NASA Conference Publication 10049

First CLIPS Conference Proceedings

Volume II

FIN.

Proceedings of a conference hosted by
the NASA-Lyndon B. Johnson Space Center and held at
the NASA-Johnson Space Center
Houston, Texas
August 13-15, 1990

(NASA-CP-10049-Vol-2) FIRST CLIPS
CONFERENCE PROCEEDINGS, VOLUME 2
(NASA. Johnson Space Center) 542 p

N96-12913
--THRU--
N96-12961
Unclas

G3/63 0064125

NASA

NASA Conference Publication 10049

First CLIPS Conference Proceedings

Volume II

Proceedings of a conference hosted by
the NASA-Lyndon B. Johnson Space Center and held at
the NASA-Johnson Space Center
Houston, Texas
August 13-15, 1990

NASA

CONTENTS

Section Page

A7 SESSION: QUALITY CONTROL APPLICATIONS

B7 SESSION: INTELLIGENT DATA BASES AND NETWORKS

A8 SESSION: SPACE STATION FREEDOM APPLICATIONS

B8 SESSION: USER INTERFACE I

A9 SESSION: SPACE SHUTTLE AND SATELLITE APPLICATIONS

B9 SESSION: USER INTERFACE II

A10 SESSION: ARTIFICIAL NEURAL SYSTEMS AND FUZZY LOGIC

Section Page

B10 SESSION: ENHANCEMENTS TO CLIPS – REASONING/REPRESENTATION

A11 SESSION: PARALLEL AND DISTRIBUTED PROCESSING I

B11 SESSION: ENHANCEMENTS TO CLIPS – OBJECT ORIENTED

A12 SESSION: PARALLEL AND DISTRIBUTED PROCESSING II

B12 SESSION: ENHANCEMENTS TO CLIPS – GRAPHICS/X-WINDOWS

A13 SESSION: AEROSPACE APPLICATIONS

B13 SESSION: ADVISORY SYSTEMS I

Section Page

A14 SESSION: SIMULATION AND DEFENSE

B14 SESSION: ADVISORY SYSTEMS AND TUTORS

A15 SESSION: INTELLIGENT CONTROL

omit

A7 Session:
Quality Control Applications

The Table of Distribution and Allowances (TDA) System Analyzer

Major John F. Mack

Artificial Intelligence Center
US Army Training and Doctrine Command
Fort Monroe, Virginia 23651

Abstract

TDA documents determine the personnel strengths for each Army installation. They reflect the number of people required to accomplish a certain mission by various characteristics. US Army Training and Doctrine Command (TRADOC) analysts continuously scrutinize these documents to ensure that they comply with provided guidance. Part of this guidance has been used to develop a set of manual rules. Analysts apply these rules to the TDA to: (1) eliminate positions; (2) downgrade positions; or (3) reduce position strength. However, this process is very time consuming. In addition, human involvement introduces inconsistencies and errors that are difficult to detect later.

This paper explains how I represented these rules using the 'C' Language Production System (CLIPS) to develop an expert system that is applied consistently and comprehensively for all TRADOC installations. The TDA System Analyzer reduces the review process from about five days to just twenty minutes; giving the user more time to analyze the results and thereby make better decisions. Furthermore, the user is assured that the rules are applied uniformly to every TDA document.

This paper also explains the integration of the TDA System Analyzer into TRADOC's On-Line TDA System. Providing the analyst an extra utility module that can be accessed from a familiar environment.

1. Introduction

Installations rarely receive the exact number of soldiers that they request. Therefore, distributing scarce personnel resources is a problem. It will become more acute as the Army reduces its current 764,000 active-duty force by 184,000 soldiers in the next four years. TRADOC, being the Army's major headquarters for training, is responsible for distributing its share of personnel resources to its subordinate installations.

This process is dynamic and complex. It involves manually reviewing large TDA documents (some having more than 10,000 records) for conflicts with policy, inaccurate grading structures, and incorrect number of soldiers filling a position. The manpower analyst must be familiar with a variety of current and new force structures, unit modernization options, and manpower relationships between units and activities. At a high level, the analyst must be able to formulate plans to distribute new personnel resources and redistribute existing personnel. At a low level, the analyst must track very detailed information to determine the implications on individual units while abiding with current policy.

The sheer size of the TDA documents often forces the analyst to spend an inordinate amount of time reviewing the documents for inconsistencies rather than analyzing them for policy compliance and distribution requirements. In addition, the review process is mundane and subject to error. These errors can adversely affect later analysis.

The purpose of the TDA System Analyzer is to conduct the initial review of the TDA document for the analyst. It scrubs each document using a dynamic rule set written in CLIPS and highlights potential inconsistencies. The analyst concentrates only on these discrepancies; devoting more time to high level analysis.

The TDA System Analyzer executes on a Zenith 248 Personal Computer (PC) with 640 kilobytes of internal memory and 20 megabytes of external hard disk space. The system is an external utility module within TRADOCs On-Line TDA System. The On-Line TDA System is a dBase III™ program that brings TDA databases residing on mainframes to manpower analysts using PCs.

2. The Rules

In 1988, the Commanding General of TRADOC, General Maxwell Thurman, initiated development of a rule set to quantify the discrepancies an analyst should detect while reviewing a TDA document. He intended that an expert system use these rules to relieve the analyst of the initial review process.

The set contains two types of rules: (1) those defining exact discrepancies in the document; and (2) those showing grading structures by position. The first type describes the conditions within the TDA document the system searches. For example, an officer cannot work for another officer of the same grade. If the situation exists where a Captain works for another Captain, the system detects this and reports a problem to the analyst.

The second type specifies the grade or rank for a position at a certain level. For instance, a rule might state that a Company Commander be a Captain. In this example, the position is Commander, the level is Company and the grade is Captain. If the system detects a Major as a Company Commander, a discrepancy is sent to the analyst. Appendix A provides a complete listing of the rules.

3. System Components and Design Methodology

I used a phased control methodology as the basic design for the TDA System Analyzer. Phase control facts are asserted and retracted depending on the current state of processing. Figure 1 shows the sequence of phases.

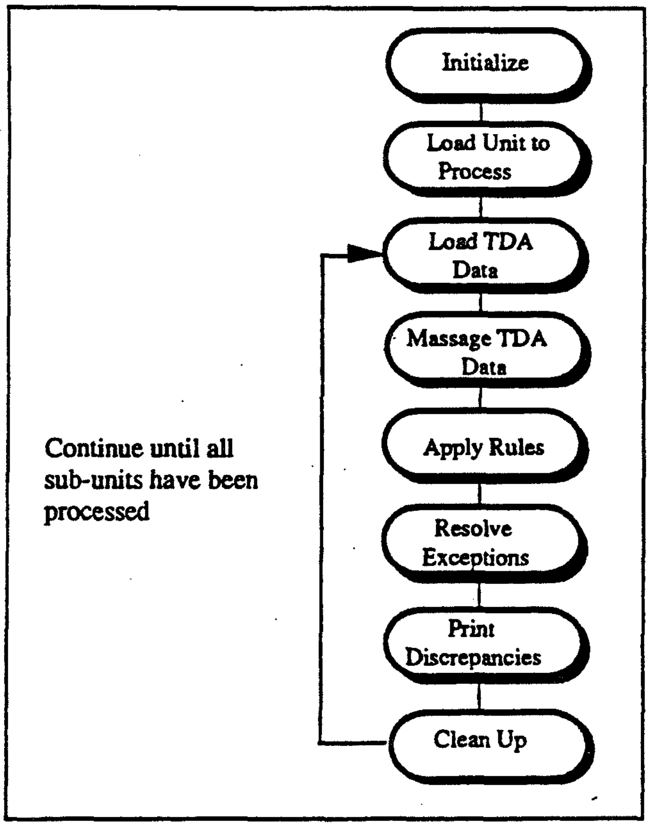


Figure 1. Sequence of Phases

3.1 Initialize

My recompiled version of CLIPS receives three parameters – the file name of the CLIPS rule set, the file name of a dBase III™ database containing position grading data (this will be discussed later), and the name of a TDA document. The system uses the first parameter to load the rule set. It uses the next two parameters to assert two relations. These are **(database "dBase III™ file name")** and **(process "file name of TDA document")**. The first indicates the database that queries will be made to. The second fact tells the system which TDA document to open for processing.

3.2 Load Unit to Process

This phase determines the type of installation that will be analyzed. Different TRADOC installations require different types of analysis as reflected in the corresponding type 2 rules. For instance, the system processes a Service School differently than a Brigade.

A dBase III™ database captures information about the processing requirements for the different installations. The information stored in this database is the triplet (level, position, grade). Type 2 rules use this triplet to detect discrepancies. This data can be thought of as system parameters that can be deleted, modified, or added by the user.

When the system determines the installation type, it queries the database for just the data it needs to review that particular installation. An external function, ssql (Small Structured Query Language), executes this. This function provides direct access to dBase III™ data files using a subset of the Structured Query Language (SQL). For instance, the following rule uses ssql to query the **Grading-Info** database for Service School data.

```
(defrule get-level-service-school
  (phase load-UIC-to-process)
  (service-schools $?service-schools)
      ;;; Service School IDs
  (UIC ?uic&:(> (member ?uic $?service-schools)))
      ;;; is this a service school
=>
  (ssql "grading-data"
        "select * from Grading-Info where
              level = service-school"))
```

Figure 2. Rule Calling ssql

ssql asserts relations of the following type into the knowledge base: **(grading-data "level" "position" "grade")**. These facts represent the permissible grading structures for this installation type.

3.3 Load TDA Data

TDA documents can contain more than 10,000 records. It is impractical to reason about all 10,000 records concurrently. Therefore, the system loads records in small segments based on sub-unit designation and asserts any necessary relations to do reasoning between the different sub-units.

3.4 Massage TDA Data

Several rules require cumulative figures (total number of officers, total number of personnel in this sub-unit, total number of officers by speciality) to determine possible discrepancies. The TDA document does not store this information explicitly. This phase gathers this information and asserts it in the knowledge base for use by the rules in the next phase.

3.5 Apply Rules

During this phase, the system applies the two rule types to detect discrepancies. Initially, I coded a separate rule for each listed at Appendix A. However, the commonality between type 2 rules permitted me to replace these rules with just one. This single rule detects discrepancies by first matching on the **grading-data** relation and then matching on any TDA records that have the same level and position but different grade. (See Figure 3)

```
(defrule grading-rule
  (phase actions)
  (grading-data ?level ?position ?grade)
  (TDA
    (LINE ?line)
    (PARA ?para)
    (SUB-PARA ?sub-para)
    (GRADE ?TDAgrade&~?grade)
    (DESCRIPTION ?desc&:(&&
      (>= (str-index ?level ?desc) 1)
      (>= (str-index ?position ?uesc) 1))))
=>
  (assert
    (discrepancy ?para ?sub-para ?line
                    ?TDAgrade ?desc ?position ?level)))
```

Figure 3. Example of a Type 2 Rule

3.6 Resolve Exceptions

It is possible with type 2 rules to have different grades for the same position and level. For example, a Major or Captain (grade) may be an Action Officer (position) at the Service School level. If one of these cases occurs in the TDA document, the system reports a discrepancy. Yet, either grade is satisfactory for this position. This phase eliminates these discrepancies before reporting them to the analyst. The TDA System Analyzer accomplishes this by first matching on the discrepancy and then searching the knowledge base for a **grading-data** fact that matches the level, position, and grade of the discrepancy. If such a fact exists, the discrepancy is removed. Other exceptions are handled during this phase, but most are the type mentioned above.

