek out important obtainable information when that information is missing. The control mechanism must carefully balance the time spent solving control problems with time spent carrying out domain tasks. It must be aware of how it is solving domain problems and change problem-solving methods to match the situation requirements.

THE CONTROL PROBLEM IN SA

The SA system is a real-time system that must perform complex inferences within very demanding time constraints. Control is critical in a real-time system because by definition problems must be solved before a deadline. The SA system has a set of meta-level control rules that interact via the control blackboard. The control rule set uses heuristics to evaluate situation characteristics which are used to choose one or more problems from a pool of several competing domain problems. Once the important domain problems are chosen, an efficient control plan is constructed to solve them. A control plan consists of a series of rule groups, or modules, which will be sequentially accessed by the system. A message to work with a specific entity is frequently sent along with the control plan.

Control planning is a way of incorporating efficiency into the system. The meta-level priority criteria do not have to be recalculated every cycle while the system is following an established control plan. The system balances the degree of commitment to the execution of control plans with a variable sensitivity to run-time conditions [5]. The degree of commitment is a function of

the uncertainty about the helicopter's actual environment. If the situation is dangerous, the system will lower its commitment to the control plan and heighten its sensitivity to run-time conditions (i.e., incoming sensor data).

Figure 4 is a diagram of the control blackboard used in SA. As in the assessment and prediction blackboards, a multilevel hierarchical blackboard structure is used. The *policy* level is where global focusing decisions are stored. These are heuristically generated at run-time depending on the state of the problem-solving situation. The *problem* level is where domain problems presented to the system are placed. The *strategy* level is where strategies (high-level general descriptions of sequences of actions) are stored. Generating a strategy is the first step in making a plan. Strategy decisions are posted at the strategy level for all accepted problems. A strategy decision is nothing more than a constraint on future actions. The *focus* level is equivalent to the strategy level but is populated by more specific strategies called foci. The *action* level represents the actual sequence of actions chosen by the system to solve the problem [6]. The action level is implemented by the CLIPS agenda mechanism.

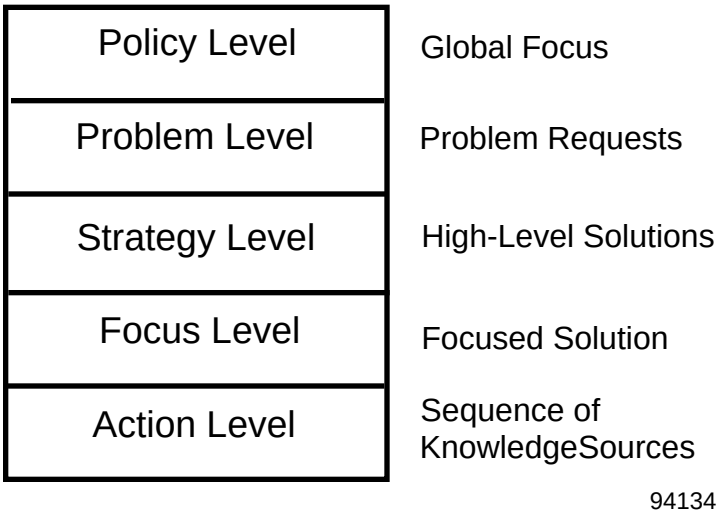


Figure 4.

META CONTROL PLANS

The knowledge represented on each of the control blackboard levels increases in abstraction from the bottom level to the top level. However, control is a top-down inferencing process. Unlike the domain blackboards, knowledge at the higher blackboard levels serves as input for the knowledge sources at the lower control blackboard levels. A meta control plan solves the control problem for the control part of the SA system. A meta control plan object is constructed at system start-up that contains the following sequence of phases:

- check for new data
- post foci at policy level
- remove problems (if appropriate)
- request problems
- accept problems
- prioritize problems
- choose problems
- formulate strategy
- formulate plan

- execute a control plan
- record any plan completions and perform system maintenance

The control rules are partitioned by phase. There is a rule set for accepting problems, prioritizing problems, etc. The early phases deal with the higher levels on the control blackboard. The system will cycle through the meta control plan sequentially unless a message is passed from one module to another.

One situation in which message passing occurs is when the system suspects that a new domain problem should be reevaluated in light of the particular domain problem that has been chosen. For instance, when the system is making a plan to do a situation assessment, it will reconsider data that has not yet been integrated into the system. The data may not be intrinsically important. But in the *context* of doing a situation assessment, the system may decide to integrate part of the unprocessed data before doing the situation assessment. Integrating the data first often adds value to the situation assessment because the system will have more information on which to base its assessment. This added value is added to the priority of the data integration plan request. The system goes back to the *prioritize problem* phase and if the data adds enough value to the situation assessment problem, it constructs an object to solve the integrate data problem. The data integration problem instance is added to the list of plan elements in the situation assessment plan object. The actual plan elements to integrate the data into the system are encapsulated within the data integration control plan object.

HEURISTICS USED BY THE CONTROL PLANNER

A real-time SA system is continuously supplied with sensor updates, pilot requests for information, anticipation of possible future events, and a plethora of cognitive actions that must be taken in order to assess the current situation. Thus, most of the work of the control part of the system is in deciding what problem to solve. Following are the five domain problems that the SA system solves:

1) Integrate new data into the domain blackboards
2) Focus sensors
3) Generate a current situation assessment
4) Generate a predicted assessment for some opportune time in the future
5) Generate a predicted assessment for time T seconds from the present

In order to solve the control problem, the SA system opportunistically chooses problems from among these five domain problems and makes efficient plans to solve them. This approach provides a built-in well-defined external interface to the system. Problems presented to the SA system by an external agent (e.g., an external system planner or the pilot) are placed in with the problems the SA system has presented to itself at the problem level of the control blackboard.

In order to illustrate the meta-level heuristics that the SA system uses to choose from among competing domain problems, we provide an example of how the SA system integrates new data into the domain blackboard (problem 1 above). The SA system places incoming sensor and intelligence data at the survey level of the assessment blackboard. This new data triggers a problem request at the problem level of the control blackboard. When the SA system decides to integrate the new data, the control rules make and carry out a plan that consists of an ordered sequence of domain modules which will be sequentially examined by the SA system.

All incoming information is rated for importance. EEOIs that have already been encountered by the system are given *Entity Assessment Ratings* (EARs):

EAR = [**Confirmed**(danger), **Plausible**(danger)]

The EAR is a belief function that represents the degree to which an EEOI is a threat to the rotorcraft. The *confirmed* danger is the degree to which the system has confirmed that an EEOI is a danger to ownship. The *plausible* danger (or potential danger) is the worst-case danger that an EEOI presents to ownship at this time. Both of these numbers must lie in the range [0, 1]. The plausible danger is always greater than or equal to the confirmed danger. The less the system knows about an EEOI, the greater the difference between the plausible and the confirmed danger.

The EAR is synthesized into an *Interesting Rating*:

Interesting Rating = 0.7 * Confirmed danger +
1.3 * ability_garner() * (Plausible danger - Confirmed danger)

The system evaluates EEOIs as “more interesting” if there is a large gap between the plausible and confirmed dangers. This means EEOIs that might be dangerous but about which there is little knowledge will be rated or ranked more interesting. Ability_garner is a function that calculates the degree to which the system thinks it currently has the ability to gather more information about the EEOI. When new data come in about an EEOI, we can use that entities' previously calculated Interesting Ratings to prioritize it.

When data about a previously unencountered entity arrives, SA the system favors the integration of the information into the domain blackboard if the data is about a nearby EEOI. The SA system especially favors it if automatic target recognition (ATR) has managed to already confirm an EEOI's vehicle type. Data about new EEOIs is considered intrinsically more important than data about previously encountered entities. The SA system attempts to reject duplicate information before any attempt is made to rate it or integrate it. The control planner always attempts to control the sensors to gain more information about interesting EEOIs.

The amount of time spent generating and evaluating heuristics must be balanced with the amount of time spent executing domain rules. It is possible to expend too many computational resources prioritizing problems and not enough time actively solving them. Entity Assessment Ratings and Interesting Ratings require processing resources for calculation. However, they must be calculated anyway for use by other parts of the system and these calculations are entirely procedural or algorithmic and are therefore computationally relatively inexpensive. Very little extra processing power is required to rate entities in this way. Such overlapping requirements often enable more sophisticated meta-level control knowledge to be produced. The results of the inferencing process represented at various blackboard levels by symbolic abstractions can thus be used as input for procedural/algorithmic computation that, in turn, produces useful metal-level control knowledge.

USING DYNAMIC SALIENCE FOR CONTROL

The planning approach to control has the disadvantage of always firing each of the rules that pertain to the chosen domain problem within the modules listed in the control plan. Another layer of control can be attained by directing the system to fire only the subset of eligible domain rules that best apply to the current domain problem. This flexibility is achieved within PalymSys™ by using an expanded form of the salience command.

The modifications to the salience command in PalymSys™ are based on the work done by Robert Orchard of the National Research Council of Canada in his extended version of CLIPS called BB_CLIPS [7]. The added syntax, called rule *features*, are descriptive characteristics of the knowledge contained in a rule that are placed within the antecedent of the rule.

```
;;
;; RULE: pred_pe2
;;
;; If the EEOI will be able to see ownship and will be able to hit
;; ownship and the EEOI's plan is combat_recon, main_attack,
;; close_air_support, artillery, or guard then EE will probably be engaging
;; you in the future (60%).  If not, then we can't be sure what the EEOI
;; will do next (40%).
;;

(defrule pred_plan_element12
   (declare                      ; feature list
      (salience 200)             ; salience type
      (reliability 35)           ; integer type
      (importance 25)            ; integer type
      (efficiency medium))       ; set type
   (Module_focus (focus domain)
      (sub_focus pred_plan_element) (entity_focus all)
      (time_focus ?time&:(>= ?time3))
      (level policy)(BB CONTROL))
   (object (is-a EEOI_Pred_location_long)
      (label ?name)
      (dist_from_ownship ?dist&:(< ?dist 6)) ;; all EEs < 6km away.
      (level interpretation) (BB PREDICTION))
   (object (is-a EEOI_Pred_capability_long)
      (label ?name) (see_capability ?seecap&:(>= ?seecap .5))
      (hit_capability ?hc&:(> ?hc .5))
      (level pred_cap) (BB ASSESSMENT))
   (object (is-a EEOI_Plan)
      (label ?name) (propagation ?prop)
      (type ?type&:(member$ ?type
                            (create$ combat_recon long_range_recon
                                     guard_forward guard_flank guard_rear
                                     main_attack close_air_support)))
      (level plan_interp) (BB ASSESSMENT))
   =>
   (assert_belief ?name pred_plan_element ?prop
      ".6 ENGAGE_OWNSHIP .4 ALL"))
```

Each feature has an associated dynamic salience value determined by its feature arguments. PalymSys™ has a combining function that evaluates the salience of each rule between rule firings. The feature argument itself is a pointer to a dynamic data structure of salience values that is modified by control rules at run-time. For instance, if the system is suddenly faced with a time constraint, a control decision can be made to globally raise the value of the efficiency feature.

A new feature in CLIPS 6.0 is the (set-salience-evaluation every-cycle) command which enables salience values to be calculated dynamically at run-time between rule firings. It is possible to achieve the same functionality described above from within the CLIPS 6.0 shell by placing a function as the argument for the salience command. Between rule firings, the function dynamically computes salience values which are based on global control variables whose values have been determined by control decisions.

A HYBRID PALYMSYS™/C++ BELIEF NETWORK

Reasoning under uncertainty is a necessity for a situation assessment system. The SA system must make prescient inferences about such things as an EEOI's plan or the associated elements (steps) in that plan. The EEOI's plans and plan elements cannot be known for certain until

various activities are explicitly observed. Other inferences, such as an EEOI's intent or an EEOI's *predicted* plan element, can never be known for certain. Hypotheses must be based on incomplete and unreliable evidence because the battlefield is a complex, uncertain environment. Reasoning under uncertainty requires a probabilistic model of reasoning that supports reasoning using contradictory and partially contradictory hypotheses in which the system has varying degrees of confidence.

PalymSys™ enables the user to construct a Dempster-Shafer (D-S) belief network quite easily. A command line interface allows the user to specify size, structure and number of instances of the network at run time. A belief network propagates the uncertainty associated with a particular piece of knowledge throughout the entire hierarchy of hypotheses that depend upon it. The SA control planner in conjunction with the Rete Pattern Matching algorithm handles the belief propagation through the system hierarchy. When rules are added to the system, no modifications of existing C++ or PalymSys™ code are necessary. By placing a single function call on the consequent of the added rule(s), the system will incorporate the new rule(s) into the belief network automatically. A formal explanation of Dempster-Shafer theory is beyond the scope of this paper. Such detailed presentations can be found in [8, 9, 10]. However, the CLIPS modifications described here can be applied to a monotonic, feed-forward belief network of any type (i.e., Bayesian).

A frame of discernment is a set of mutually exclusive hypotheses. Exactly one hypothesis in a frame of discernment is true at any one time. Each module that uses D-S reasoning in SA has its own frame of discernment. For example, the frame of discernment corresponding to the Plan Element module is the set of fourteen distinct plan elements that an entity is capable of performing within the context of all possible plans. When an entity is encountered, it is assumed to be performing one and exactly one of these plan elements. The purpose of the plan element module is to assign belief values to each of the members of the plan element frame of discernment. A set of propagation values for the plan element frame of discernment is also calculated. The propagation values serve as input to other frames of discernment that use plan elements as evidence.

Recall that the SA system follows entity specific control plans in order to integrate new data into the system. A control plan is an ordered list of modules that the system will sequentially visit to solve a domain problem. The exact order of the control plan will vary depending on what type of data is being integrated into the system. When a control plan element is executed, a particular module fires its rules, assigns belief to the members of its frame of discernment, and then control proceeds to the next module in the control plan which is typically on the next higher level within the domain blackboard hierarchy.

A typical SA domain rule within the belief network will look much like the following rule from the Plan module:

```
RULE: plan_rule12
Description:
;; If the EE's current Plan Element is Reporting then his Plan might be
;; (40%) surveillance, combat recon, or long range recon.  It also might
;; be, to a slightly lesser degree (30%), guard forward, guard flank, or
;; guard rear.  If it's not in those two sets, then the EE could be
;; performing any Plan (30%).

(defrule plan_rule12
   (Module_focus (focus domain)
                 (sub_focus plan) (entity_focus ?name)
                 (level policy)(BB CONTROL))
   (object (is-a EEOI_Plan_element) (label ?name) (type report)
```

```
(propagation ?prop_value&:(> ?prop 0))
   (level plan_interp)(BB ASSESSMENT))
=>
(assert_belief ?name Plan_Module ?prop_value
   ".4 SUR CRP LRRP .3 GFR GFL GR .3 ALL"))
```

The variable *name* is needed as a tag because SA makes an instance of the belief network for each new EEOI encountered. In this case, the assert_belief function places the belief into the frame of discernment associated with the Plan module with a propagation value of *prop_value*. The Module_focus template values in the rule will allow this rule to fire only when the system is firing the rules on the assessment blackboard in the plan module. In this way, the control plan assures that the rules in each frame of discernment for a particular EEOI are finished firing before advancing up to higher levels on the blackboard. The same technique can be used with any monotonic feed-forward blackboard belief propagation scheme.

When evidence is obtained from another frame of discernment, as is the case with the EEOI_plan_element object in the rule above, the propagation value is also sent to the assert_belief function. Evidence without a propagation value associated with it is assumed to have a probability of truth of one. However, the system can easily adapt to any uncertain data by attaching a propagation value to them.

The last rule fired within each module calls the function get_belief to access the appropriate belief and propagation values for the module's frame of discernment. The propagation values will be passed to the next level up in the blackboard hierarchy via the propagation slot in the object that corresponds to the current entity and the current blackboard level.

New domain rules are easily added to the system by placing the assert_belief function on their RHS. No modifications to the reasoning under uncertainty function code are required since propagation is handled entirely by the Rete Pattern Matching algorithm and the SA control planner.

SIMULATION AND TEST ENVIRONMENT INTERFACE

SA uses interprocess communication to communicate with our rotorcraft mission simulation and test environment (STE). The STE is a graphical simulation test bed written in C++ and implemented on a RS/6000 workstation. During simulation runs, the STE sends SA sensor information and SA sends the STE directional parameters to control the sensors. A communications class was written for the STE and integrated with PalymSys™. Shared memory in our system is accessed via the standard system C libraries<*sys/shm.h*> and <*sys/types.h*>. More specifically, the STE uses the function calls *shmat, shmget, shmdt* to attach a process, grab the shared memory and detach the process, respectively.

Standard CLIPS terminates when the agenda is empty. PalymSys™ can be directed to run continuously even though the agenda is empty by adding an optional argument to the run command. The inference engine will idle, waiting for facts to be asserted into the system. This capability is essential whenever the system depends on an independent process, like the STE, as a source for fact assertions.

Three functions were embedded into PalymSys™ in order to communicate with the STE. One function checks the communications link to see if information had been passed over from the STE. The maximum buffer size was 200 characters, so the PalymSys™ function got the name of a file from shared memory that had just been created by the STE. Another PalymSys™ function reads the file just created by the STE and asserts the contents as facts into the PalymSys™ fact base using the *AssertString* CLIPS C library call. Finally, another function lets the STE know via

shared memory when SA has sent it information via file transfer. This two-way real-time interprocess communication provides a realistic simulation of a rotorcraft environment.

SUMMARY AND CONCLUSIONS

We have implemented a situation assessment blackboard expert system in PalymSys™ -- an extended version of CLIPS. Blackboards are an excellent paradigm for CLIPS expert system implementations. The control blackboard architecture is especially well-suited to real-time applications like SA. We developed a control planner in PalymSys™ that chooses the most important problems to solve based on complex meta-level situation characteristics. The control planner creates domain plans to solve the problems that it chooses. SA uses a monotonic feed-forward Dempster-Shafer belief network implemented in C++. The size and number of instances of the network is dynamic and completely controlled at run-time from the PalymSys™ shell. Finally, we interfaced the SA system to our Simulation and Test Environment using interprocess communication techniques. A continuous run feature was added which enables the inference engine to idle even when the agenda is empty.

ACKNOWLEDGMENTS

Mr. Joe Marcelino was instrumental in the implementation of the assessment and prediction blackboards. Mr. Richard Warren implemented the terrain reasoning system. Mr. Jerry Clark provided invaluable advice on the reasoning under uncertainty mechanism used in SA. Mr. Clark served as the rotorcraft domain expert on this project. Mr. Steve Schnetzler implemented the interprocess communications between SA and the Simulation and Test Environment.

BIBLIOGRAPHY

[1] D. Ballard and L. Rippy, “A knowledge-based decision aid for enhanced situational awareness,” in *Proceedings of Thirteenth Annual Digital Avionics Systems Conference*, Phoenix, AZ, 1994, in press.

[2] H. P. Nii, “Blackboard Systems - Part I,” *AI Magazine*, vol. 7, no. pp. 38 - 53, 1986.

[3] H. P. Nii, “Blackboard Systems - Part II,” *AI Magazine*, vol. 7, no. 4, pp. 82 - 107, 1986.

[4] N. Carver and V. Lesser, “A planner for the control of problem-solving systems,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1519 - 1536, 1993.

[5] B. Hayes-Roth, “Opportunistic control of action in intelligent agents,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1575 - 1587, 1993.

[6] B. Hayes-Roth, “A blackboard architecture for control,” *Artificial Intelligence*, vol. 26, pp. 251 - 321, 1985.

[7] R. Orchard and A. Diaz, “BB_CLIPS: Blackboard extensions to CLIPS,” in *Proceedings of First CLIPS Conference*, Houston, Texas, 1990, pp. 581 - 591.

[8] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[9] G. Shafer, *A Mathematical Theory of Evidence*. Princeton, NJ: Princeton University Press, 1976.

[10] G. Shafer and J. Pearl, *Uncertain Reasoning*. San Mateo, CA: Morgan Kaufmann, 1990.

A GENERIC ON-LINE DIAGNOSTIC SYSTEM (GOLDS) TO INTEGRATE MULTIPLE DIAGNOSTIC TECHNIQUES

Pratibha Boloor, Ted Leibfried, Terry Feagin, Joseph Giarratano,
University of Houston Clear Lake

David Skapura
Inference Corp.

ABSTRACT

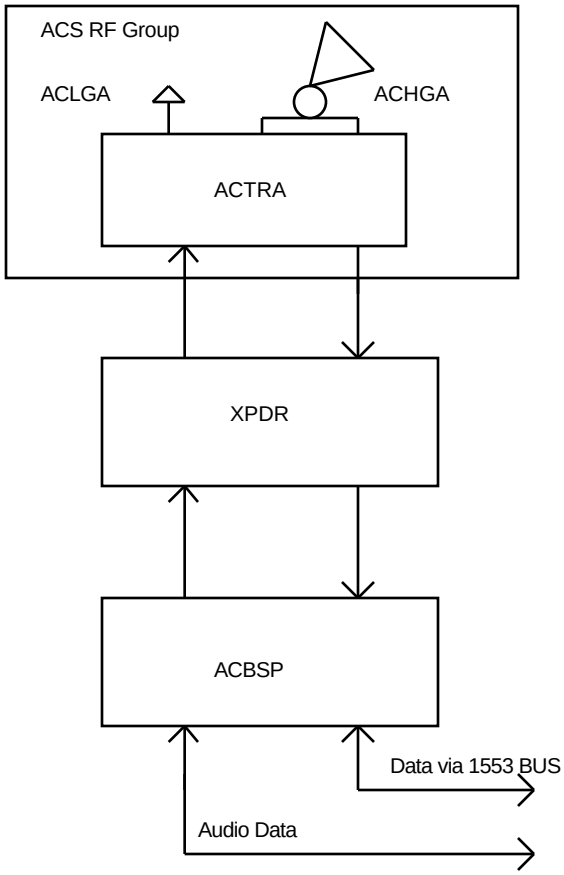
The goal of this research is to study the parallel application of multiple AI paradigms and select the best diagnosis based on the data available [4]. An automated generic on-line diagnostic system (GOLDS) has been developed as a research prototype, using several AI techniques including a blackboard architecture, an expert system, and a neural network. The system was developed using CLIPS rules and objects.

Sample data and models used are for a spaceborne communications system. The software has been written in a manner to make it as domain independent as possible, so that it can be easily extended to other applications or completely different problem domains. GOLDS can be embedded in a wide variety of systems since it has been written as domain independent as possible.

INTRODUCTION

The Space Station Freedom will be the largest and most complex space structure ever built. It will incorporate numerous hardware and software systems. One of the most important is the Communications and Tracking (C&T) system. The function of the C&T system is to provide communications -- data, audio, and video -- between space and ground, as well as tracking capabilities for the onboard crew. As shown in Fig. 1, one of the major subsystems is the Assembly/Contingency S-Band Communications Subsystem (ACS.) The ACS is chosen as the problem domain for GOLDS, since it represents a critical facility of a type where diagnostic techniques are known.

The C&T system for the Space Station Freedom will also be larger and more complex than previous implementations of the functions required for the space shuttle and earlier manned space programs. To manage and control such a complex system is a challenging task. The size and the complexity of a manually operated C&T would demand too much of the astronaut’s time and effort. An astronaut's time in space is valued at more than $40,000 an hour [29, 30]. There is also a need to reduce ground costs by minimizing the number of ground support personnel. Automation is now perceived as the only viable technique to reduce human error, shorten response time, and reduce costs.



ACLGA - ACS Low Gain Antenna
ACHGA - ACS High Gain Antenna
ACTRA - ACS Transmitter/Receiver Amplifier
XPDR - Transponder|
ACBSP - ACS Baseband Signal Processor

Figure 1. Assembly/Contingency Subsystem

One desirable automated system is diagnosis of equipment faults. The ability of an automated system to diagnose equipment faults quickly and reliably could be the difference between mission success and a compromised or failed mission. Conventional algorithmic methods such as a simple decision-tree approach are not very effective for diagnostic applications when the solution space of the system is very large or complex. The search time to discover a solution may involve a combinatorial explosion and thus not be practical, especially in real-time, mission-critical applications such as the space station. Artificial Intelligence (AI) techniques, such as expert systems and artificial neural networks are better at reducing the size of the search space to yield a practical solution. Although the solution may not be optimum, it is often adequate when no optimum solution is practical.