3.7 Print Discrepancies

The knowledge base contains only valid discrepancies at this point. The system writes these to an output file that can be later reviewed and manipulated by the analyst.

3.8 Clean Up

During the data massage phase, the system asserted a number of relations containing cumulative figures. This information is only valid while processing the current sub-unit. This data must be removed before the TDA System Analyzer can review the next sub-unit. This phase retracts these facts.

The program loops through phases three to eight until all sub-units of the TDA document are processed. The system then closes the TDA document file and discrepancy output file.

4. Performing a TDA Document Review

We integrated the TDA System Analyzer in the On-Line TDA System as a utility module. The analyst uses the On-Line TDA System daily and is comfortable with its structure and user interface. These characteristics make using the TDA System Analyzer easier.

The analyst first calls the On-Line TDA System's utility module. He then chooses the TDA System Analyzer. At this point, the user can elect to process the document with the current parameters or he can access the dBase III™ database to change them. The ability to change system parameters so readily permits him to alter dynamically the way the system will review a document. In addition, the analyst can do "what-if" exercises. For example, he may wonder what discrepancies will be found if he changes Battalion Commanders at the Service School level from Lieutenant Colonels to Majors. The analyst does this easily using the appropriate dBase III™ commands.

When the analyst is satisfied with the system parameters, he chooses the specific TDA document to process and tells the system to execute. A review of 10,000 records requires approximately 25 minutes. This performance compares with a typical 40 man-hour analysis – an improvement of almost 100:1. In addition, the review is complete and consistent. After execution, the analyst can browse the discrepancy output on-line or print it to hard-copy for future reference.

The On-Line TDA System also supports batch processing of the TDA System Analyzer. Analyses can be run during non-duty hours on many TDA documents and/or on the same document with different parameter settings.

5. Summary

The TDA System Analyzer represents an innovative way of analyzing TDA documents. It gives the manpower analyst the power to change dynamically the way a document will be processed, but isolates him from the mundane task of actually doing the review. He can focus more on issues, policy, and personnel distribution problems.

The power and flexibility of the CLIPS' environment supported the rapid development of a system that could be iteratively refined. I was able to implement quickly improvements and changes. The power of this environment permitted me to develop a complete system with extensions to the basic functionality of CLIPS in less than three months.

The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

Appendix A
TDA System Analyzer Rules

I. Type 1 Rules

a. Officers will not work for other officers of the same grade.

b. There will be no deputies or assistants except at general officer commanded installations.

c. All ROTC military schools (e.g., Citadel, Norwich) will be allowed one Major (MAJ) as Commandant of Cadets.

d. Support an additional ROTC Captain (CPT) position at historically black colleges.

e. No more than 30% of total officers will be field grade in any element at the Service School level.

f. Support only 75% of CAS^3 instructors in grade of Lieutenant Colonel (LTC).

g. Where there is more than one position in a single job title/specialty, indicate that 50% of these may be downgraded.

h. Support only one LTC in the office of a TRADOC System Manager.

II. Type 2 Rules - indicate the appropriate grade for a particular position and level.

LEVEL	POSITION	GRADE
Installation	Commander	Major General
Installation	Commander	Brigadier General
Installation	Deputy Commander	Brigadier General
Installation	Deputy Commander	Colonel (COL)
Installation	Chief of Staff	COL
Installation	Resource Manager Director	LTC
Installation	Engineering & Housing Director	LTC
Installation	Inspector General	LTC
Installation	Airfield Commander	MAJ
Installation	S-1	(Null)[1]
Installation	S-2	(Null)
Installation	S-3	(Null)
Installation	S-4	(Null)

[1] A null value indicates that this position is not valid at any grade for this level.

II. Type 2 Rules (continued)

LEVEL	POSITION	GRADE
Army Training Center (ATC)	Commander	Brigadier General
ATC	Deputy Commander	(Null)
ATC	Chief of Staff	(Null)
ATC	Resource Manager Director	(Null)
ATC	Engineering & Housing Director	(Null)
ATC	Inspector General	(Null)
ATC	Airfield Commander	(Null)
ATC	S-1	CPT
ATC	S-2	MAJ
ATC	S-3	MAJ
ATC	S-4	CPT
Brigade	Commander	COL
Brigade	Deputy Commander	(Null)
Brigade	Executive Officer	LTC
Brigade	S-1	CPT
Brigade	S-2	CPT
Brigade	S-3	CPT
Brigade	S-4	CPT
Battalion	Commander	COL
Battalion	Deputy Commander	(Null)
Battalion	Executive Officer	MAJ
Battalion	S-1	CPT
Battalion	S-2	CPT
Battalion	S-3	CPT
Battalion	S-4	(Null)
Service School	Liaison Officer	MAJ
Service School	Liaison Officer	CPT
Service School	Action Officer	MAJ
Service School	Action Officer	CPT
Service School	System Manager	COL
Service School	Threat Manager	MAJ
Service School	Communication Skill Officer	MAJ
Service School	Proponency Officer Chief	LTC
Service School	Department Director	COL
Service School	Department Director	LTC
ROTC	Professor of Military Science	LTC
ROTC	Enrollment Team Officer	CPT

[2] LTC will be a Department Director if there are less than 65 people in the department.

52-63

N96-12915

p. 8 358461

MOM: A Meteorological Data Checking Expert System in CLIPS

Richard O'Donnell
Geophysics Laboratory
Hanscom AFB, MA

ABSTRACT

Meteorologists have long faced the problem of verifying the data they use. Experience shows that there is a sizable number of errors in the data reported by meteorological observers. This is unacceptable for computer forecast models, which depend on accurate data for accurate results. Most errors that occur in meteorological data are obvious to the meteorologist, but time constraints prevent hand-checking. For this reason, it is necessary to have a "front end" to the computer model to ensure the accuracy of input. Various approaches to automatic data quality control have been developed by several groups.

MOM is a rule-based system implemented in CLIPS and utilizing "consistency checks" and "range checks". The system is generic in the sense that it "knows" some meteorological principles, regardless of specific station characteristics. Specific constraints kept as CLIPS facts in a seperate file provide for system flexibility. Preliminary results show that the expert system has detected some inconsistencies not noticed by a local expert.

I. Introduction

Large amounts of meteorological data must be processed in order to study and forecast our weather. The accuracy and utility of forecasting models and techniques depend heavily on the accuracy of the input data.

At the Geophysics Laboratory, Hanscom Air Force Base, in Bedford, Massachusetts, there is a meteorological data collection facility called AIMS (Air Force Interactive Meteorological System). AIMS is a VAXcluster with many sources of automated continual data input, including the FAA 604 line, and a GOES ground station, which supplies satellite imagery and data. (FAA stands for the Federal Aviation

Administration, which oversees many flight related forecasting operations. GOES is an acronym for Geostationary Operational Environmental Satellite. The GOES ground station supplies satellite imagery for the Western Hemisphere every 30 minutes.) The FAA 604 line includes Service-A data (hourly data within North America), synoptic data (data from worldwide sources every six hours), radar data, forecasting model results, and other types of data.

The purpose of this facility is to develop new techniques to study and forecast atmospheric behavior. The forecasting models being developed use these sources of data as input to generate a forecast. It is obvious that this data needs to be accurate in order for these models to provide accurate output. This is the problem of meteorological data validation. One purpose of our study was to determine exactly how frequently inaccurate observations are reported.

II. Meteorological Data Validation

The errors in meteorological data are produced by two sources: human error, and machine error. Human errors could entail a misreading of an instrument, a mismeasurement, or even a "typo", while machine errors include malfunction, breaking of equipment, and even noise in the data lines. All of these factors combine to cause data available to the scientists and the computer models to be in error.

The Meteorological Observation Monitor (MOM) is an attempt to weed out errors in the database by identifying errors that are found. MOM is written in CLIPS and is still in the process of being tested and further developed.

MOM is a system made up of four basic parts: a main knowledge base of CLIPS rules, a base of specific meteorological facts, a module which extracts the data from the database and puts the data into the form of CLIPS facts, and, of course, the database itself. The main knowledge base and the meteorological fact base are the components to be studied, since they represent the expert system part of MOM, and are the parts written in CLIPS.

In order that MOM be made more flexible and expandable, as well as maintainable, only general priciples were included in the main knowledge base, and specific meteorological information was left out. The specific data needed to make decisions was included in the fact base. For example, the main rule base contains the general information that there is a minimum air temperature at which rain may occur. The specific temperature that will be used to determine whether the type of precipitation is correct resides in the fact base. This modular design lends itself well to maintenance, especially since data is sometimes invalid because it does not conform to reporting conventions, and these conventions can change. For example, wind gusts speed may not be reported unless the gusts of wind are at least 10 knots greater than the low wind speed for the hour. This convention has changed through the years, and it is possible it will change again. Updating this type of information would require only minimal maintenance to MOM, since only the smaller fact base would need to be changed.

Before getting any further into the design of MOM, it would be best to discuss what specific problems arise with meteorological data, and several methods to validate data. It may seem obvious, but meteorological data is invalid whenever it does not accurately represent the real world. Choosing which of these data are accurate, and which are not, is not always possible. In many cases, however, situations arise which clearly show the existence of invalid data. For example, the temperature at Logan Airport in Boston may truly be 64°F, but is being reported as 69°F. To the scientist sitting in the lab in Bedford, 69°F seems well within the realm of possibility, and that data will never be found to be invalid. This is not catastrophic, because if this kind of invalid data goes unnoticed, it is not very disruptive to the computer models that produce forecasts. However, there are times when the scientist in the lab may know for certain that the data is invalid, if Logan is reporting 75°F in January, for example.

There are in principal two reasons data can be invalid. First of all, it can break physical laws of nature. Rain is highly improbable when it is 5°F. The other reason reported data can be invalid is that it can break conventions, such as the wind gusts convention mentioned earlier. While it may be true that winds are from the north at 6 knots with gusts up to 9 knots, to report that is not helpful, and would cause others to question the validity of the data, since it is not possible for the difference from lull to peak wind to be 10 knots. There are several such conventions, and we will see some of these later.

There are at least four different methods one may use to successfully recognize invalid data. The first method, and the one most often used by a human meteoroligist scanning the weather maps, is "buddy" checks: that is, checking the nearest neighbors of the station that is reporting the data to validate it. If Boston is reporting 14°F, and Bedford 50°F, there is an enormous discrepancy to account for. The second method is to do a time check. If New York's Kennedy Airport is reporting a temperature that is in question, a time check would look at the most recent reports of temperature at Kennedy and compare them to the data in question. The third method, and the primary method employed by MOM, is to do a consistency check. A consistency check takes an hourly report consisting of several observed parameters, and determines whether the relationships between the parameters are consistent. For example, if a station reports a temperature of 50°F but also reports snow, there is an inconsistency in the report. A fourth method of validation is to do a range check. A range check takes a single data item and determines whether it falls within climatological extremes for the reporting station and month. A primitive range checker is also included in MOM. A complete system would use all of these methods to best validate data.