Most AI techniques, including expert systems, case-based reasoning and others, are heuristics with associated strengths and weaknesses that work well for certain problem domains, but not in others [6, 7]. Depending on the amount of knowledge available concerning the fault (or perhaps multiple faults), different AI paradigms may give different diagnoses. In this investigation, we chose to implement multiple, cooperating technologies that are controlled by a blackboard

system. The purpose of that choice is to see if such a distributed architecture could overcome the limitations inherent in a single technology solution.

As illustrated in Figure 2, GOLDS consists of three independent and loosely coupled technical systems, and a blackboard managerial system. The technical systems are referred to as: the Domain Expert System (DES), which is an implementation of a rule-based expert system; an Artificial Neural System (ANS), which is a highly parallel, non-symbolic pattern matching system; and the Procedural Diagnostic System (PDS), which is a collection of procedural computer algorithms to map the symptoms to pre-computed diagnosed faults as determined from the problem domain fault net. The managerial system is called the Blackboard Management System (BBMS) since it provides the common data seen by all three paradigms for fault diagnosis, and selects the best overall answer from that proposed by each system. Like the DES, the BBMS is also an expert system. Depending on how much data is known about the fault, each system may suggest a fault diagnosis that is uncertain to some extent. The BBMS decides which is best (described in more detail later in this paper).

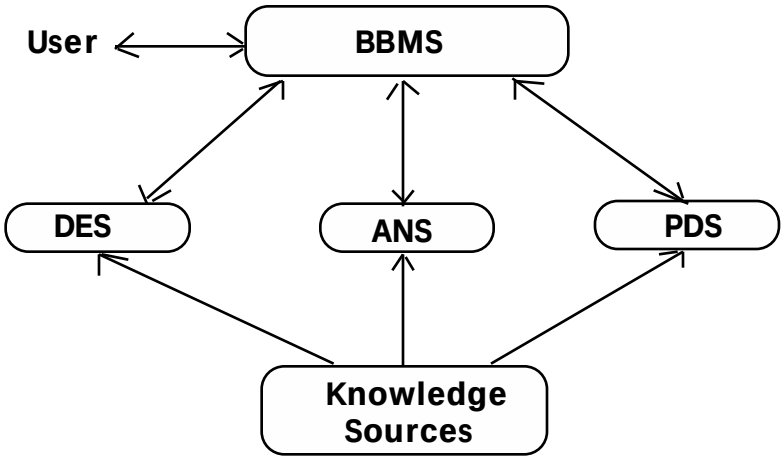


Figure 2. Overview of BOLDS

Both expert systems and artificial neural systems have had some successes when applied to diagnostic problems. However, they also have some weaknesses based on how much information about the fault is known. In real-world systems, there may not be enough time to do exhaustive testing, or it may be prohibitively expensive (like going to the physician). In these situations where time and/or cost are critical factors, it is better to obtain the best currently available diagnosis rather than to wait too long. In other words, the diagnosis may be 100% accurate but the patient died while waiting for the results.

The approach used by BOLDS is to use three different paradigms to diagnose any malfunction. Two of the techniques, the expert system and the artificial neural network, are AI techniques. The third technique is a procedural system based on a decision tree that will yield the optimum answer if enough information is known. Since different paradigms are used to diagnose the same problem, there is a need for a managerial system to select the best answer and supervise the three subordinate technical systems.

The concept of the BBMS is modified from the conventional blackboard architecture. Normally, the conventional blackboard has more than one independent knowledge source (KS), each of which is used to solve a part of the problem. This concept is depicted in Figure 3. For our prototype application, there is only one KS for all three technical systems since each one is expected to provide an answer.

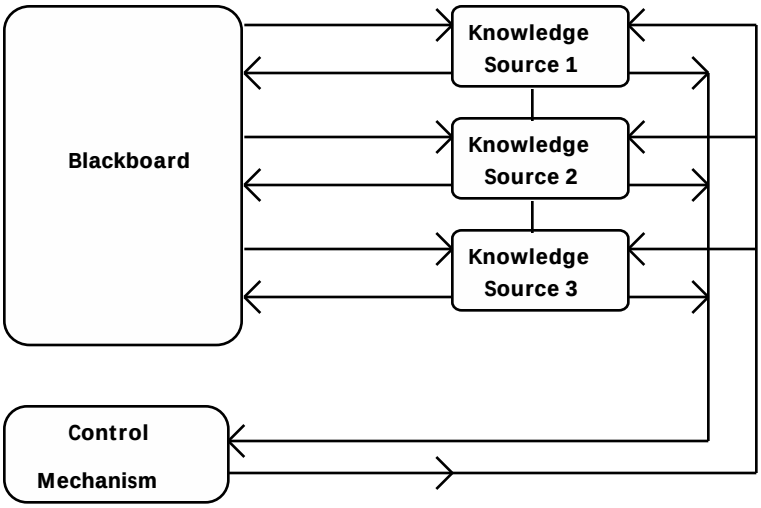


Figure 3. The Conventional Blackboard Architecture

The overall strategy of our prototype is for all three technical systems to diagnose the same problem, and the BBMS monitors their progress. After each of the technical systems has executed successfully, each gives its solution to the BBMS, and the BBMS evaluates the solutions derived by the three systems. Finally, the BBMS presents the ranked solutions to the user.

The three technical systems are designed as "generic" diagnostic systems. A new BOLDS system can be created for any new application by modifying or replacing data files rather than source code. For simplicity, as presently structured, only single-point failures are isolated; although extension to multiple failure diagnosis is feasible. It is thus assumed that a new problem will not arrive until the present one is solved.

APPROACH TO BLACKBOARD MANAGEMENT SYSTEM-BBMS

The BBMS is a knowledge-based expert system written using CLIPS (C-Language Implementation Production System). CLIPS was developed by the Software Technology Branch at NASA/Johnson Space Center. CLIPS is an object-oriented, expert system tool in which knowledge can be represented as rules or objects.

BOLDS is interactive, and allows the user and BBMS to communicate with each other. The BBMS is the interface between the user and the three technical systems. It relays the symptoms, the new knowledge, and the current sensor or signal readings to the technical systems as needed.

The BBMS reads a symptom as it is entered. For the Space Station problem domain, the symptom consists of

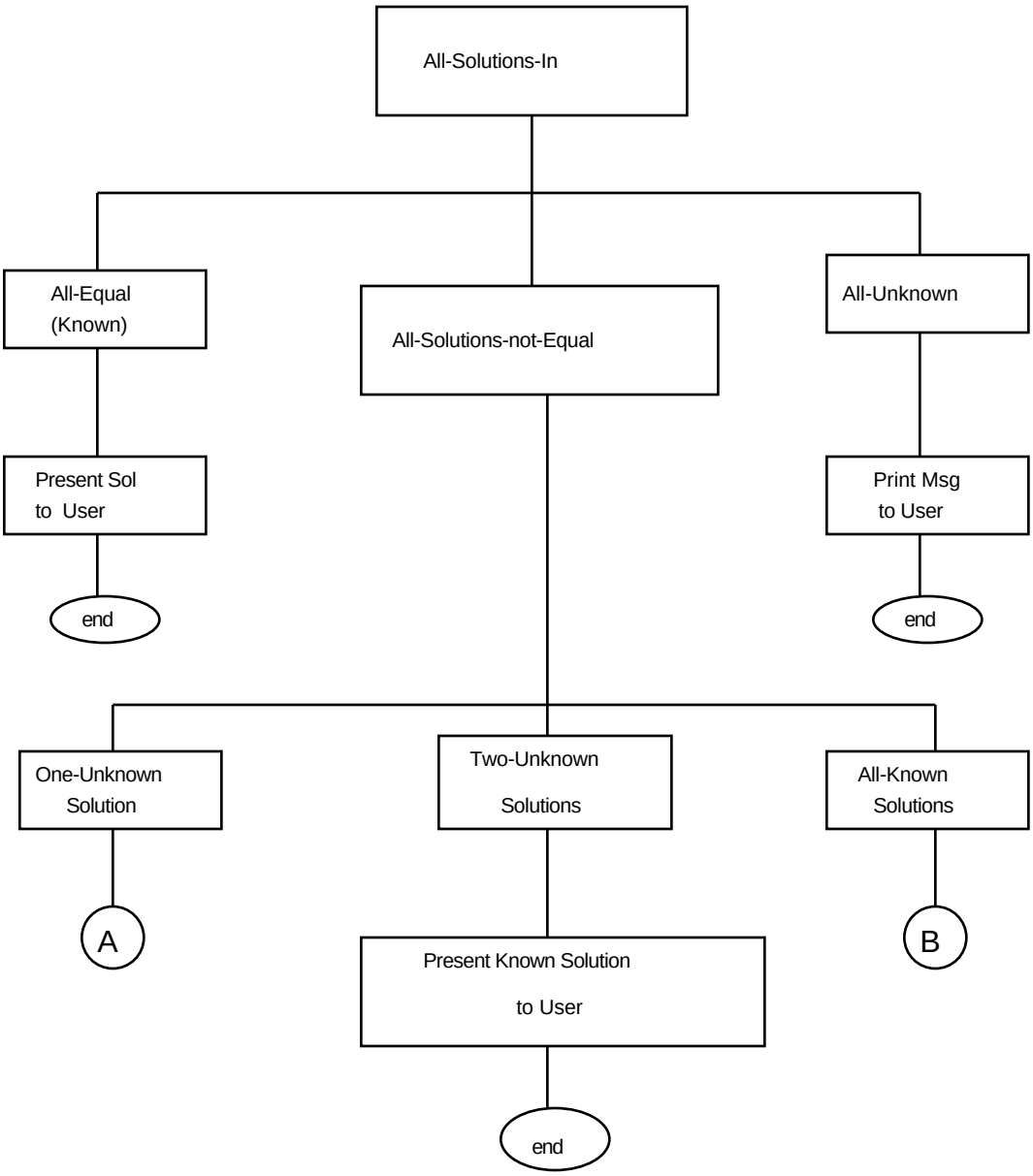
- Symptom identification (ID)
- Monitoring message ID (MSID or sensor ID)
- Corresponding reading.

For example, a working memory element from the DES takes the form (1 ACRFG29M2 200), to indicate "1" as the first symptom number for that problem, "ACRFG29M2" is a MSID (MeSsage IDentification), and "200" is the current sensor reading. The sensor reading could be numeric

data or a Boolean value representing the status of the sensor. If the reading is numeric, then the BBMS checks that reading against the minimum and maximum limits that are in its corresponding object. It then sends this information to all three technical systems. The three systems then execute concurrently, each sending their respective solutions back to the BBMS for evaluation.

The DES may need more information about certain sensors or the current signal level of some units in question. The DES then asks the BBMS for this information and the BBMS queries the user. The user enters the current reading using the keyboard. The BBMS reads the data given by the user, checks whether the reading is in allowed limits, and transmits this information to the DES.

Figure 4 illustrates in detail the decision tree by which the BBMS evaluates which solution of the three paradigms is best.