There are problems with each of these methods. Some of these problems are meteorological and some are computational. The buddy checking method has a problem in that each station would need to have buddies, and not all have near neighbors. Not only is that the case, but sometimes, because of geographical elements, a nearby neighbor would not be as good a choice as a further neighbor. Therefore, a table of neighbors would need to be created so that only those neighbors which would contribute similar data would be consulted. Time checking also has problems, primarily meteorological. In many places, drastic changes in temperature can take place within an hours time, which is the normal reporting interval. These drastic changes may be extremely improbable elsewhere. Self-consistency checks have the problem of being too limited. The data may not disagree; however, that does not necessarily indicate that there are no errors present. Range checking is similar; if data is flagged for being out of a reasonable range, it is a good bet it is invalid, but alot of invalid data meets the requirements of that test, and therefore is not discovered. Any one method alone will not discover all errors present in the data.

III. MOM and Validation of Data: Consistency and Range Checks

When the problem of data validation was first considered in this study, it was decided that MOM would represent a first attempt to address the concern. The data chosen to be validated was Service-A data, and MOM was to employ consistency checks and range checks on this data. The reason consistency checks were selected was that, of all the methods described, it lends itself most handily to a "rules" oriented knowledge base.

Service-A data is hourly data reported from all stations in North America. MOM examines nine parameters in a report for self-consistency: air temperature, dewpoint temperature, pressure, altimeter setting, wind speed, wind gust speed, wind direction, visibility, and current weather. Pressure is not reported from a number of smaller airfields, and instead, these stations only report an altimeter setting. Except in one case, each of these parameters is a floating point number which is defined by a specific range of possibilities. For example, the range of wind direction is 0.0 to 360.0. The exception is current weather. This is defined by a string of characters, each representing a different weather pattern or phenomenon. If nothing is currently happening at a particular station, the current weather string is empty. Examples of current weather are fog and rain. A complete list of possibilities is given in Table 1.

Current Weather		
Obstructions to Vision	Weather Symbols	Intensity Symbols
F = Fog	T = Thunderstorm	+ = heavy
GF = Ground Fog	L = Drizzle	- = light
IF = Ice Fog	ZL = Freezing Drizzle	W = showers
D = Dust	R = Rain	
K = Smoke	ZR = Freezing Rain	no modifier indicates moderate intensity
H = Haze	S = Snow	
BD = Blowing Dust	SP = Snow Pellets	
BN = Blowing Sand	SG = Snow Grains	A and IC have no intensity symbols
BS = Blowing Snow	IP = Ice Pellets	
BY = Blowing Spray	IC = Ice Crystals	
	A = Hail	T may only have +

Table 1: Reporting codes for current weather

The representative letters in Table 1 can be combined in many ways, with precipitation types coming first, and obstructions to vision last, to describe the wide variety of possible weather conditions. The intensity symbols are modifiers that add to the meaning of the character preceding them. For example, the string "RF" means the reporting station is experiencing both rain and fog, while "R-F" means the station is experiencing light rain and fog. Intensity symbols are not used with obstructions to vision. These strings can be arbitrarily long to describe very mixed kinds of weather, like the weather we get in New England. On an unusually bleak winter day, a report could be "ZL-ZR-S-F" which means a mix of light freezing drizzle, light freezing rain, light snow, and fog. "TRW" means thunderstorms and rain showers. A problem with this system is that strings can be ambiguous. For example, the string "SGF" could mean either snow with ground fog, or snow grains with fog.

The nine data items discussed have many different interrelations that force a large number of rules governing consistency checking between the parameters. Table 2 shows which parameters are closely related.

Parameter	Related Parameter
temperature	dewpoint temperature
temperature	current weather
dewpoint depression*	current weather
visibility	current weather
wind speed	wind gusts speed
wind speed	current weather
pressure	current weather
pressure	altimeter setting
altimeter setting	current weather

* dewpoint depression is temperature minus dewpoint temperature

Table 2: Reported parameters which have relationships to each other

As you can see, current weather is the most commonly related parameter. Current weather is related to almost all the other parameters, and, although there are only nine distinct relationships shown in the above table, the variety in current weather forces a large number of rules. For example, the visibility relationship with current weather is just one relationship listed above. There are a large number of rules required to describe this relationship, however. For virtually every obstruction to vision and precipitation type and intensity, a rule must be created to identify the lower and upper bounds of visibility possible under the circumstances.

IV. Preliminary Results

Preliminary results show that 1 out of every 100 incoming data sets are prone to error. These results are based on close to 1200 reports that have been examined by MOM. This is a result achieved only with consistency checks. A system incorporating time and buddy checks will find many more errors. On days with mixed weather, the number of errors has been as high as 1 in 60 data items. Again, however, these results are preliminary, because most of the testing period has taken place during periods in which little or no current weather has been reported. Testing is still in process, and will continue for some time.

The majority of the errors found thus far have been reports that do not abide by conventions. A common error is reporting of wind gusts which are less than 10 knots. Another common "convention-breaking" error is a report of less than 7 miles visibility without a corresponding report of an obstruction to vision. The convention states that if a visibility less than seven miles is to be reported, an accompanying obstruction to vision must be reported.

Table 3 is an example of input to MOM. The table is a copy of a file which is read in CLIPS and processed.

```
(data station-id WORCESTER)
(data time z30-JAN-1990:11:00)
(data airtemp 86)
(data wind-dir 20)
(data wind-speed 15)
(data visibility 2)
(data current-weather freezing-rain fog)
(data precip-intensity light-freezing-rain)
```

Table 3: Sample input to MOM

The results of processing the input from Table 3 are seen in the output from MOM in Table 4.

```
CLIPS> (run)

 *** DATA FOR WORCESTER AT z30-JAN-1990:11:00 ***
    airtemp 86
    dewpt MISSING
    pressure MISSING
    altimeter MISSING
    wind-speed 15
    wind-gust MISSING
    wind-dir 20
    visibility 2
    current-weather freezing-rain fog

 *INCONSISTENT*   AIRTEMP CURRENT-WEATHER
 current weather reports freezing rain at a temper-
 ature greater than which it is likely to occur
 (max temperature for freezing drizzle is 39)

13 rules fired
Run time is 0.3203125 seconds
CLIPS>
```

Table 4: Sample output of MOM corresponding to input from Table 3

V. Future Paths of MOM

MOM is not a completed effort. Future work on MOM will be based on the outcome of testing. If work does continue on the system, there are at least four areas which require further study. First, MOM should have a more complete range checking subsystem. The current range checking in use is primitive, and does not take into account individual station characteristics, or seasonality. Second, MOM should be expanded by adding buddy checking and time checking methods of validation. These features would allow MOM to be more functional, and help to find more errors. Third, MOM should be delivered out of the test environment and into the working environment. Currently MOM is still running in CLIPS interactively, and testing has been taking place using batch files. A delivery environment for MOM would mean better run time, and a capacity to test more data. Finally, and most ambitiously, an error correction facility could be implemented.

Automated Decision Stations

Mark Tischendorf
Eastman Kodak Company
Artificial Intelligence Laboratory
(716) 477-1357

March 15, 1990

Automated Decision Stations

Abstract

This paper discusses the combination of software robots and expert systems to automate everyday business tasks. Tasks which require people to repetitively interact with multiple systems screens as well as multiple systems.

1. Objective

This paper describes a system created to automate decisions. Either an independent system capable of performing specific business tasks or an intelligent assistant which helps individuals by collecting information, offering recommendations, and carrying out decisions.

The term "Information System" implies an end goal of providing a person with information. The person is responsible for deciding what the information means. Our intent is to integrate the system's information with automated human decision making without altering the existing information systems.

For many potential applications, the business case compares system implementation costs with the cost of clerical labor. Therefore, implementation costs need to be as small as possible.

2. Introduction

Information systems are an integral part of many business operational environments. These systems normally provide users with information about a single domain. This results in individuals being assigned to act as interfaces between such systems. For specific tasks, a person needs to gather information by referencing multiple systems or screens, decide what to do with the information, then carry out the decision within one or more systems. These types of jobs exist because building interfaces between the existing information systems is not possible or cost effective.

Software robots combined with expert systems can emulate the type of human activity described above.

A software robot is a programming tool for automating the use of existing software. Software robot tools are also called surround tools, agents, or script files. Software robots can emulate keystrokes and monitor screen activity. In most cases software robots can automate any repetitive task that a person performs at a terminal.

Occasionally a software robot application requires significant reasoning capabilities in order to make decisions. This is where expert systems come into play. The software robot collects pertinent information and feeds it to the expert system. The expert system performs its reasoning and either tells the software robot what to do or makes a suggestion to an individual. In either case, the software robot can then carry out the decision.

The software robot acts as "the eyes and hands." It knows how to traverse systems and screens, and where to locate data on screens. The expert system, on the other hand, acts as "the brains." It reasons about information provided by the software robot. Although I've called this type of system an Automated Decision Station, one could also call this an Expert Software Robot.

3. Implementation

The hardware involved is a personal computer equipped with multiple session 3270 emulation capability. The 3270 emulation software, robot software, and expert system software need to be simultaneously resident in the PC's 640K memory. Therefore, memory restrictions are a primary constraint on software tool selection. The PC need not be a high-speed 386-based machine. Host system response time is the limiting factor on software robot execution speed. A 286-based PC works well and is sometimes easily obtainable since they're somewhat out of date.

The software robot tool we've used is AUTOMATOR-MI from Direct Technology. AUTOMATOR is capable of surrounding software on the PC as well as any type of host system. AUTOMATOR also works over a wide variety of connectivity alternatives. The current version of AUTOMATOR uses about 96K of memory.

The expert system shell CLIPS, from COSMIC/NASA, handles the decision reasoning. CLIPS is desirable due to its low memory overhead, low cost, and ability to import data files.

The robot controls execution of the other software and interfaces with the user if necessary. The robot accesses one or more systems by way of the 3270 emulation. Concurrent access to multiple systems is accomplished over separate emulation sessions. The robot collects pertinent data from these systems and creates a file on the PC containing this information. The robot then jumps from 3270 emulation into DOS where the expert system is already running but suspended. The robot starts up the expert system, which reads the pertinent data file and reasons about a decision. The expert system displays the decision on the screen so the robot can see it, then suspends itself. The robot reads the decision and jumps back into 3270 emulation to carry out the decision. Note that the robot is controlling all the activity. The robot treats the expert system as a decision-making calculator.

4. Example

We've built an Automated Decision Station to assist order entry credit checking in one of Kodak's distribution regions. A two-to three-week programming effort has yielded a system which can automatically handle about 20% of the credit referral activity. More significantly, the automated data collection considerably aids manual processing of the remaining credit referrals.

As described above, the robot collects data for each credit referral from two different mainframe systems. The pertinent information is summarized from six or more different screens. The expert system identifies the type of credit referral and performs any appropriate calculations. A printout communicates the pertinent information, recommended action, and reasoning explanation. If the referral looks okay, the system will approve it, given user confirmation. Otherwise, the system places the referral on hold for manual handling.

This application has been in use since the beginning of February 1990. As of this writing, several thousand transactions have yielded no significant problems. Future enhancements are identified to provide additional automation capability.

This Automated Decision Station offers management the opportunity to combine manual credit referral operations. We can concentrate activity from several regions into two regions (east-coast and west-coast). Alternatively, a Decision Station can be put in each region to streamline each existing process.

5. Costs / Benefits

The primary cost of this Credit Referral Expert Software Robot was the two- to three-week programming effort. A spare PC was resurrected from a storage shelf. A software robot run-time license cost $250, and a 3270 emulator board cost $750.

The benefits include:

Reduced labor, increased productivity, and faster workload turnaround from automating repetitive terminal activity. These systems sometimes cut out the need for users to interface with any systems.

Better job quality due to the absence of typing errors.

Users of such systems gain an increased sense of self-worth. Rather than spending time keying and calculating, they are now free to concentrate on the highly skilled parts of their job.

These inexpensive decision stations offer a new way of accomplishing things. They are an alternative means of interfacing information between systems for specific business purposes. These systems are a cost-effective way to do things previously considered unjustifiable.

By capturing the rules on how to make specific business decisions, we are preserving corporate know-how. We then apply this know-how consistently to suitable problems. This know-how can also help with the training of new people.

6. Observations

This type of system is easy to introduce into new environments. The low cost certainly helps, but the implementation methodology also plays a big role. The software robot needs to surround the existing work environment. So the system is typically build right in the end-user work place. This results in close contact with both the users and their management. They see the system evolve as it is built, fostering a sense of ownership.

These Expert Software Robots lend themselves to modular implementation. It is often possible to build only the robot component, keeping the person in the loop for decision making. Then, build the expert system component when resources become available. It is helpful to use the robot to collect actual test cases, to aid the expert interviewing process.

It turns out that it is very easy to migrate these systems from prototype status into a production-worthy system. It is so easy, in fact, that we make this migration even though programming enhancements are pending. The catch here is that once in production, programming changes have to be more carefully coordinated and are therefore more time consuming.

The one disadvantage to software robots are their vulnerability to host system screen changes. If screens change in the surrounded environment, the robot can get confused. So far this has not been a problem. It has only taken minutes to fix a couple such occurrences. This does, however, imply that a trained person needs to be available to attend to these types of unexpected situations.

7. Conclusions

Automatic Decision Stations (or Expert Software Robots) are easy and inexpensive to build. The learning curve on the software tools is relatively short. These types of applications can increase productivity while improving quality. These systems offer a new way to solve problems, as well as an alternative way to view existing systems environments.

B7 Session:
Intelligent Data Bases and Networks

54-61

358495

p.6

N96- 12917

ISLE: Intelligent Selection of Loop Electronics A CLIPS/C++/INGRES Integrated Application

Lynn Fischer, U S WEST Advanced Technologies
Judson Cary, U S WEST Advanced Technologies
Andrew Currie, Bolder Heuristics

1.0 Abstract

The Intelligent Selection of Loop Electronics (ISLE) system is an integrated knowledge-based system that is used to configure, evaluate, and rank possible network carrier equipment known as Digital Loop Carrier (DLC), which will be used to meet the demands of forecasted telephone services. Determining the best carrier systems and carrier architectures, while minimizing the cost, meeting corporate policies and addressing area service demands, has become a formidable task. Network planners and engineers use the ISLE system to assist them in this task of selecting and configuring the appropriate loop electronics equipment for future telephone services.

The ISLE application is an integrated system consisting of a knowledge base, implemented in CLIPS; a planner application, implemented in C++, and an object database created from existing INGRES database information. The embedibility, performance, and portability of CLIPS provided us with a tool with which to capture, clarify, and refine corporate knowledge and distribute this knowledge within a larger functional system to network planners and engineers throughout U S WEST.

2.0 Overview

The selection of Digital Loop Carrier equipment has a significant impact on network operations and business costs. In today's environment, the network planner faces a growing number of DLC equipment vendors and potential carrier systems. Perhaps more significantly, new carrier architectures (such as dynamic concentration and integration) have been introduced. Each system has different features, functions, capacities, and costs. Determining the best carrier system and architecture, while minimizing the cost, has become a formidable task. The planner must consider all possible choices, screen out incompatible solutions, and then rank the remaining contenders in a manner that optimizes functionality, minimizes cost, and meets corporate objectives. In addition, corporate policies set forth as guidelines for equipment selection need to be included in the planning process.

The ISLE system was developed to bring the appropriate knowledge to the network engineer and support the evaluation of many more DLC equipment options than previously possible. U S WEST network planners and engineers currently use ISLE to assist them in choosing the appropriate DLC electronics equipment and architectures when configuring equipment within a specific geographic area for future telephone services. The ISLE program assures that corporate policies are implemented and provides a thorough analysis of all applicable systems.

The ISLE system is an integrated knowledge-based system which is currently deployed on UNIX workstations in U S WEST Communications. The knowledge-base module of the system was developed using CLIPS, an OPS-like, rule-based language implemented in C. The control module was written in C++ with interfaces to INGRES/SQL databases.

3.0 The ISLE system

The major functionality of the ISLE system is to evaluate telephone service requirements and produce recommendations about DLC equipment which has optimal price and performance to meet those requirements. The ISLE system generates all potential DLCs or sets of various DLCs, evaluates those solutions, ranks the solutions, and determines the housing required for each equipment solution.

ISLE provides planners with the following information:

-A cost summary of all equipment/architectures and housing technically capable of providing the forecasted service for the geographic area.
-A configured parts listing for the recommended DLC system(s).
-Economic comparison graphs for potential DLCs.
-Comments on corporate strategies related to DLC equipment.
-Service capabilities of a given DLC system.
-Engineering information and assumptions used to configure DLCs.

Some of the benefits derived from the ISLE system are:

- improved decision-making in the planning process with the result of more effective and cost-efficient DLC installations

- consistent application of corporate guidelines

- increased productivity for planners

- superior training for planners with an accelerated learning curve for the design process.

By developing a uniform process for carrier design, planners throughout U S WEST benefit from the knowledge of the company's expert engineers coupled with the assimilation of large amounts of data collected over many years and stored in the U S WEST engineering databases. Planners can more thoroughly and quickly evaluate equipment configurations, ultimately arriving at a better solution. The previous manual approach to planning required the planner to complete many computations and search the databases for information. Even after that time-consuming process, the planner still did not have the benefit of the knowledge collected and delivered by ISLE. Delivering that expertise on an on-going basis shortens the learning curve for planners, both new and experienced, by systematically increasing their awareness of new solutions and corporate polices in the planning process.

4.0 ISLE Architecture

ISLE is an integrated system which uses the CLIPS rule-based system, C++, and information from INGRES/SQL databases, all running under a UNIX* operating system. Currently ISLE contains over 200 CLIPS rules, consists of 20 C++ modules, and uses data from two INGRES databases. The major components of the system are shown in Figure 1.

*UNIX is a registered trademark of American Telephone and Telegraph Co.

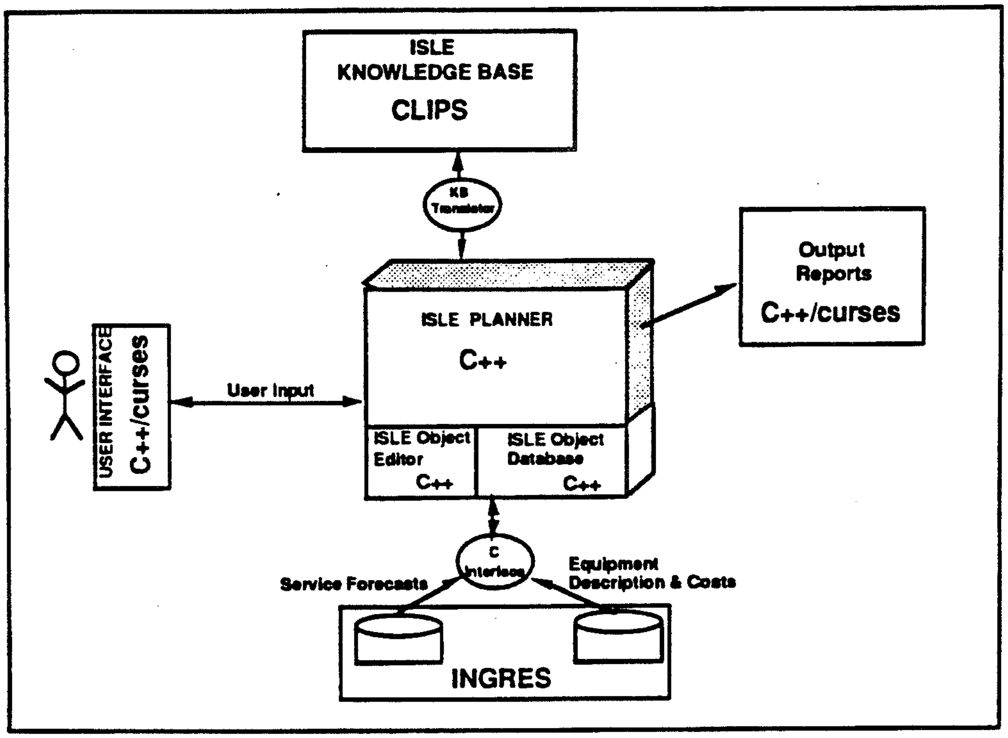


Figure 1. Integrated Components the ISLE system

The high-level program control is maintained by the C++ modules which call CLIPS for rule-based constraint satisfaction and equipment configuration. INGRES/SQL databases provide service forecast information, equipment descriptions, and costs. All DLC equipment, parts, and housings are represented as objects in C++. The user provides additional information concerning characteristics of the geographic area. ISLE then begins generating possible DLC solutions which can meet the forecasted service demand in the area. During the generation/evaluation process, various sets of CLIPS rules are invoked to eliminate or comment on various DLCs. Figure 2 is a high-level data flow which depicts the high degree of integration between the the C++ processes and the CLIPS rules.

ISLE relies on two INGRES databases which contain data related to the various equipment models, (eg. part ids, costs, modes) and telephone service forecasts for various geographic areas. ISLE uses SQL queries to several tables in these databases to retrieve information that is referenced by CLIPS rules. The data from the database query is fetched into a structure, then used to construct new C++ objects. When this information is needed by a ruleset that is about to be run, an assert method is called on one or more of these C++ data objects. This two step process also allows us to translate the data from the format and structure found in the INGRES tables to one more useful for the ISLE systems. In the course of rules firing, the CLIPS rules also generate information that must be returned to the C++ environment. For this task, ISLE has a special interface ruleset whose sole task is to pass back certain types of information (costs, comments, etc.) and create new C++ objects or revise that information in existing C++ objects.