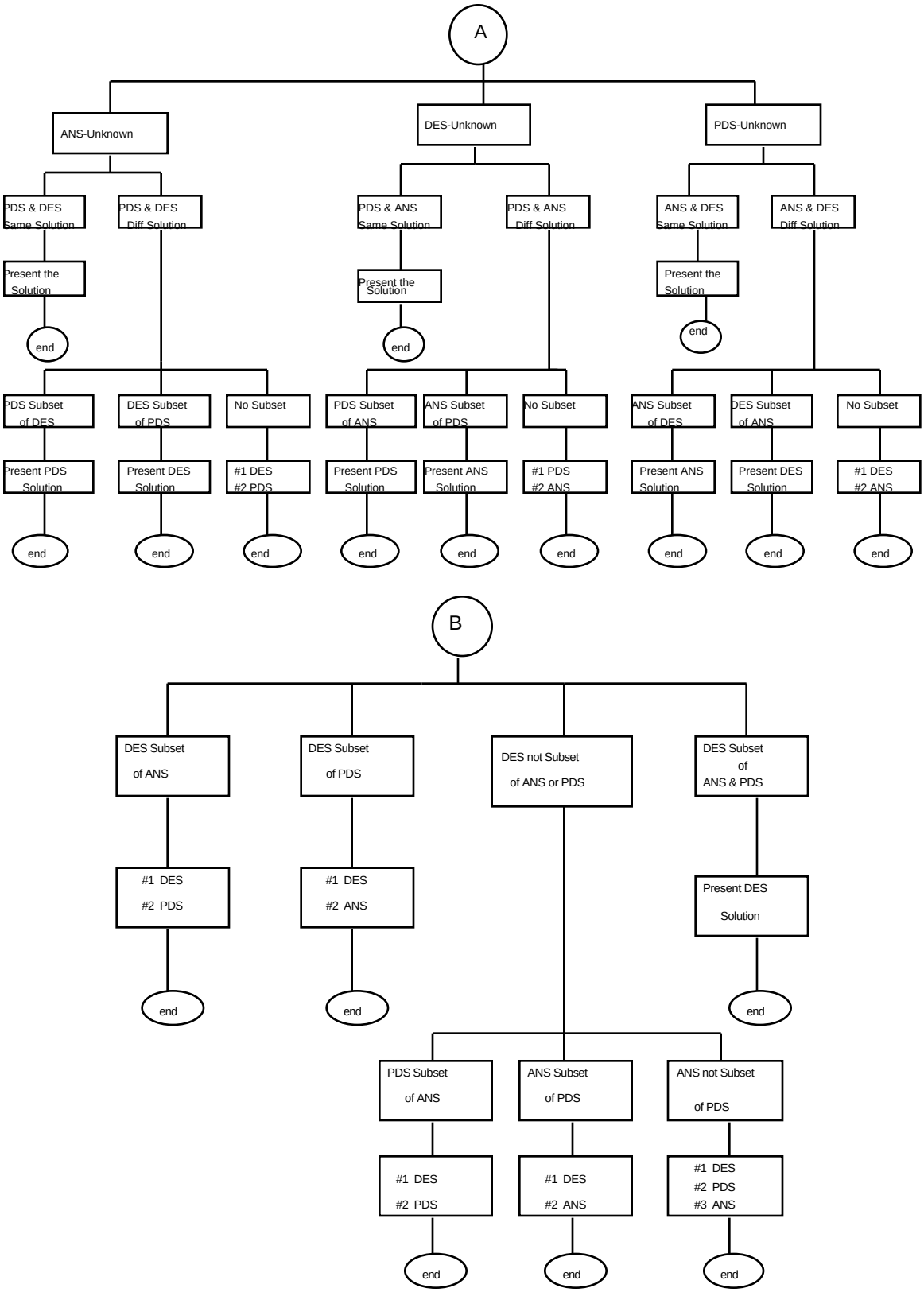


Figure 4. BBMS Decision Tree

Once, the BBMS receives the solutions from all three systems, it prints all the solutions on the screen. The BBMS then starts evaluating the solutions as follows:

- First the BBMS checks whether all the solutions are equal or not. If they are equal, then it presents the solution as one solution to the user without any ranking.

- If the solutions are not the same, then the BBMS begins the process of evaluating the solutions. The BBMS always has more confidence in the solution given by the DES, since the DES is essentially a decision tree that is guaranteed to give the correct answer if enough information is available. It has capabilities to ask for more information about the unit in question and can even check neighboring units of the suspected unit to investigate ancillary faults.

- The BBMS checks if any of the answers is indeterminate, i.e., no answer is possible. If the answer from one of the systems is “Unknown Problem”, then that system is eliminated from the evaluation process.

- If two systems' answers are “Unknown Problem”, then the third and the only answer is presented to the user as the first (but unreliable) choice.

- If two out of three systems agree with the answer then that answer may have the highest rank.

- If the three systems have different solutions and any solution is not a subset of any other solutions, then the solution derived by the DES gets the first rank, then the PDS's solution and finally the solution given by the ANS.

The best solution also depends upon which of the two systems have the same answer. If the PDS and the ANS agree on a solution, i.e., they come up with the same unit name, say A, which receives the signal from the unit B, then there is a very high possibility that unit B is the unit where the problem started. The DES is designed so that it can isolate the faulty unit more explicitly by backtracking the units. The PDS and the ANS have no explicit knowledge of the unit connectivity. Hence, in such cases, the solution given by the DES has the higher rank.

The DES normally comes up with a solution which contains the name of only one unit. However, the PDS and the ANS can come up with a solution that contains more than one unit. In such cases, the BBMS first checks whether the unit name derived by the DES is in the list of units of the other solutions. If so, the BBMS gives the highest rank to the common unit or units and ignores the remaining units.

The reasons for this evaluation are as follows: The DES is designed with more intelligence. It can justify its reasoning by giving explanations. Hence, its solution gets the first rank. The neural network's results are not always reliable if not trained with ample data [19]. The PDS is a conventional approach, so its solution is more reliable compared to the ANS's solution. The PDS is as reliable as the knowledge used to construct its bit strings, it is a completely deterministic and predictable system.

DES PARADIGM

The basic concept of an expert system is shown in Fig.5. The DES gets one symptom at a time from the BBMS. After receiving all the available symptoms, the DES is ready to fire one of the diagnostic rules. If the DES cannot fire any of the diagnostic rules, then it asks the BBMS for more needed information. When it gets the current sensor reading of the needed unit, it fires the

corresponding diagnostic rule. If a rule is fired then it comes up with a solution as explained before.

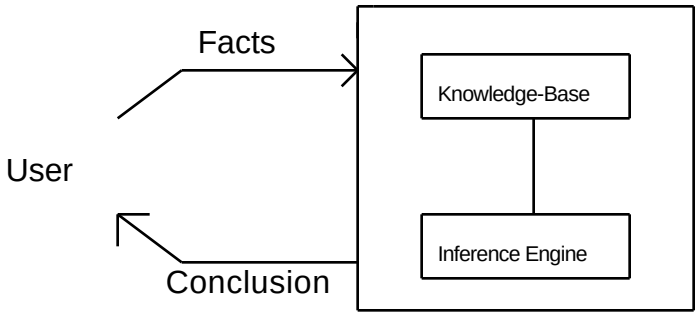


Figure 5. The Expert System Paradigm

The DES always saves its original solution before doing any further investigation. The DES has explicit knowledge of the connectivity of the units of the system to be diagnosed. It contains the rules to backtrack the suspected unit and check status and health of the previous units along the connective path. The DES will try to backtrack the unit or check the list of units to isolate the fault. If the unit does not have any inputs or outputs, or if the unit is a source unit, then the DES cannot backtrack that unit. The final derived solution along with the original solution is presented to the BBMS. There is also a possibility that the DES, even after trying to get more information, cannot fire any of the diagnostic rules. It then sends the message, "Unknown Problem" to the BBMS.

Not all of the units or SRUs in the ACS have sensors. If a failed unit does not have a sensor, it is not straight forward to pinpoint the problem for that failed unit. Hence, it is possible that the suspected unit or the solution derived by the systems is not accurate. There is a need to check the neighboring units. The DES has such capability. The DES identifies its original suspected unit by using the given symptoms and its diagnostic rules. It has the capability to do further investigations on its original suspect unit. It asks the BBMS for the sensor readings of the previous unit along the connectivity. The BBMS in turn asks the user to get this reading and passes the reading to the DES. Thus, the DES continues to check the previous units which are propagating signal to the suspected unit till it finds the sensor reading which is within the limits. Hence, the final solution derived by the DES has more confidence and it also gives an explanation of the derived solution.

ANS PARADIGM

A backpropagation network (BPN) simulator, written in C is used for the system's artificial neural system (ANS) subsystem in the BOLDS application. The BPN was selected as the most appropriate neural network architecture for this application due to its ability to generalize relationships between abstract pattern vectors [15]. However, an important design consideration for the system is to make the architecture as domain independent as possible, so that other applications may be easily fitted into the system architecture. We achieved that domain independence with the ANS subsystem by making the network simulator a single C-callable function. Thus, other neural network paradigms can easily be integrated into the BOLDS ANS subsystem by simply replacing the BPN simulator with another network paradigm simulator.

One of the most difficult aspects of using a neural network to perform complex pattern matching is the development of a suitable data representation scheme [31] for the application. For the prototype BOLDS application, we chose to implement a three-layer BPN, using two discrete

units to represent each problem symptom that the network would be asked to recognize. Table 1 illustrates the selected scheme. We chose to represent the problem in this manner, since there are three states for each symptom that had to be modeled. Since most neural network processing elements use bi-stable activation functions, it is necessary to find a way to represent three symptom states using binary units.

An additional benefit derived from this data representation scheme is the separation of similar patterns in Euclidean hyperspace. By making each symptom component of the input pattern vector orthogonal to each state, the pattern matching task performed by the neural network is simplified. To see why this is so, consider that each of the parallel processing elements implemented in a neural network uses a weighted-sum calculation, which is essentially an inner-product computation between two n-dimensional pattern vectors, to compute its input activation. Then, each unit responds with an output signal that is developed from the total input stimulation received by the unit. Thus, in a very real sense, the BPN used in this application is matching the symptoms contained in the input pattern vector with a set of predefined (or prelearned) symptom combinations that tend to indicate certain failure modes of the ACS system.

There are three layers in the BPN, which includes one hidden layer. The data representation in the ANS is summarized in Table 1.

Binary	Sensor Reading
00	Don't care
01	Low or False
10	High or True
11	Within Limit

Table 1

The input and the output data are in binary format. Symptoms are represented by a pair of binary numbers for the four different states: 00, 01, 10, and 11. The state 00 is the *Don't care* condition. The state 01 represents False or if the reading is less than the lower limit of the sensor reading. The state 10 means the current sensor reading status is True or the reading is greater than the upper limit of the sensor reading. The state 11 is used when the sensor reading is within limits.

PDS PARADIGM

The PDS is a procedural diagnostic system written in C. The PDS employs a technique called Fault Isolation using Bit Strings or FIBS [13]. Use of this method requires preprocessing of the fault net thereby producing bit strings that encapsulate the essential cause-effect relationships between problems and their symptoms [22]. Each bit string represents a symptom and contains information regarding all possible causes for that symptom. If a given problem can ever give rise (directly or indirectly) to a given symptom, then the bit corresponding to that problem is set in the symptom’s bit string. These bit strings are computed in advance and stored for subsequent use during fault isolation.

When a fault is detected, the PDS logically combines the bit strings producing a resulting bit string that has bits set corresponding to those problems any one of which could explain the set of observed symptoms. Thus, the method produces a list of possible causes for the symptoms that have been observed.

RESULTS FOR THE SPACE STATION ACS DIAGNOSTICS

Figures 6 and 7 display the simplified functional block diagrams of two Orbital Replacable Units (ORUs) of the ACS and their corresponding units and connectivity.