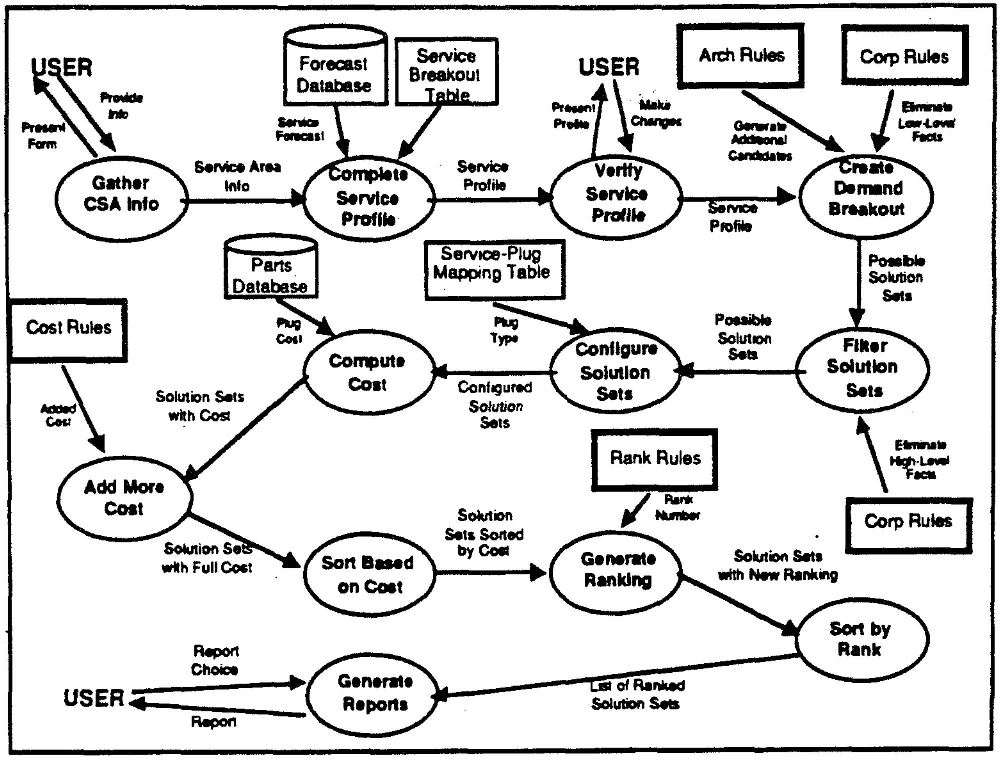


Figure 2. High-level Data Flow of the ISLE System

ISLE also contains some information that is not stored in databases, but is information necessary for the generation of solution sets of DLC equipment which can meet the telephone service forecast. Most of this information is a kind of "engineering expert common knowledge" that is best represented as association lists. An example of this information is the mapping of specific types of telephone services to the appropriate hardware components which support each service.

Most of the information that is required by the rules, but not available in U S WEST databases is obtained from the user through the use of simple menus and forms created using C++ and the curses package in UNIX. Some additional information may be required based on the user input to the ISLE forms. This information is obtained by direct query of the user from the rules themselves via a curses interface function.exported to CLIPS. In all cases where ISLE is using data from the databases, or deriving data in some other way, the user is given the opportunity to inspect and change any data that does not seem appropriate for the analysis. The data change task is performed by using object browsers written in C++ or using forms written in curses.

Currently all CLIPS rules remain in working memory during the entire ISLE session and context facts are used to activate the appropriate group of rules as ISLE performs its analysis. With this approach, CLIPS performance (i.e. speed and memory usage) has been acceptable to date, although some rule optimization has been necessary to work within memory constraints and maintain an appropriate total run time. This is especially true when ISLE is used to analyze geographic areas with more than 1000 service lines. Such a scenario can generate over 500 different types of objects which must be evaluated by the CLIPS rules. The likely introduction of new types of DLC or different architectures could significantly increase the number of objects in CLIPS working memory and dramatically impact the system performance.

5.0 ISLE Knowledge base

The ISLE knowledge base contains knowledge related to corporate policies, equipment limitations, and qualitative costs. The knowledge represented in ISLE is well-suited to representation in a rule structure. The following is a typical ISLE rule along with the CLIPS representation:

```
If      integration architecture is allowable in this carrier serving area
        and the central office switch is a type-a
        and the total number of lines forecasted exceeds 600
        and no special services are forecasted or specials could be groomed to copper,
then    Integration is recommended in this carrier serving area.
```

```
(defrule csa-integ-type-a-600
   (csa-integration possible)
   (csa-info-object ?idcsa co-switch type-a)
   (csa-service-forecast-object ?sfo total-forecast ?forecast-lines&:(> ?forecast-lines 600))
   (or (not (csa-service-forecast-object ?sfo total-specials  ?no))
      (could-groom-specials-to-copper))
=>
     (assert (csa-info-object ?idcsa arch-comment
     "Integration is recommended within this CSA for the following reasons:
       1. A type-a switch can be integrated.
       2. The service forecast is over 600 lines.
       3. The number of specials (non vf-asgn services) is insignificant."))
     (assert (csa-integration recommended)))
```

A wide variety of knowledge sources were required to obtain all the information necessary for the generate/evaluate task. Figure 3 identifies some of the information sources which were used to generate the ISLE knowledge base.

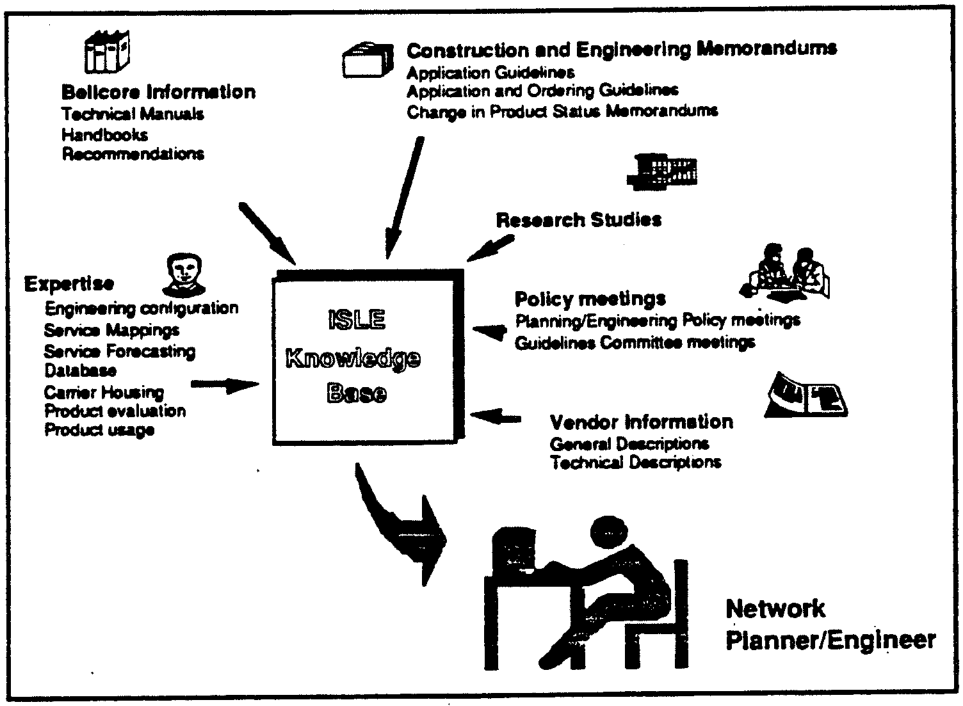


Figure 3. ISLE Knowledge-base Sources

The ISLE knowledge base is divided into five areas which focus on different concerns related to generation and evaluation of DLC equipment and housing. Those knowledge base subject areas are corporate rules, architecture rules, cost rules, housing rules, and rank rules.

CORPORATE RULES provide knowledge about which DLC electronics equipment is the most desirable in terms of future cost-effectiveness and viability. These rules focus on overall corporate strategies and also corporate policies on the use of specific carrier systems. This knowledge will encourage planners to explore unfamiliar solutions in planning for geographic areas and adhere to corporate guidelines.

ARCHITECTURE RULES help configure DLC candidates by providing knowledge about concentration and integration architectures at the individual DLC level, and identify the most appropriate configuration modes for a given DLC. Recommendations for integration architectures, remote switching units, and fiber use are made based on the general characteristics of a geographic service area.

COST RULES provide estimates of costs for factors such as training, installation, the number of T-lines and P-lines, and other expenses. Comments on qualitative costs are also provided.

HOUSING RULES define which type of housing can be used with a specific solution set. This cost is then added to the total cost of the solution set.

RANK RULES re-rank ISLE solution candidates based on factors other than cost, such as corporate strategies or equipment features.

6.0 Summary

In summary, ISLE is a system "that uses human knowledge to attain high levels of performance in solving difficult problems within a narrow problem domain". The U S WEST network planner is aided in the difficult problem of planning for geographic areas by the data and knowledge collected and assimilated within ISLE. The planner becomes more productive and the corporation benefits from higher-quality, lower-cost installations which fulfill corporate strategies and policies.

CLIPS provided an integral component to the overall business solution of the ISLE system. The versatility and portability of CLIPS allowed us to deliver the ISLE system on the user's chosen platform of UNIX. This approach also allowed us to integrate the system with C++ modules and existing INGRES/SQL databases on the delivery machine. In general, we have found that a majority of real-world AI business applications are best delivered as integrated business solutions, rather than stand-alone systems. CLIPS seems to allow for the high portability and integration with external systems necessary for production knowledge-based systems.

S5-61

N96-12918

p. 9

358498

An SQL Query Generator for CLIPS

James Snyder and Laurian Chirica

CAD Research Unit
California Polytechnic State University
San Luis Obispo, Ca.

ABSTRACT

As expert systems become more widely used, their access to large amounts of external information becomes increasingly important. This information exists in several forms such as statistics, tabular data, knowledge gained by experts and large databases of information maintained by companies. Because many expert systems, including CLIPS, do not provide access to this external information, much of the usefulness of expert systems is left untapped. The scope of this paper is to describe a database extension for the CLIPS expert system shell.

The current industry standard database language is SQL. Due to SQL standardization, large amounts of information stored on various computers, potentially at different locations, will be more easily accessible. Expert systems should be able to directly access these existing databases rather than requiring information to be re-entered into the expert system environment. The ORACLE relational database management system (RDBMS) was used to provide a database connection within the CLIPS environment.

To facilitate relational database access, a query generation system was developed as a CLIPS user-function. The queries are entered in a CLIPS-like syntax and are passed to the query generator, which constructs and submits for execution, an SQL query to the ORACLE RDBMS. The query results are asserted as CLIPS facts.

The query generator was developed primarily for use within the ICADS project (Intelligent Computer Aided Design System) currently being developed by the CAD Research Unit in the California Polytechnic State University (Cal Poly). In ICADS, there are several parallel or distributed expert systems accessing a common knowledge base of facts. Each expert system has a narrow domain of interest and therefore needs only certain portions of the information. The query generator provides a common method of accessing this information and allows the expert system to specify what data is needed without specifying how to retrieve it.

Dr. Laurian Chirica is a Professor of Computer Science; James Snyder is a student in the Computer Science Department at the California Polytechnic State University, San Luis Obispo, California.

INTRODUCTION

Currently, Cal Poly's CAD Research Unit is developing an Intelligent Computer Aided Design System (ICADS). This system is composed of several domain expert systems running concurrently under the control of a blackboard [Pohl, Myers, Chapman, Cotton, 1989]. The current application area under development is architecture, but the system's applicability can be easily extended to other disciplines. In order for the domain expert systems to evaluate a design, a large amount of information needs to be available to the expert systems. This body of information does not remain static and therefore needs a management system. In addition, there are two major classes of information needed by an expert system: reference information and prototype information. Reference information can be described as tabular information such as a parts catalog. Each part has an identifier, a description, and a price. Another example of reference information is thermal lag times for various construction materials.

Prototype information comes from a knowledge representation scheme called Prototypical Information [Gero, Maher, Zhang, 1988]. A prototype describes the general characteristics that most objects have. For example, the ICADS project uses a Building Type Prototype Database. This database stores information about typical high-rise apartments. Some of the kinds of information stored are: owner goals and objectives, user group profiles, and designer criteria.

Prototypical information has a very complex structure unlike reference information. Complex retrieval methods are necessary for certain information and application programmers should not be concerned with the details of retrieving information. Not only is prototype and reference information needed within expert systems, but it is needed in other environments as well, such as C programs. Because common information is needed in disjoint environments, a common storage mechanism is needed, namely a Data Base Management System (DBMS). A DBMS provides a recoverable and concurrent method of storage and retrieval of data. These features are very necessary within the ICADS project because there are many independent expert systems executing, all of which could access the database. Figure 1 shows the ICADS system architecture and how expert systems, which we refer to as Intelligent Design Tools (IDT), need access to database information.