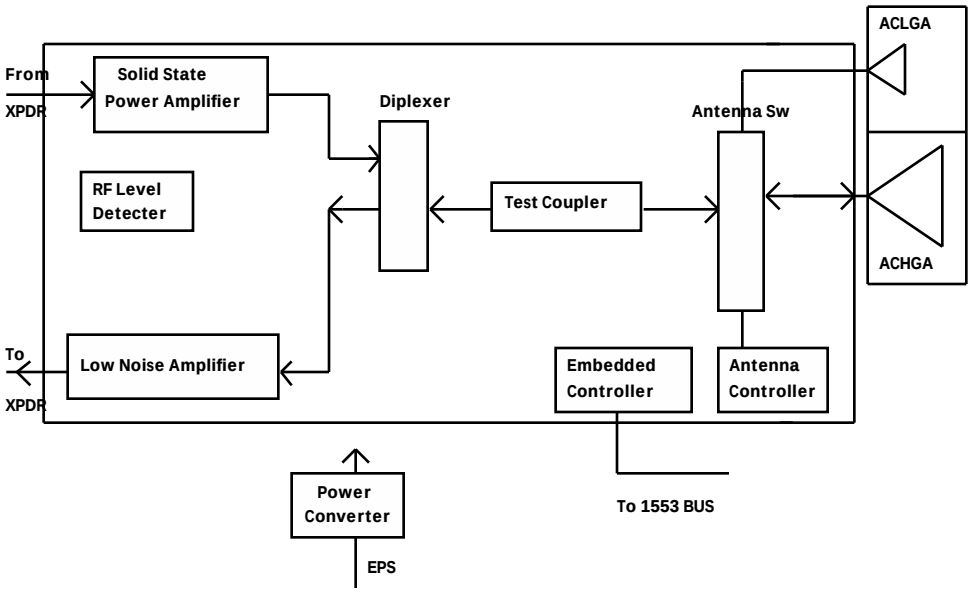


Figure 6. The ACS RF Group (ACRFG) Functional Block Diagram

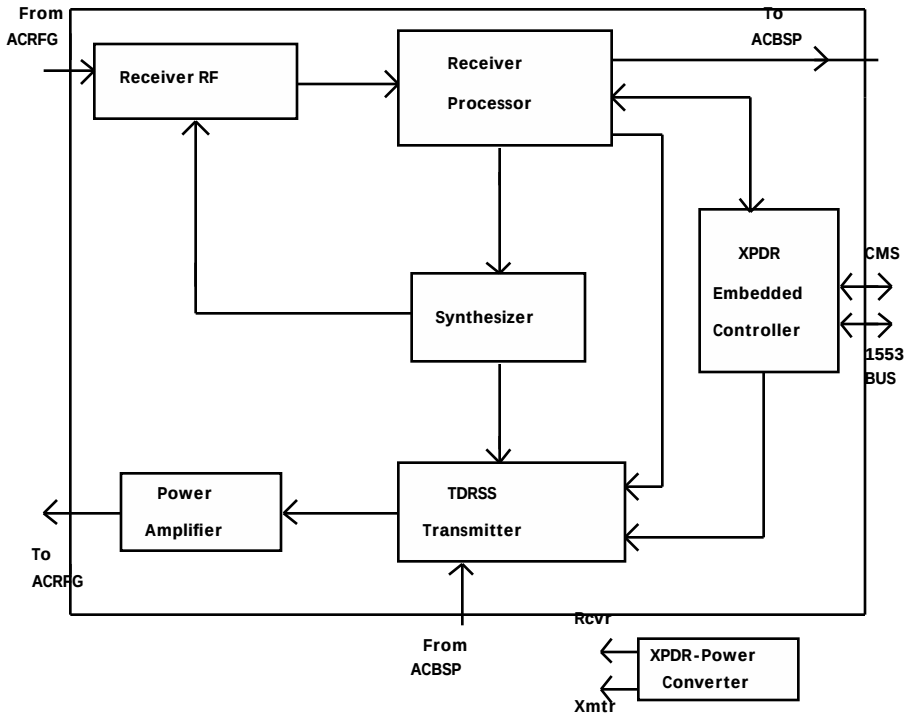


Figure 7. ACS-Transponder (XPDR) Functional Block Diagram

It has been shown that each of the diagnostic systems works well under different circumstances. Table 2 displays some examples of few symptoms and their corresponding suspected

malfunctioning unit(s) of the ACS system. The table also shows the confidence level of the solution given by each diagnostic system for that problem. The confidence levels are given as High, Med, and Low. An explanation of the word “High” as a confidence level is in order. One of the reasons why a DES might have a high confidence when only one symptom is available while the other two systems have low confidence factors is the interactive nature of an expert system

Problem numbers 1 and 2 have only one symptom and several suspected units. In such cases, only the DES is guaranteed to diagnose the problem to one malfunctioning unit. The ANS may converge to a solution which is considered most likely. The PDS will give the list of suspected malfunctioning units and will not do further investigation. The DES will check each unit in the list and also check the status, through the blackboard, of any units connected to these units. Thus, it will provide a final diagnosis with a high degree of certainty.

Problem Number	Number of Symptoms	Symptoms	Suspected Malfunction Unit/s	Diagnostic Systems		
				DES	ANS	PDS
1	1	RF Group to TDRSS Erroneous High Rate Data Output is True	High Gain Antenna (ACHGA), Antenna Switch, Solid State Power Amplifier,, ACRFG Embedded Controller	High	Low	Low
2	1	XPDR Embedded Controller Loss of Output is True	XPDR Embedded Controller, Receiver-Processor, XPDR Power Converter, Synthesizer	High	Low	Low
3	1 2 3	Solid State Power Amplifier-Output Power is Within Limit RF Group to 1553 BUS Loss of Data is True Solid State Power Amplifier-Reflected Power is High	Low Gain Antenna (ACLGA)	High	Med	High
4	1	XPDR Power Converter-Temperature is Out of Limit	XPDR Power Converter	High	High	High
5	1	RF Group Power Converter - Voltage/s is Out of Limit	RF Group Power Converter	High	High	High
6	1 2	XPDR Embedded Controller's Loss of Output XPDR Carrier Lock is False	Receiver Processor	High	Low	Med

Table 2. A Few Sample Faults

All the three systems can diagnose the problem with high certainty when symptoms (data) are complete and known to the systems (see problem number 3, 6). However, if the observed symptoms are not complete, e.g., in problem number 3, if symptom number 3 is not observed and the only remaining two symptoms are observed, then the PDS may not be able to reduce the diagnosis of the problem to one precise unit. It may reduce the problem to a list of units containing the actual malfunctioning unit. The ANS and the DES can solve the problem with the incomplete symptoms.

Some of the problems have only one symptom and one suspected unit (see faults 4 and 5). All three systems diagnose all such problems accurately without failing.

ANALYSIS AND CONCLUSIONS

BOLDS demonstrates that a blackboard architecture can be a good technique for integrating different diagnostic paradigms. By allowing different paradigms to work in parallel, a more accurate diagnosis is found more quickly and efficiently than by using one paradigm exclusively. The blackboard approach mirrors human problem solving in that people do not rely exclusively on a single paradigm to solve a problem. Instead, multiple problem solving approaches working in parallel can often yield a solution more quickly and efficiently with a higher level of confidence.

BIBLIOGRAPHY

1. Barr, Cohen, Feigenbaum, *Blackboard Systems, The Handbook of Artificial Intelligence*, Vol IV, Addison-Wesley, pp. 4-82, 1989.

2. Beck, Hal, Dan Bergondy, Joe Brown and Hamed Sare-Sarrf, "A Neural Network Data Analysis Tool for a Fault-Isolation System," *Proceedings of Second Workshop on Neural Networks: Academic/Industrial/NASA/Defense*, pp. 621-630, Alabama, February 1991.

3. Bigus and Goolsbey, "Integrating Neural Networks and Knowledge-Based Systems in a Commercial Environment," *Proceedings of the IJCNN-90-WASH-DC*, Vol. 2, pp. II-463 II-466, January 1990.

4. Boloor, Pratibha, "An Automated On-Line Diagnostic System (OLDS) Using Blackboard Architecture," *M.S. Thesis*, University of Houston Clear Lake, December 1992.

5. Blackboard Technology Group, Inc. "The Blackboard Problem-Solving Approach," *AI Review*, pp.27-32, Summer 1991.

6. Caudill, Maureen, "Using Neural Nets - Diagnostic Expert Nets Part 5," *AI Expert*, pp. 43-47, September 1990.

7. Caudill, Maureen, "Using Neural Nets: Hybrid Expert Networks Part 6," *AI Expert*, pp. 49-54, November 1990.

8. Caudill, Maureen, "Expert Networks," *Byte*, pp. 108-116, October 1991.

9. Corkhill, Daniel, "Blackboard Systems," *AI Expert*, pp. 41-47, September 1991.

10. Dixit V., "EDNA: Expert Fault Digraph Analysis Using CLIPS," *First CLIPS Conference Proceedings, Vol I*, pp. 118-124, August 1990.

11. Engelmore, Robert and Tony Morgan, *Blackboard Systems*, Addison-Wesley Publishing Company, 1988.

12. Everett, Dankel, Pierce Jones and J.W.Jones, "A Study in Acquiring Knowledge from an Expert," *Advances in Artificial Intelligence Research*, Vol 1, pp. 145-157, Jai Press Inc, London, England, 1989.

13. Feagin, Terry, "Real-Time Diagnostic Expert Systems Using Bit Strings," *Proceedings of the Fourth International Symposium on Methodologies for Intelligent Systems*, pp. 47-56, Oak Ridge National Laboratory, 1989.

14. Flores, L. and R. Hansen , “System Control Module Diagnostic Expert Assistant,” *First CLIPS Conference Proceedings,* Vol I, pp. 240-246, August 1990.

15. Freeman, James and David Skapura, *Neural Networks Algorithms, Applications, and Programming Techniques*, Addison-Wesley, 1991.

16. Giarratano, Joseph and Gary Riley , *Expert Systems - Principles and Programming*, PWS Publishing Company, 1989.

17. Guo and Nurre, “Sensor Failure Detection and Recovery by Neural Networks,” *IJCNN*, pp. I-221 - I-26, Vol.1, Seattle, Wa, 1991.

18. Hayes-Roth, Waterman, Lenat, *Building Expert Systems*, Addison-Wesley Publishing Co. Inc., 1983.

19. Hillman, David, “Integrating Neural Nets and Expert Systems,” *AI Expert*, pp. 54-59, June 1990.

20. Hoskins, Kaliyur and Himmelblau, “Insipient Fault Detection and Diagnosis using Neural Networks,” *Proceedings of the IJCNN-90,* San Diago, Vol. 1, pp. I-81 - I-86, 1990.

21. Krishnamraju, V. Penmatch , Kevin Reilly and Hayashi Yoichi, “Neural and Conventional Expert Systems: Competitive and Cooperative Schemes,” *Proceedings of Second Workshop on Neural Networks: WNN-AIND 91*, pp. 649-656, Alabama, February 1991.

22. Leibfried, Schmidt, Martens, Feagin, Garner, Overland, Glorioso, Lekkos and DeAcetis, “Control and Monitoring Communications Management System 3-B Prototype Development and Demonstration Experience,” RICIS Rept. June 1990.

23. Nii Penny, “Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures,” *The AI Magazine,* Vol VII No 2, pp. 38-53, Summer 1986.

24. Nii Penny, “Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective,” *The AI Magazine*, Vol VII No 3, pp. 82-106, Summer 1986.

25. Padalkar, Karsai, Biegl, and Sztipanovits, “Real-Time Fault Diagnostics,” *IEEE Expert*, pp. 75-85, June 1991.

26. Pohl, Jens and Myers, “ICADS: A Cooperative Decision Making Model with CLIPS Experts,” *Second CLIPS Conference Proceedings,* Vol 1, pp. 102-111, 1991.

27. Pomeroy, Bruce and Russell, “Blackboard Approach for Diagnosis in Pilot's Associate,” *IEEE Expert*, IEEE Computer Society, Vol 5 No 4, August 1990.

28. Raoult, Oliver, “Survey of Diagnosis Expert Systems,” *Knowledge Based Systems for Test and Diagnosis*, Elsevier Science Publisher B. V., North Holland, 1989.

29. Schmidt, Oron, “Expert Systems for the Control and Monitoring of the Space Station Communications and Tracking System,” NASA/JSC, 1986.

30. Schmidt, Oron, “Automated Communications and Tracking Interfacing Operations Necleus (ACTION),” *The AI Magazine*, pp. 112-113, Summer 1986.

31. Skapura, David M., *Building Neural Networks*, Addison-Wesley Publishing Co. Reading, MA, manuscript in preparation.

32. Waterman, Donald, *A Guide to Expert Systems*, Addison-Wesley Publishing Company, 1986.

DYNACLIPS (DYNAMIC CLIPS): A DYNAMIC KNOWLEDGE EXCHANGE TOOL FOR INTELLIGENT AGENTS

Yilmaz Cengeloglu
P.O. Box 4142
Winter Park, FL, 32793-4142
E-mail: yil@engr.ucf.edu

Soheil Khajenoori, Assoc. Prof., Embry-Riddle Aeronautical University,
Department of Computer Science,
Daytona Beach, FL, 32114, E-mail: soheil@erau.db.erau.edu

Darrell Linton, Assoc. Prof., University of Central Florida,
Department of Electrical and Computer Engineering,
Orlando, FL, 32816, E-mail: dgl@engr.ucf.edu

ABSTRACT

In a dynamic environment, intelligent agents must be responsive to unanticipated conditions. When such conditions occur, an intelligent agent may have to stop a previously planned and scheduled course of actions and replan, reschedule, start new activities and initiate a new problem solving process to successfully respond to the new conditions. Problems occur when an intelligent agent does not have enough knowledge to properly respond to the new situation. DYNACLIPS is an implementation of a framework for dynamic knowledge exchange among intelligent agents. Each intelligent agent is a CLIPS shell and runs a separate process under SunOS operating system. Intelligent agents can exchange facts, rules, and CLIPS commands at run time. Knowledge exchange among intelligent agents at run time does not effect execution of either sender and receiver intelligent agent. Intelligent agents can keep the knowledge temporarily or permanently. In other words, knowledge exchange among intelligent agents would allow for a form of learning to be accomplished.

INTRODUCTION

Applications of expert systems to variety of problems are growing rapidly. As the size and complexity of these systems grow, integration of independent cooperating expert systems is becoming a potential solution approach to large scale applications. In this paper, the blackboard model of distributed problem solving is discussed and architecture, implementation and usage of DYNACLIPS is explained.

Distributed Problem Solving (DPS)

Distributed problem solving in artificial intelligence is a research area which deals with solving a problem in a distributed environment through planning and cooperation among a set of intelligent entities *(i.e., agents)*. Each intelligent agent can run in parallel with other intelligent agents. Intelligent agents may be geographically distributed or operate within a single computer. An intelligent agent may possess simple processing elements or a complex rational behavior. A paramount issue in DPS is the communication and information sharing among participating intelligent agents, necessary to produce a solution. The blackboard model of problem solving is one of the most common approaches in the distributed artificial intelligence area. In the following section we will focus on blackboard architecture as a model for distributed problem solving.

Blackboard Model of Distributed Problem Solving

The blackboard architecture (BBA) is one of the most inspiring cooperative problem solving paradigms in artificial intelligence. The approach had generated much interest among researchers. BBA is a relatively complex problem solving model prescribing the organization of knowledge, data and the problem-solving behavior within an overall organization. The blackboard model is becoming a useful tool for complex applications whose solution requires a set of separate though interrelated sources of knowledge and expertise.

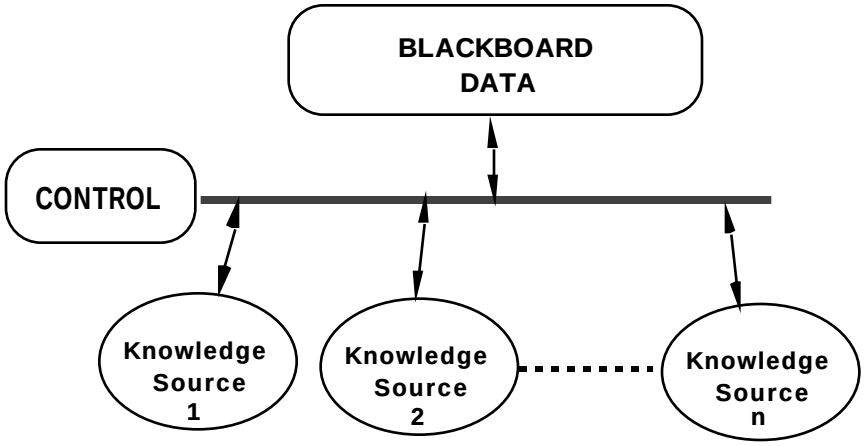


Figure 1. Blackboard Model For DSP

A blackboard model contains blackboard, control, and knowledge sources. Figure 1 shows a basic blackboard model for DPS applications. The Knowledge Sources are the knowledge needed to solve the problem; they are kept separate and independent. Each knowledge source can use different knowledge representations techniques. The Blackboard Data is a global database that contains problem-solving states. The knowledge sources produce changes to the blackboard that lead incrementally to the solution of the problem being solved. Communication and interaction among the knowledge sources takes place solely through the blackboard. The Control determines the area of the problem solving space on which to focus. The focus of attention can be either the knowledge sources, the blackboard or a combination of both. The solution is built one step at a time by using a cooperation of different knowledge sources. The blackboard model is a complete parallel and distributed computation model. The parallel blackboard model involves the parallel execution of knowledge sources and the control component. The distributed blackboard model involves the communication of blackboard data among blackboard subsystems. The main issue here is to decide what to communicate and where and when to send data. The blackboard systems are being used increasingly in real time systems. Architectural extensions to the basic blackboard model can be added to increase its performance for real time applications. [1,2,3]

An Overview of Existing BBA Tools

A number of BBA tools are reported in the literature. Here we provide an overview of some of these systems.

BB_CLIPS (Blackboard CLIPS) [4, 5] is an extended version of CLIPS version 4.3 developed by National Research Council of Canada. In BB_CLIPS, each CLIPS rule or group of rules serves as a knowledge source. The fact base of BB_CLIPS serves as the blackboard, and its agenda manager serves as the scheduler.

RT-1 architecture [6] is a small-scale, coarse-grained, distributed architecture based on the blackboard model. It consists of a set of reasoning modules which share a common blackboard data and communicate with each other by “signaling events”.

PRAIS (Parallel Real-Time Artificial Intelligence Systems) [7] is an architecture for real-time artificial intelligence system. It provides coarse-grained parallel execution based upon a virtual global memory. PRAIS has operating system extensions for fact handling and message passing among multiple copies of CLIPS.

MARBLE (Multiple Accessed Rete blackboard linked Experts) [8] is a system that provides parallel environment for cooperating expert systems. Blackboard contains facts related to the problem being solved and it is used for communication among expert systems. Each expert shell in the system keeps a copy of the blackboard in its own fact base. Marble has been used to implement a multi-person blackjack simulation.

AI Bus [9] is a software architecture and toolkit that supports the construction of large-scale cooperating systems. An agent is the fundamental entity in the AI bus and communicates with other agents via message passing. An agent has goals, plans, abilities and needs that other agents used for cooperation.

GBB (Generic Blackboard) [10,11] is toolkit for developers needs to construct a high-performance blackboard based applications. The focus in GBB is increasing the efficiency of blackboard access, especially for pattern-based retrieval. GBB consist of different subsytems : a blackboard database development subsystem, control shells, knowledge source representation languages and graphic displays for monitoring and examining blackboard and control components. GBB is an extension of Common Lisp and CLOS (Common Lisp Object System).

GEST (Generic Expert System Tool) [12] has been developed by Georgia Tech Research Institute. The main components of the GEST are the central blackboard data structure, independent experts or knowledge sources and the control module. The blackboard data structure holds the current state of the problem solving process. It is also common communication pathway among knowledge sources.

CAGE and POLIGON [13] have been developed at Stanford University. They are two different frameworks for concurrent problem solving. CAGE is a conventional blackboard system which supports parallelism at knowledge sources level. The knowledge source, the rules in the knowledge source, or clauses in a rule can be executed in parallel. CAGE is a shared memory multiprocessing system. POLIGONs' functionality is similar to CAGE.

Hearsay-II [2,3,14] is a speech understanding system developed at Carnegie-Mellon University. Hearsay-II provides a framework that different knowledge sources cooperate to solve a problem.

More recently, significant work is being done to develop Knowledge Interchange Formats (KIF) and Knowledge Query and Manipulation languages (KQML) by Stanford University [15,16]. KIF is a computer-oriented language for the interchange of knowledge among disparate programs that is written by different programmers, at different times, in different languages. KIF is not a language for the internal representation of knowledge. When a program reads a knowledge base in KIF, it converts the knowledge into its own internal form. When the program needs to communicate with another program, it maps its internal data structures into KIF. KQML messages are similar to KIF expressions. Each Messages in KQML is one piece of a dialogue between the sender and receiver programs.

IMPLEMENTATION OF DYNACLIPS

Using the SunOS operating system multiprocessing techniques and interprocess communication facilities, we have developed a prototype system on a Sun platform to demonstrate the dynamic knowledge exchange among intelligent agents. In the following sections we provide a short overview of SunOS InterProcess Communication Facilities (IPC) and describe the implementation aspect of the DYNACLIPS.

InterProcess Communication Facilities (IPC)

Interprocess Communication involves sharing data between processes and coordinating access to shared data. The SunOS operating system provides several facilities and mechanism by which processes can communicate. These include Messages, Semaphores and Shared Memory.

The *Messaging facility* provides processes with a means to send and receive messages, and to queue messages for processing in an arbitrary order. Messages can be assigned specific types and each would have an explicit length. Among other uses, this allows a server process to direct message traffic between multiple clients on its queue. Messages sent to the queue can be of variable size. The application programmer must insure that the queue space limitations are not exhausted when more than one process uses the same queue. The process owning the queue must establish the read/write permissions to allow/deny other processes access to the queue. Furthermore, it is the responsibility of the owner process to remove the queue when it is no longer in use or prior to exiting.

Semaphores provide a mechanism by which processes can query or alter status information. They are often used to monitor and control the availability of system resources, such as shared memory segments. Semaphores may be operated as individual units or as elements in a set. A semaphore set consists of a control structure and array of individual semaphores.

Shared memory allows more than one process at a time to attach a segment of physical memory to its virtual address space. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent contentions [17].

Architecture of DYNACLIPS

In this section we provide an overview and discussion of each component of the DYNACLIPS. Figure 2 represents the overall architecture of the DYNACLIPS. Shared memory has been used to implement the system main blackboard to broadcast messages from control to intelligent agents. Message queues have been used to transfer messages from the intelligent agents to the control. Semaphores are used to make intelligent agents and control to sleep or wake up in order to reduce the load on CPU. Each intelligent agent and control has an input/output port to the world outside of the application framework to interface with the other processes outside of their environment. These outside processes might be any program that uses the application framework. In DYNACLIPS, intelligent agents and control can also use IPC facilities to interface with the outside programs.

Control

The control component of the system has been implemented using the C programming language. It runs as a separate process and communicates with the other processes using IPC facilities. The control can be loaded and executed by entering the word “control” at SunOS prompt. The control always has to be loaded first. Once the control is loaded, it creates the three incoming control message queues for the requests coming from the intelligent agents, one shared memory to be

used as the main system blackboard and two semophores for control and intelligent agents. The control has read/write access to the main system blackboard and has only read access from the message queues. If there is no request from intelligent agents, control sleeps until any agent makes a request.

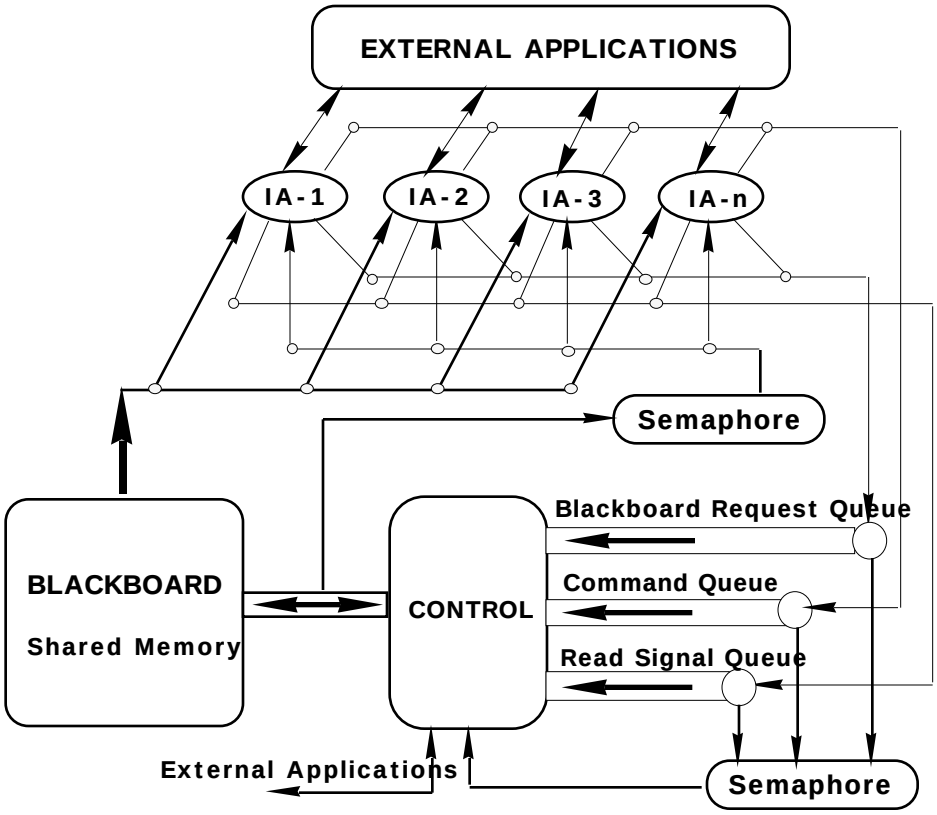


Figure 2. Architecture of DYNACLIPS

The DYNACLIPS takes advantage of the multiprocessing capabilities of the SunOS operating system. A message facility has been used for setting up a communication link from the intelligent agents to the control. The control creates incoming queues to receive messages from the intelligent agents. These control message queues use “First-In First-Out” (FIFO) methodology as shown in Figure 3.