The current DBMS of choice is the Relational DBMS (RDBMS). Because of its simplicity and power, it has become the DBMS standard. RDBMSs use a Fourth Generation Language (4GL) or query language to perform operations on database objects [Korth, Silberschatz, 1986]. The *de facto* standard 4GL is the Structured Query Language (SQL). This query language is available on most hardware platforms and operating systems from PCs to super-computers.

PROBLEM DEFINITION AND REQUIREMENTS

The ICADS project uses CLIPS as its expert system shell, which in its standard version does not support RDBMS access. Because an RDBMS provides a common storage and retrieval method, we decided that relational database access within CLIPS was necessary. The solution had several general requirements which needed to be met to be useful. They were:

- To allow the use of the standard RDBMS features.

- To be easy for the expert system developer to use.

- To allow for easy integration into the CLIPS source code.

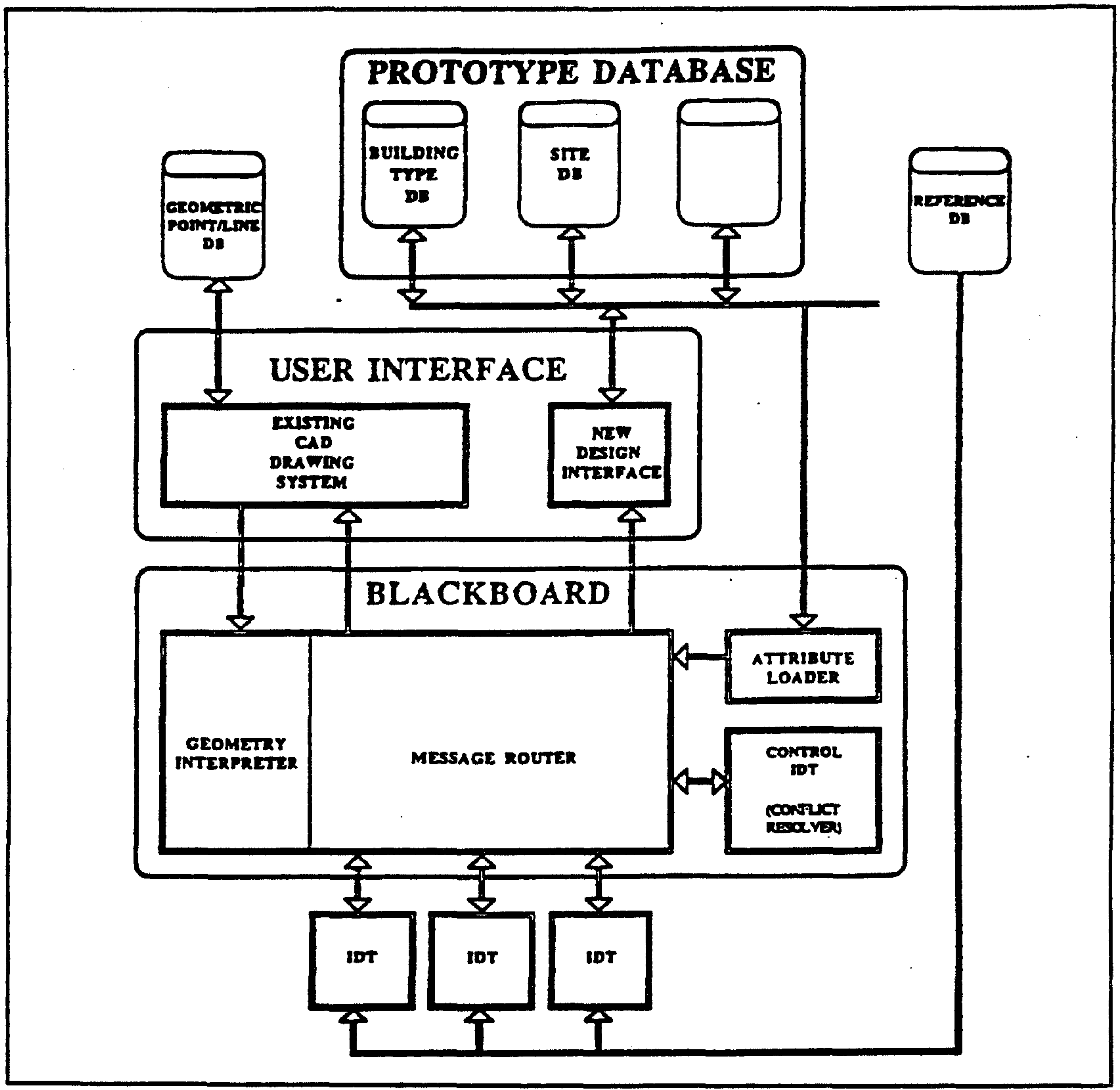


Figure 1 - ICADS system architecture

The CLIPS database access system needed to allow for most of the database queries possible. Figure 2 illustrates the general form of an SQL query. The SELECT clause defines the particular attributes to be retrieved. For example, the description of a part would be listed in this clause. The FROM clause defines which database relations the information is to come from. In this scheme, data can come from multiple relations in a single query. A user retrieves only the information needed from each relation.

The WHERE clause defines constraints under which data is retrieved. For example, only the employees in department 10 should be considered. In addition, the WHERE clause can contain a join condition which tells the database system that a join between two or more relations needs to be executed. For example, an attribute of each employee is the department number they belong to. You want a list of all employees and the name of the department they work in. This information is not contained in one

```
(getsql query1 employee.name department.name
       = employee.d_no department.d_no
       = department.d_no 10
)
```

Figure 4 - Sample CLIPS SQL query

addition, each row from the database that is returned is prefixed with the user's query label and asserted as CLIPS fact by calling a C function provided by CLIPS.

The translation from the CLIPS syntax to SQL is very natural. The SELECT clause is obtained from the <*column-list*> previously defined. The FROM clause is obtained by building a list of relation names from the <*column-list*>relation names. The WHERE clause is obtained from the <*condition-list*> defined above. The operator and the first value are inverted to conform to the SQL syntax. The ORDER BY clause is implicitly built by the <*column-list*>. The data will be sorted based on the order in which the columns were listed.

SAMPLE APPLICATIONS

The Attribute Loader

In ICADS, the Attribute Loader is a special expert system which reads information from the database and asserts it into the semantic network. The ICADS project uses a frame-based representation to store information within its expert systems [Pohl, Myers, Chapman, Cotton, 1989]. The primary function of the Attribute Loader is to read the information from the database, assemble frames and assert them as facts.

The information is obtained from the ICADS Prototype Database, which contains information about typical building types and typical site locations. The structure of the prototype database is shown in Figure 4. The boxed items represent base relations; the circles represent relationship relations between base relations. Currently, objects, attributes, and values are retrieved from the database and asserted. The query generator allows the expert system source to remain constant even if a new database management system is used.

DBRESOLVE

Because we use frame-based knowledge, expert systems have frames as patterns in rules. Some frames can be quite complicated and can contain typographical errors. To increase programmer productivity, we created a program which resolves frames in an expert system with the information contained within the database. The DBRESOLVE programs function is very similar to a cross-referencing tool but is applied to frames.

The DBRESOLVE program scans the expert system source and identifies the occurrences of frame information which needs to be verified. This information is then checked against the database information. Any frames which are not contained in the database are flagged as incorrect.

place. It resides in two relations: employees and departments. A join condition specifies that the employee's department number must match a department number in the department relation.

The ORDER BY clause sorts the data in a specific ordering. If this clause is not specified, the data is returned in a system-dependent order which may not remain constant over time.

```
SELECT      <list of column names>
FROM        <list of relation names>
WHERE       <list of boolean conditions>
ORDER BY    <list of columns names>
```

Figure 2 - General form of an SQL query

From the point of view of an expert system developer, the database access should be intuitive and easy to use. The ideal solution would allow the user to specify the desired information in à CLIPS-like syntax. This considerably reduces the learning curve of the database access system.

Placing the database access system within the CLIPS environment should be as simple as adding any other user-defined CLIPS function. The database access system should be as small and fast as possible.

AN SQL QUERY GENERATOR FOR CLIPS

Our solution to the above problem was to develop a query generator for CLIPS. The function of the query generator is to take a CLIPS database query within a rule, translate it into SQL and submit the query for execution. The results of the query are then asserted as CLIPS facts.

The implementation of the query generator can be divided into three areas: the CLIPS interface syntax, the SQL interface, and the translation process. The general CLIPS syntax is defined in Figure 3. A *<label>* defines a unique label to prefix the facts when they are asserted. A *<column-list>* is a list of columns prefixed with a relation name to eliminate any ambiguous references to columns. For example, two different relations may have a column named "description". There must be a way to differentiate between each column, so they are prefixed with their relation names. A *<condition-list>* is a relational operator followed by constants or column names.

```
(getsql <label> <column-list> <condition-list>)
```

Figure 3 - General CLIPS - SQL syntax

Figure 4 shows a complete example. The query label is "query1". The employee names and department numbers from department 10 will be asserted as facts. Notice the join condition between the employee and department relations.

The SQL interface is invisible to a CLIPS user. The interface pertains only to the RDBMS that is used. In the ICADS system, we used the Embedded SQL option which allows C programs to submit queries for execution [Korth, Silberschatz, 1986]. Embedded SQL naturally falls in line with CLIPS, which is also written in C. We designed the system to take any SQL query and submit it for execution. In

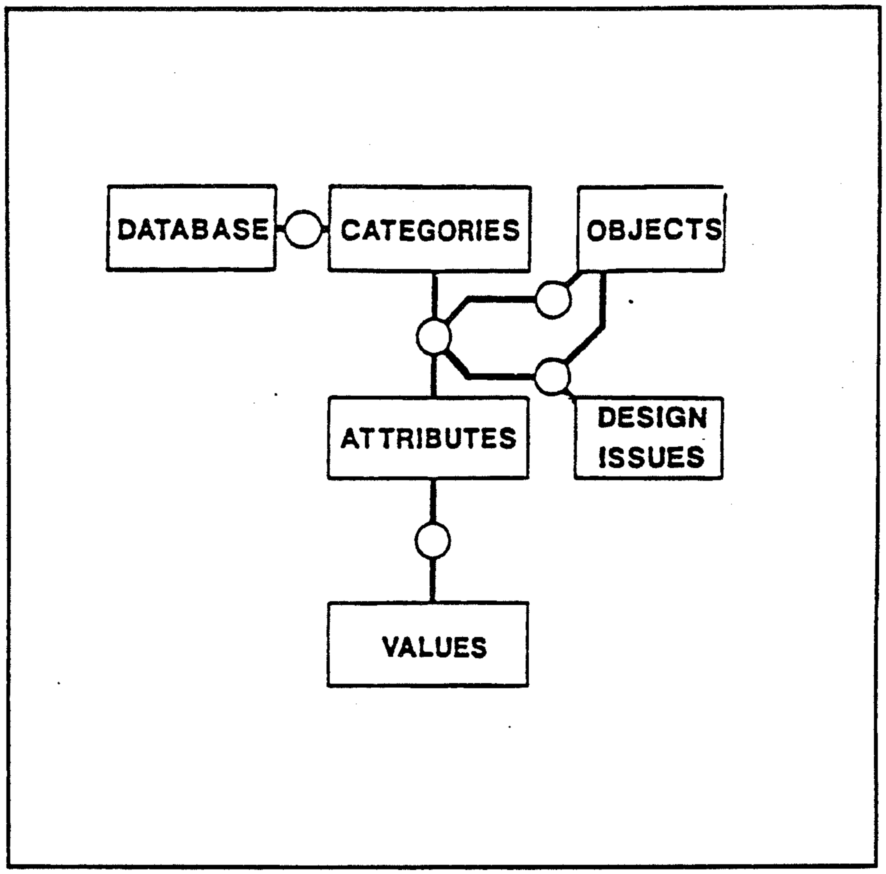


Figure 5 - Prototype Database Structure

CONCLUSIONS

Our initial hopes for decreased development time were easily met. Because of system's simplicity, queries can be easily written. Using a RDBMS allows the application programmer to only retrieve the information they need, which is much better than storing information in hard coded facts or reading information from disk files. In addition, other environments can access the same information.