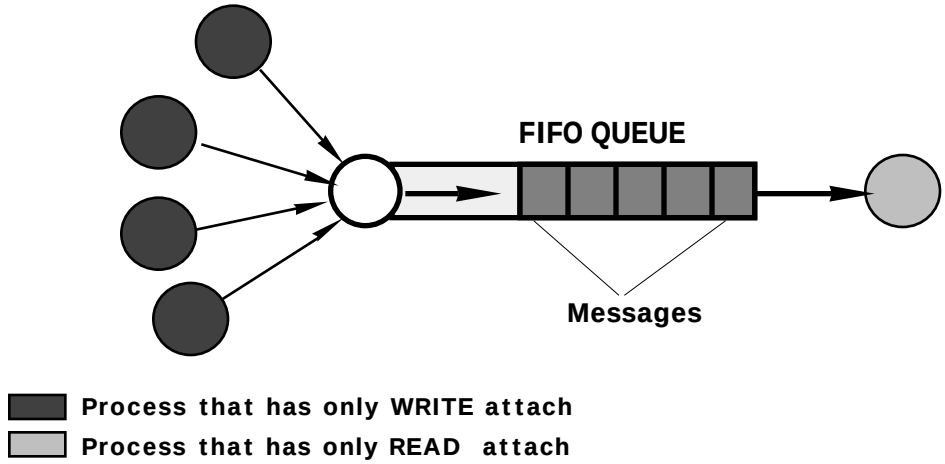


Figure 3. Message Transfer Among Processes

In order to communicate with the control process, intelligent agents send their requests to the control message queues. Intelligent agents have only write permission to these queues. Message queue facility of IPC can also be utilized to allow the control module and/or intelligent agents to communicate with other application programs running outside of their environment.

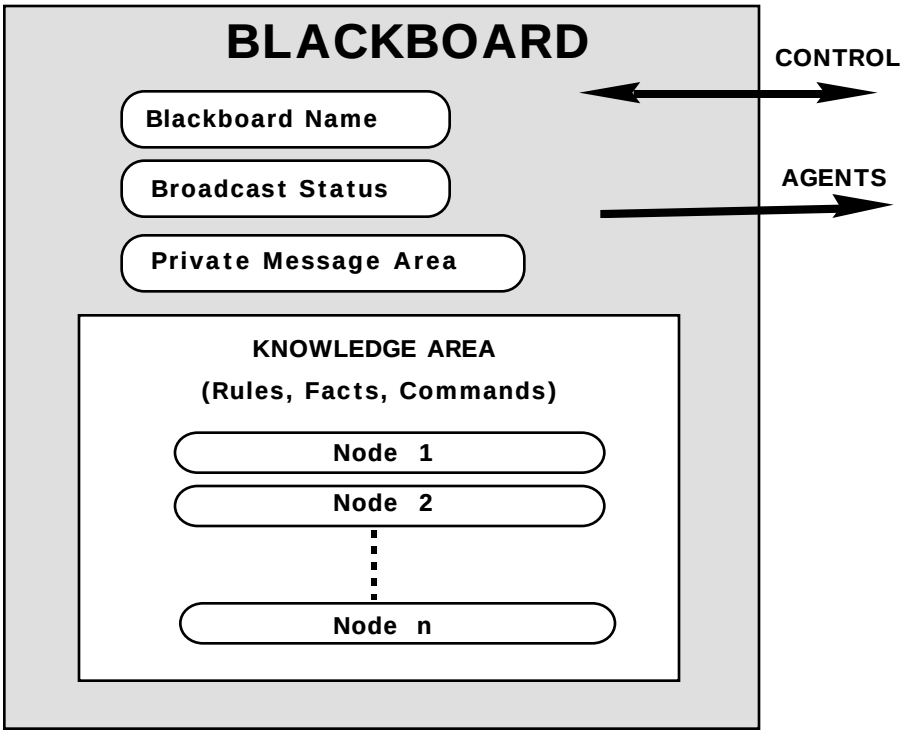


Figure 4. Blackboard

Blackboard

The blackboard is a shared memory that holds information and knowledge that intelligent agents use. The blackboard is organized by the control component. The blackboard has four different

members as shown in Figure 4. These members are: 1) *Blackboard name* which holds the name of the blackboard that is being used, this is necessary if multiple blackboards are in use, 2) *Blackboard status* which is a counter that is incremented after each update of the blackboard, 3) *Private message area* which is used by the control component to send system related messages to the intelligent agents, 4) The *knowledge area* which holds, rules, facts and commands that needs to be broadcast to the intelligent agents.

Shared Memory facility of IPC has been used to broadcast messages to all intelligent agents via the control. Figure 5 represents broadcasting data using shared memory configuration. When a process that has a write attach, writes a message to the shared memory, this message will be visible to all processes that have read attach to shared memory. When there is more than one process able to write to shared memory, semaphores should be used to prevent processes accessing the same message space at the same time. In DYNACLIPS semaphores were not used for shared memory, because the control component is the only process that has read and write accesses to the shared memory and intelligent agents can only read from the shared memory.

In DYNACLIPS, there is a local blackboard for each intelligent agent and one main blackboard for the control. The DYNACLIPS uses the shared memory facility to implement the main system blackboard. This implementation was possible because all intelligent agents run on the same computer. If intelligent agents were distributed on different computer systems, then a copy of the main blackboard needs to be created and maintained in each computer system.

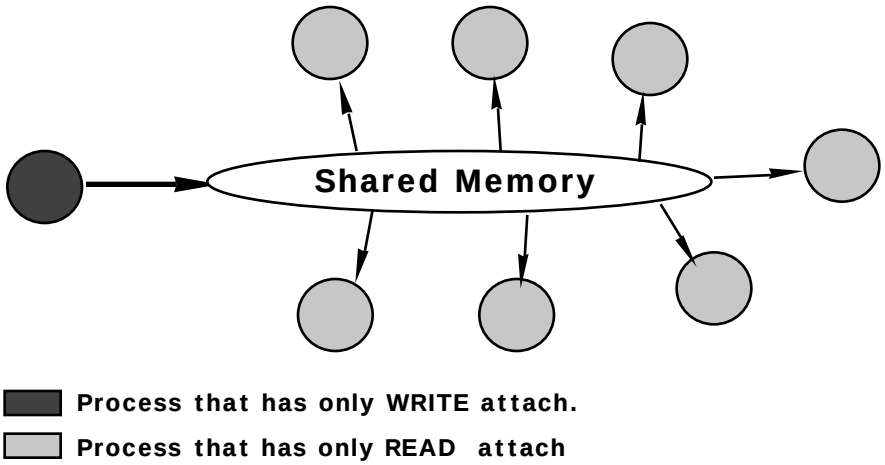


Figure 5. Message Broadcasting With Shared Memory

Intelligent Agents

An intelligent agent can be any application, such as an expert system shell, that is able to use IPC facilities on the SunOS operating system. In this prototype system, CLIPS (C Language Production System) was used as intelligent agents. CLIPS [18] is an expert system shell which uses a rule-based knowledge representation scheme. We have extended the CLIPS shell by adding a set of functions to provide the necessary capabilities to use IPC facilities.

The following command should be executed at SunOS prompt to load and activated a CLIPS based intelligent agent.

```
dynaclips <Knowledge base name> <Intelligent agent name> [Group name]
```

The *Knowledge base name* is a file that contains CLIPS rules. The *Intelligent agent name* and *Group name* are names given to intelligent agent by the user or application program that loads this knowledge base. Multiple copies of the CLIPS expert system shell, each representing an agent, can be loaded and activated at the same time using the above command. *Intelligent agent name* and *Group name* are also inserted as initial fact to intelligent agents. Following will be initial facts in the intelligent agents:

```
(knowledge name is  <Intelligent agent name>)
(member of group  <Group name>)
```

Once an intelligent agent is loaded, 1) it finds out the names of the control message queues and attaches itself to the queues as a writer. 2) It finds out the name of the main blackboard and attaches itself to it as a reader. 3) It sends a message to control component to inform that it has joined the system. If there is no rule to fire and nothing is changed in the blackboard, an intelligent agent sleeps until something changes in the blackboard.

As we mentioned previously, we have added number of functions to the CLIPS shell. The following functions can be used by an intelligent agent to communicate with the control as well as with other intelligent agents.

```
(ADD_BB_STR  <message string>)
(DEL_BB_STR  <message string>)
(CHANGE_GROUP_NAME  <new_group_name>)
(EXIT_CLIPS)
```

ADD_BB_STR *(Add BlackBoard String)* is called when an intelligent agent wants to add a message to the main blackboard. DEL_BB_STR *(Delete BlackBoard String)* is called when an intelligent agent wants to delete a message from the main blackboard. CHANGE_GROUP_NAME is called when intelligent agent needs to change group. EXIT_CLIPS is called when an intelligent agent is exiting from the system permanently. In the above commands, *Message string* takes the following format :

```
"<destination>    <type>    <message>"
```

The *Destination* field should be the name of an intelligent agent or group currently active in the system, or “ALL” which specifies that the message should be received by all active intelligent agents. *(In the above format, blank characters were used to separate each field.)*

The *Type* field should be “FACT”, “RULE” or “COMMAND”, which describes the type of the message in the *message string*. If the *type* is FACT, the *message* will be added or deleted, depending on the functions, to shared fact base of the intelligent agent specified by the *destination* field. If the *type* is RULE, the *message* will be added or deleted to the dynamic knowledge base of the intelligent agent(s) as specified in the *destination* field. If the *type* is COMMAND, then *message* will be executed as a command by the intelligent agent specified in the *destination* field. Commands are always removed from the main blackboard after intelligent agents receive a copy of the blackboard.

Message can contain facts, rules or commands. Since the current implementation of the prototype system only uses CLIPS shell to represent an intelligent agent, we have chosen to follow the syntax of the CLIPS to represent facts, rules or commands. If another expert system shell is used to represent an intelligent agent, this common syntax should be observed to transfer facts, rules or commands. It is the intelligent agent's responsibility to translate this common syntax to its own internal syntax.

The following sections describe, in more detail, the process of transferring facts, rules, and commands among intelligent agents.

Fact Transfer Among Intelligent Agents

An intelligent agent can send fact(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Facts stay in the main blackboard until removed by the sender intelligent agent or by other intelligent agent(s) that has/have the permission to delete the fact. ADD_BB_STR and DEL_BB_STR commands are used to insert, or remove fact(s) from the main blackboard. The following examples show how facts transferred among intelligent agents in the system.

Given the following information:

?y *is a string containing "ALL FACT Hi there, I have just joined the system"*
?x *is a string containing "IA-2 FACT apple is red"*
IA-2 *is the name of an intelligent agent active in the system.*

Then:

(ADD_BB_STR ?y) adds the message *(i.e., the fact) "Hi there, I have just joined the system"* to the main blackboard as a fact and all intelligent agents insert this fact to their internal shared fact base.

(DEL_BB_STR ?y) deletes the message *(i.e., the fact) "Hi there, I have just joined the system"* from the main blackboard as well as from internal shared fact base of any intelligent agent that has this fact in its shared fact base.

(ADD_BB_STR ?x) adds *"apple is red"* to the main blackboard as a fact and only IA-2 is allowed to insert this fact to its shared fact base.

(DEL_BB_STR ?x) deletes *"apple is red"* from the main blackboard and causes IA-2 to remove this fact from its shared fact base.

Command Transfer Among Intelligent Agents

An intelligent agent can send command(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Commands do not remain on the main blackboard, the receiver intelligent agent executes the command immediately upon its arrival. Commands are deleted by the receiver intelligent agent(s) as soon as they are executed. ADD_BB_STR function should be used for transferring commands among intelligent agents. DEL_BB_STR function is not available on COMMAND type. Following examples demonstrate how commands are transferred among intelligent agents. In the following examples all the commands are standard commands available in the CLIPS expert system shell.

Given the following information :

?y *is a string containing "ALL COMMAND (rules)"*
?x *is a string containing "IA-2 COMMAND (watch facts)"*
?z *is a string containing "GROUP1 COMMAND (CHANGE_GROUP_NAME "GROUP2")"*
IA-2 *is the name of the an intelligent agent active in the system.*

Then:

(ADD_BB_STR ?y) all intelligent agents in the system execute *(rules)* command which means print all rules in the intelligent agent knowledge base.

(ADD_BB_STR ?x) IA-2 executes *(watch facts)* command.

(ADD_BB_STR ?z) all intelligent agents in the GROUP1 will change their group name to GROUP2.

All CLIPS commands are supported by the DYNACLIPS. Hence, an intelligent agent can modify the knowledge of other intelligent agents via sending the appropriate command. Application programmer should be careful when designing the system since it is possible to remove static knowledge and local facts of the intelligent agent receiving the commands.

Rule Transfer Among Intelligent Agent

An intelligent agent can send rule(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Rules stay on the main blackboard until removed by the sender intelligent agent or other intelligent agent(s) that has the right permission to delete the rule. CLIPS format should be followed to represent rules. The type RULE should be used in the type field of the message string. ADD_BB_STR and DEL_BB_STR commands can be used to add or delete rules from the main blackboard. The following presents examples for transferring rule among intelligent agents.

Given the following information :

?x *is a string containing* "IA-2 RULE *(defrule* ***rule1*** *(apple is red) => (facts)) "*
?y *is a string containing* "ALL RULE *(defrule* ***rule2*** *(red is color) => (facts)) "*
IA-2 *is the name of an intelligent agent active in the system.*

Then :

(ADD_BB_STR ?x) rule1 will be added to the main blackboard and only IA-2 can insert the ***rule1*** into its dynamic knowledge base.

(ADD_BB_STR ?y) rule2 will be added to the main blackboard and all intelligent agents can insert ***rule2*** into their dynamic knowledge base.

Knowledge Transfer Among Intelligent Agents

Knowledge can be exchanged among intelligent agents by using combination of facts, rules and commands transfers. Different methodologies can be used for knowledge transfer; knowledge can be exchanged among intelligent agents in temporary or permanent bases.

Under temporary knowledge transfer option, the sender intelligent agent specifies the rule(s) that needs to be transferred as well as specifying when the rules needs to be removed from dynamic knowledge base of the receiver intelligent agent. The following example shows how to transfer a temporary knowledge among intelligent agents.

Given the following information :

?x *is a string that contains the following :*

" ALL COMMAND (defrule rule1 (lights are on) =>

(turn of the lights)

(assert (lights are off))) "

?y *is a string that contains the following :*

" ALL COMMAND (defrule rule2 (lights are off) =>

(undefrule rule1)

(undefrule rule2)) "

Then:

(ADD_BB_STR ?x)

(ADD_BB_STR ?y)

In the above example, two rules were broadcasted to all intelligent agents in the system. All intelligent agents will insert these two rules into their dynamic knowledge base. The rules will remain in the intelligent agent's dynamic knowledge base until *rule1* is fired. *Rule2* will be fired after *rule1,* which would cause *rule1* and itself to be removed from the intelligent agents' dynamic knowledge bases. Using type COMMAND will cause that the two rules be deleted from the main blackboard as soon as all intelligent agents have read them into their dynamic knowledge base. By eliminating the second function *(i.e., (ADD_BB_STR ?y))* from the previous example, the rule can be placed permanently in the receiver intelligent agents' knowledge bases. Hence, the knowledge encoded in the rule can be used by the receiver intelligent agent from that point on. The following example demonstrates this concept.

Given the following information

?x *is a string that contains the following :*

" ALL COMMAND (defrule rule1 (lights are on) =>

(turn of the lights)

(assert (lights are off)) "

Then:

(ADD_BB_STR ?x)

Type RULE should be used to transfer knowledge when the intelligent agents are joining, exiting or re-joining the framework continuously. In this case, knowledge stays in the main blackboard until deleted explicitly by the sender intelligent agent.

CONCLUSIONS AND FURTHER STUDIES

By introducing simple communication protocols among intelligent agents, we have introduced a framework through which intelligent agents can exchange knowledge in a dynamic environment. Using the DYNACLIPS common knowledge can be maintained by one intelligent agent and broadcasted to the other intelligent agents when necessary.

In a dynamic environment, intelligent agents must be responsive to unanticipated conditions. When such conditions occur, an intelligent agent may be required to terminate previously planned and scheduled courses of action, and replan, reschedule, start new activities, and initiate a new problem solving process, in order to successfully respond to the new conditions. Problems occur when an intelligent agent does not have sufficient knowledge to properly respond to the

new condition. In order to successfully respond to unexpected events in dynamic environments, it is imperative to have the capability of dynamic knowledge exchange among intelligent agents.

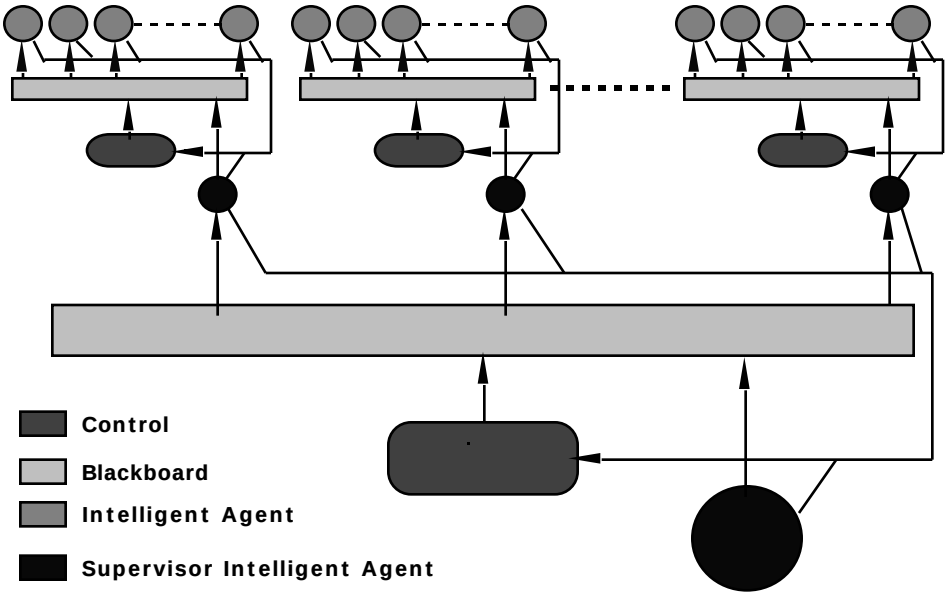


Figure 6. An Example of a Framework which Includes Multilayer Aspects

We believe that dynamic knowledge exchange would be an important feature for any application in which unanticipated conditions or events occur. Using the proposed dynamic knowledge exchange capability, cooperative problem solving sessions can be initiated where each intelligent agent can share its problem relevant knowledge with other intelligent agents to resolve the problem. An obvious advantage of this capability is the elimination of redundant knowledge and hence the improved utilization of the system memory capacity. In addition, by using this framework a form of learning can take place and thus additional problem solving knowledge is created.

The basic framework presented in this research could be extended to include a multilayer environment. This can be done by providing supervisory blackboard and control components to create a single entity by combining separate frameworks. (See Figure 6.)

The proposed framework can easily be expanded to accommodate more than one blackboard when it is necessary. In this case one control module is associated with each blackboard as in Figure 7. The basic process of communication with the control modules would be the same as presented in the previous sections.

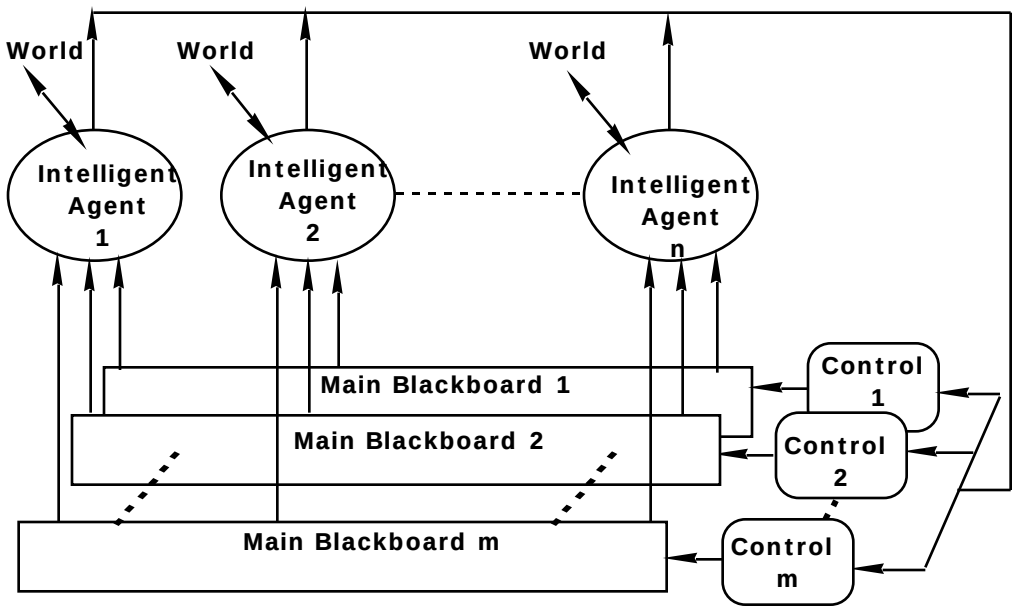


Figure 7. Using Multiple Blackboard

The prototype system is currently running on a single computer system using multiprocessing facilities. By using networking, it is possible to extend the system functionality to support distributed environments. Hence, intelligent agents can be geographically distributed but able to communicate via the system main blackboard.

Since the current implementation of the DYNACLIPS only uses CLIPS shell to represent an intelligent agent, we have chosen to follow the syntax of the CLIPS to represent facts, rules or commands. Knowledge Interface Format (KIF) [15,16] can also be used in the future to transfer facts, rules or commands.

REFERENCES

1. Gilmore, J. F., Roth, S. P. and Tynor S. D. *A Blackboard System for Distributed Problem Solving.* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

2. Nii, H. P. *Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures* , AI Magazine, Summer 1986, pp. 38-53.

3. Nii, H. P. *Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective.* AI Magazine, August 1986, pp. 82-106.

4. Diaz, A. C. and Orchard, R. A. *A Prototype Blackboard Shell using CLIPS.* Fifth International Conference on AI in Engineering, Boston, MA, July 1990.

5. Orchard, R. A. and Diaz, A. C. *BB_CLIPS: Blackboard Extensions to CLIPS,* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

6. Dodhiawala, R. T., Sridharan, N. S. and Pickering C. *A Real-Time Blackboard Architecture,* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

7. Golstein, G. *PRAIS: Distributed, Real-Time Knowledge_Based Systems Made Eassy.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

8. Myers, L. Johnson, C and Johnson, D. *MARBLE: A Systems for Executing Expert System in Parallel.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

9. Schultz, R. D. and Stobie, I. C. *Building Distributed Rule-Based Systems Using the AI Bus.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

10. Gallagher K. Q. and Corkill, D. D. *Performance Aspects of GBB,* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

11. Blackboard Technology Group, inc. *The Blackboard Problem Solving Approach,* AI Review, Summer 1991, pp. 37-32

12. Gilmore, J. F., Roth, S. P. and Tynor S. D. *A Blackboard System for Distributed Problem Solving.* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

13. Rice, J., Aiello, N., and Nii, H. P. *See How They Run... The Architecture and Performance of Two Concurrent Blackboard Systems.* Blackboard Systems. Addison Wesley, Reading, Mass, 1988.

14. Erman, L. D., Hayes-Roth, F., Lesser, R. V. and Reddy, D. R. *The Hearsay-II Speech-Understanding System : Integrating Knowledge to Resolve Uncertanity.* Blackboard Systems. Addison Wesley, Reading, Mass, 1988.

15. Genesereth, M.R., Fikes, R. E. *Knowledge Interchange Format Reference Menual,* Stanford University Logic Group, 1992.

16. Genesereth, M.R., Ketchpel, S. P., *Software Agents,* ACM Communication, July 1994, pp. 48-53.

17. Cengeloglu, Y., Sidani, T. and Sidani, A. *Inter/Intra Communication in Intelligent Simulation and Training Systems (ISTS).* 14th Conference on Computers and Industrial Engineering, Cocoa Beach,March 1992.

18. CLIPS Version 5.1 User's Guide, NASA Lyndon B. Johnson Space Center, Software Technology Branch, Houston, TX, 1991