Initially, we had concerns that query times would be too large. This proved to be quite the opposite. Because of the buffer management of the RDBMS, many queries execute faster than if the same information were read from a disk file.

Perhaps the most important feature of using a RDBMS is the concurrency, integrity and reusability of data in many orthogonal environments. Within ICADS, many programs and expert systems access the same relations. If any of the data within a relation changes, every system which accesses it retrieves the current and correctly updated values. Concurrency and integrity control would be extremely complicated to add to CLIPS, but it comes automatically by using a RDBMS.

The above factors make an RDBMS a superior method of information storage and retrieval. We have not yet encountered any drawbacks to using this approach.

BIBLIOGRAPHY

[Gero, Maher, Zhang 1988] Gero,J., M. Maher and W. Zhang; 'Chunking Structural Design Knowledge as Prototypes'; Working Paper, Architectural Computing Unit, Department of Architectural Science, University of Sydney, Australia, January, 1988.

[Korth and Silberschatz 1986] Korth H.F. and A. Silberschatz; 'Database System Concepts'; McGraw-Hill, 1986.

[Pohl, Myers, Chapman, Cotton 1989] Pohl, J., L. Myers, A. Chapman, J. Cotton; 'ICADS: Working Model Version 1'; Technical Report, CADRU-03-89, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, Calif., USA, December, 1989.

ΣCLIPSE = Presentation Management + NASA Clips + SQL

Bernard P. Wess, Jr. *

May 1, 1990

Abstract

ΣCLIPSE provides a expert systems and "intelligent" data base development program for diverse systems integration environments that require support for automated reasoning and expert systems technology, presentation management, and access to "intelligent" SQL data bases. The ΣCLIPSE technology and and its integrated ability to access 4th generation application development and decision support tools through a portable SQL interface, comprises a sophisticated software development environment for solving knowledge engineering and expert systems development problems in information intensive commercial environments—financial services, health care, and distributed process control—where the expert system must be extendable—a major architectural advantage of NASA Clips.

ΣCLIPSE is a research effort intended to test the viability of merging SQL data bases with expert systems technology.

1 Goals

ΣCLIPSE provides the Management Information Systems (MIS) expert systems developer a unique expert systems environment for:

Integration Much expert systems technology is too difficult for MIS developers to modify or does not adequately integrate with MIS data processing environments which demand support for corporate data bases and sophisticated visual presentation management facilities for professional and clerical users. [Schur88 Scown85] ΣCLIPSE supports commercial "mission critical" and "strategic" applications by extending NASA Clips to take advantage of Presentation Management (PM) functions and ANSI SQL data base access to enhance existing enterprise files and data bases. [Date88]

Standards ΣCLIPSE comprises functional extensions for portable text screens, windowing, fields, and menu development on a variety of operating systems and full graphics capabilities for the IBM PC under MSDOS and Borland Turbo C BGI graphics. ANSI SQL data base management is provided through an SQL C interface to a variety of file and data managers. [MIS89] ΣCLIPSE is an *extended*, not modified version of NASA Clips Version 4.3 [Giarr89] and includes objects and Clips facts and rules language source code for defining and manipulating—windows, forms, screens, reports, menus, fields, and icons.

*The author may be reached at Mentor Communications Ltd, 790 Highland Avenue, Needham Heights, MA 02194, (617)449-0086, Fax (617)449-0476.

Power ΣCLIPSE provides high performance expert systems development capabilities to MIS professionals who need continuing compatibility with future NASA Clips upgrades, portable text windowing, and IBM PC graphics capabilities. All text-based presentation management is portable and IBM PC graphics-based presentation management is MSDOS "extended" to support 16 megabyte Clips applications for the Intel 386/486 processors.

Compatibility ΣCLIPSE offers a complete implementation of NASA Clips and ANSI standard SQL including automatic ROLLBACK and COMMIT functions for commercial transaction processing. SQL is the only ANSI standard relational language for query, data manipulation, data definition and security. Applications developed IBM's SQL/DS and DB2 are very similar to ΣCLIPSE SQL which is ANSI compatible.

Portability Text-based presentation management applications developed with ΣCLIPSE can easily be ported to a wide-range of operating systems and computers, including: IBM PC/OS/2, Unix, DEC/VAX VMS. No Clips source code changes are required. SQL access to data bases and file managers is transparent within ΣCLIPSE Clips rules and provides the application developer with the widest possible range of data retrieval and storage means, including: dBASE, Btrieve, C-tree, CB-tree, and in the future VAX/VMS Rdb, RMS, and Oracle.®

Systems Integration Future access to Oracle's distributed architecture (SQL*Connect), DEC DECNET, or TCP/IP ΣCLIPSE applications will enable *distributed* Clips knowledge bases, when distributed processing is enabled within Clips, to reside on multiple computers and to access DBMS relations transparently through distributed SQL remote procedure calls. [Symb90 Adler90]

Architectural Freedom ΣCLIPSE is divided into three layers—the User Front-End (Presentation Manager) in text or graphics modes, the Clips expert system compiler, Clips language, and the Back-End SQL data base engine implemented as both a Clips external function and internal Clips rule. All functional ΣCLIPSE layers are independent architectural code layers which can be supplemented or replaced based on the changing requirements of the NASA Clips community of users and the demands of commercial MIS users.

2 Presentation versus Data Management

This section outlines the "front-end" of the ΣCLIPSE product. Additionally, ΣCLIPSE provides a dynamically re-configurable data base "back-end" which supports multiple data bases, platforms, applications, and communications environments based on SQL relational data bases. [MIS90]

The independence between presentation management and back-end data base management is provided by a data object object, the actions applied to an object, its relationship(s) to other object(s), and the screen representation of the object from access, through SQL to underlying components or sub-objects of the parent object. [Shu89] Thus an unlimited range of presentation metaphors can be used to represent user interactions. The ΣCLIPSE front-end enables a vastly expanded level of functionality to be incorporated in the presentation, display, manipulation and interaction between application and screen processes and the user. The level of complexity of screen presentation and interaction is greatly enhanced over existing front-ends which either:

- rely on a single metaphor for interacting with the user or display of visual objects or

- require that the front-end be used to build complex displays through the use of icons.

These methods do not enable the use of more complex forms of interaction to be integrated or enhanced in the graphic front-end.

3 Architectural Overview

ΣCLIPSE has the following characteristics:

Object-oriented "Intelligent" displays are comprised of visual objects that have meaning and actions associated with them. Screens are built from complex objects and icons and their associated actions, predetermined by the user/develop or the application logic, lead the user through an application. Icons and/or complex objects (for example, data base tables or spreadsheets) can be moved, manipulated, or act as triggers when activated or changed.

Active Screen Metaphors Any graphical metaphor can be set up which makes sense to the user and aids in representing the underlying application logic or data base. For example, a manager may interact with the program through a spreadsheet where each cell is an active object that itself may be another spreadsheet or piece of a data base. Or a screen may represent a chemical processing plant from which the user can control the operations by manipulating dials, meters, switches, etc.

Virtual Objects The physical screen is not a limitation on the size of a screen object and the data or image it represents. For example, a spreadsheet could handle large data base tables of virtually unlimited size with numerous graphs located in cells as associated screen "child" objects. Or a physical window may represent only a portion of a larger graphical image and the image may be zoomed, panned, expanded, etc.

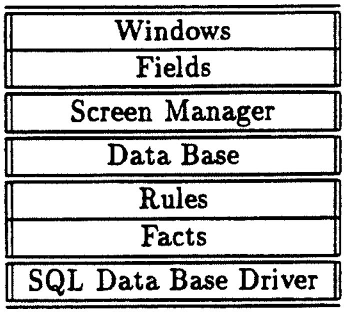
Ease of Use The user interface is highly intelligent, intended for use by professional managers and office workers as well as by MIS professionals.

Integrated DBMS Display tools, such as forms builders and spreadsheets may be integrated in a "seamless" manner through the defintion of more complex objects.

Flexibility Because of the modular nature of the product, design flexibility and independence of architecture, interfaces among ΣCLIPSE modules, underlying application programs, communications technology, and the data base manager are easily modified or replaced. Any component of the program or any associated application can be replaced by another product, such as Excel in the spreadsheet arena or Sybase in the DBMS area.

The simple architecture is outlined in Figure 1.

Figure 1: The simple architecture of the ΣCLIPSE .



Windows

Fields

Screen Manager

Data Base

Rules

Facts

SQL Data Base Driver

4 ΣCLIPSE Front-End Features

4.1 General Features

The *general features* of the the program include the following and are available regardless of the Front-end (FE) mode of operation. Both text and graphics modes of operation can be executed simultaneously.

FE Drivers The FE driver can be replaced or enhanced to extend the functionality of the FE.

C Interface The user's application and the FE itself may be extended or merged by *registering* external C functions to be recognized by ΣCLIPSE and/or the user's application.

Compiled Screens All screen objects—windows and fields, for example, are compiled once at application run-time if they are not loaded in binary format. Therefore, screen objects are manipulated on the display with maximum speed so the user sees fast screen updates.

4.2 The Text-based Front-End Features

The text-based FE provides the following functionality:

4.2.1 Window & Display Control

Windows A window is an area of the *logical* screen that is treated as a separate display entity. Windows may have borders, overlap, or cover one another and have a priority that is user or application assigned in real-time. They may also be larger then the physical display.

Forms or Pages A "page" of "forms" is a virtual screen which may be smaller, larger, or the same size as a physical display. Pages or windows may be named.

Scrolling Automatic scrolling is accomplished to orient the proper current window. Scrolling may be horizontal or vertical, as required. The application may write to hidden window areas without causing scrolling.

Logical Write An application may write to a window without causing a window to update the physical screen until all application output has ended.

Logical Attributes A display "attribute" table is maintained which is logical in nature. The logical display attributes, for example, "red" are converted to display driver output in real-time by the FE.

Character Writing Characters may be written with or without attributes and for one or more characters.

String Writing Strings may be written with or without attributes and for one or more characters.

Cursor Control Full "virtual" cursor control is available including the ability to write to a window's "current position," with a virtual or physical row and column attached. The cursor may "drag" the virtual windows, leave the display unchanged, or make the cursor position the current position.

4.3 Field Level Functions

The following functions are available for fields:

Repeating Lines A formatted "block" may be entered once and repeated for scrolling formatted windows of identical lines of input. The lines in a block can exceed the actual physical window size and automatic scrolling will occur.

Validation Format strings control the character field-level input and output. ΣCLIPSE rules can intercede to more fully control field and character I/O based on external application function calls or rule execution.

Field Editing A full-featured text editor is automatically invoked for each field. Intra- and extra-field movement can be controlled within a window or field.

Field Functions Window and field level functions include exit to next field or previous field, field above or field below, beginning or end of window, previous or next window, line up or down, send data, delete data, or abort.

Field Data Types String; 8-, 16-, 32-bit signed and unsigned binary, 32-bit monetary, date, time, 32- and 64-bit floating point.

Edit Patterns Character strings format a field on input or output so that formatted fields are properly presented, for example, account, SSNs and telephone numbers.

4.4 Graphical User Interface Functions

The following functions are available within the GUI and are unique to the bit-mapped GUI FE:

Graphics System Automatic detection of hardware and resolution and driver loading for more than 30 modes of operation. Movement from character-based I/O to bit-mapped graphics is supported. Multiple pages of graphics are supported for the appropriate hardware drivers.

Graphics An unlimited variety of objects can be drawn directly on the physical screen (not in window buffers). Arcs, circles, polygons, ellipses, lines, points, 3-D and 2-D bars and bar charts, line and point charts, pie slices, rectangles, as well as icons of any size are available. Functions to flood and pattern fill objects, rotate, zoom, move and manipulate lines and polygons are available.

Fonts & Icons Multiple fonts are available, including *sans-serif*, gothic, triplex, and roman. Fonts may be oriented, sized, colored, and transposed. Icons may be loaded and displayed from external font and icon tables. Pixels, characters, strings, and images may be interrogated or manipulated. Both bit-mapped and "stroked" fonts are available.

Graphic Function Library Many internal functions, accessible to user processes, programs, and functions are provided for business and engineering graphics.

Text Output Full-text control is available including style, centering, color, orientation, size, and magnification.

Color Control Color can be applied through a "color palette" to objects, characters, windows, and pixels. A "color table" is defined to control colors.

State Control The "state" of an object, character, window, display or pixel can be interrogated and the results sent as a message as a fact into the data base or to a function. Full application control is enabled through the message system to maintain flexibility.

Icon/Object Library Graphical objects may be created with the graphics editor and dynamically called from memory or disk. A library of icons and presentation management metaphors is available for customization and use in new user defined applications.

5 Front-End Communications Interface

The FE utilizes the ΣCLIPSE development language and support utilities to "define" or to "create" display objects such as windows and fields. Moreover, ΣCLIPSE can manipulate any defined object. The FE or the application can *directly* execute SQL commands.

ΣCLIPSE allows any data base or graphical object to be modified by sending command *messages* to ΣCLIPSE from user-developed applications, external events, and changes in the state of objects or data. Also, objects may send messages directly to other objects for processing, without the need for application program or user intervention.

6 References

[Schur88] Stephen Schur,"The Intelligent Data Base", *AI Expert*, pp. 26–34, Jan. 1988

[Scown85] Susan B. Scown, *The Artificial Intelligence Experience: An Introduction*, Digital Equipment Corporation, 1985.

[Date88]Date & White, *A Guide to DB2*, Second Edition, Addison Wesley, New York, 1988.

[MIS89]*CQL Data Base/Program Development System*, Version 6.0, Machine Independent Software, Fourth Edition, 1989.

[Giarr89]Giarratano & Riley, *Expert Systems: Principles and Programming*, PWS–Kent, Boston, MA, 1989.

[Symb90]Symbiotics, *MetaCourier User's Guide*, Symbiotics, Inc., Jan. 1990, Pre-Release Version 1.2.4, Cambridge MA 1990.

[Adler90]Adler & Cottman,*A Development Framework for AI Based Distributed Operations Support Systems*, Fifth Conference on AI for Space Applications, Huntsville, Alabama, May 22–23, 1990

[Shu89]Shu, N., "Visual programming: Perspectives and approaches", *IBM Systems Journal*, Vol. 28, No. 4, 1989, pp. 525–547.

CLIPS1.TEX

omit

A8 Session:
Space Station Freedom Applications

A PC Based Fault Diagnosis Expert System

Christopher A. Marsh
The MITRE Corporation
1120 NASA Road One
Houston, Texas 77089

Abstract

The Integrated Status Assessment (ISA) prototype expert system performs system level fault diagnosis using rules and models created by the user. The ISA evolved from concepts to a stand-alone demonstration prototype using OPS5 on a LISP Machine. The LISP based prototype was rewritten in C and the C Language Integrated Production System (CLIPS) to run on a Personal Computer (PC) and a graphics workstation. The ISA prototype has been used to demonstrate fault diagnosis functions of Space Station Freedom's Operation Management System (OMS). This paper describes the development of the ISA prototype from early concepts to the current PC/workstation version used today and describes future areas of development for the prototype.

Introduction

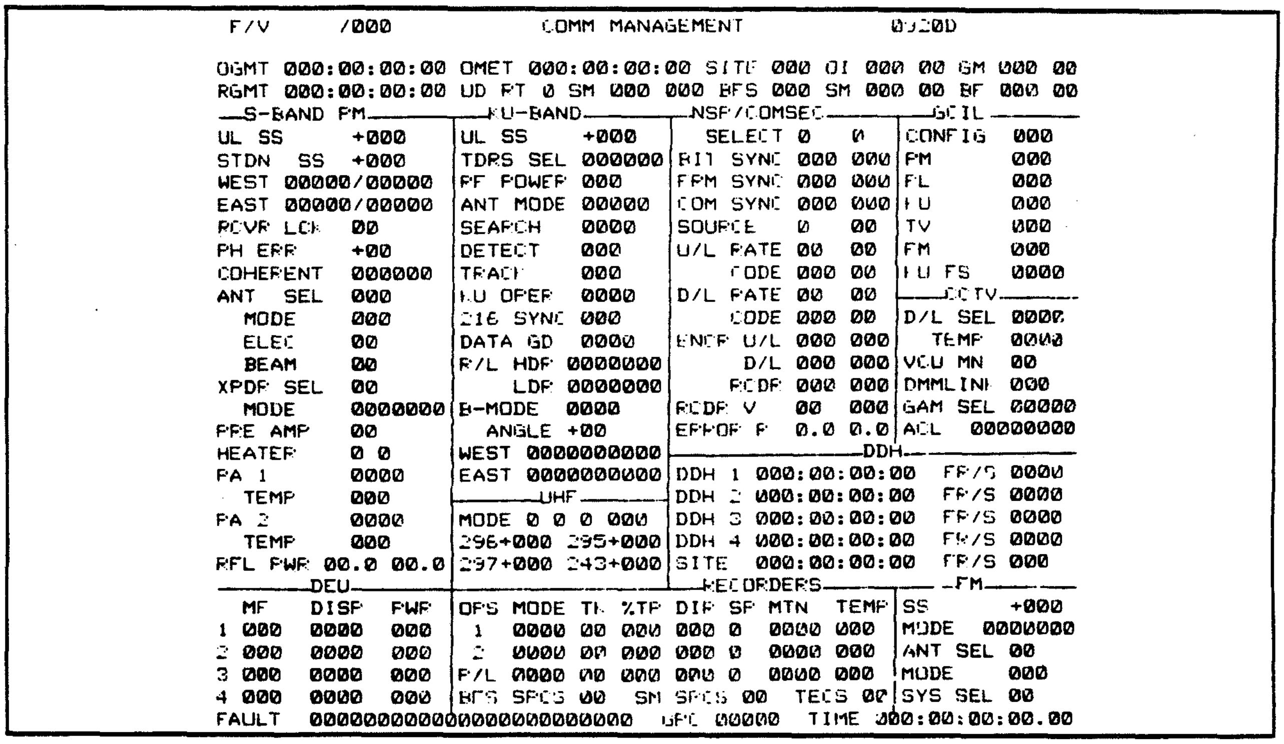
The Integrated Status Assessment (ISA) expert system is a fault diagnosis system that has moved from a concept to the integration phase of development. It started out as a demonstration prototype to help develop Operations Management Application (OMA) requirements for Space Station Freedom and not as a delivery product. The ISA has gone beyond its early demonstration prototype to an integrated field prototype to help answer operations and integration issues. The ISA will continue to evolve as a research prototype and it will be used to influence the development of a delivery fault diagnosis system for Space Station Freedom.

Concepts

In 1985 the Mission Operations Directorate (MOD) asked the MITRE Corporation to help develop requirements for system management of Space Station Freedom. The MITRE task addressed Space Station systems' control and monitoring. It helped develop concepts and requirements for the management of Freedom's onboard systems. The focus was to define the interfaces among the integrated systems management functions and the interfaces between integrated systems management and the individual core systems [1]. This task lead to the development of requirements for the Operations Management System (OMS) that performs the integrated systems management function for Space Station Freedom.

In developing the System Management concepts, the example of Space Shuttle flight control was used as a model. The Space Shuttle is managed on the ground by a flight director responsible for the overall mission, several front room flight controllers each responsible for a different system, and many back room controllers who each support a front room controller. This approach has been used since the early days of spaceflight and is manpower intensive.

Prior to 1989, flight controllers spent much of their time watching screens full of changing numbers representing sensor readings onboard the Space Shuttle (refer to Figure 1 for a typical flight controller screen). When a fault was detected, the flight controller refers to the malfunction procedures and flight rule books to guide them through the isolation of the fault and the reconfiguration of the shuttle. Each of these books were hundreds of pages long and sat in book shelves behind each controller. In addition to the books on the ground the astronauts carried hundreds of pounds of material in the Flight Data File (FDF) on each flight to guide them on the operation of the Shuttle. Failure analysis on the Shuttle was very manpower intensive and automation could play a major role in supporting the flight controllers. Much has been learned from the way systems management and failure analysis was done on the Shuttle for the Space Station Freedom program.



```
F/V      /000             COMM MANAGEMENT                 0320D

OGMT 000:00:00:00 OMET 000:00:00:00 SITE 000 OI 000 00 GM 000 00
RGMT 000:00:00:00 UD RT 0 SM 000 000 BFS 000 SM 000 00 BF 000 00
--S-BAND PM----------KU-BAND-----------NSP/COMSEC---------GCIL-------
UL SS       +000  |UL SS     +000   |  SELECT 0    0  |CONFIG   000
STDN  SS    +000  |TDRS SEL  000000 |BIT SYNC 000 000 |PM       000
WEST 00000/00000  |RF POWER  000    |FRM SYNC 000 000 |PL       000
EAST 00000/00000  |ANT MODE  00000  |COM SYNC 000 000 |KU       000
RCVR LCK    00    |SEARCH    0000   |SOURCE   0   00  |TV       000
PH ERR      +00   |DETECT    000    |U/L RATE 00  00  |FM       000
COHERENT    000000|TRACK     000    |    CODE 000 00  |KU FS    0000
ANT  SEL    000   |KU OPER   0000   |D/L RATE 00  00  |----CCTV-----
  MODE      000   |216 SYNC  000    |    CODE 000 00  |D/L SEL  0000
  ELEC      00    |DATA GD   0000   |ENCR U/L 000 000 |  TEMP   0000
  BEAM      00    |R/L HDR 0000000  |     D/L 000 000 |VCU MN   00
XPDR SEL    00    |    LDR 0000000  |    RCDR 000 000 |DMMLINK  000
  MODE      0000000|B-MODE  0000    |RCDR V   00  000 |GAM SEL  00000
PRE AMP     00    |  ANGLE +00      |ERROR R  0.0 0.0 |ACL   00000000
HEATER      0 0   |WEST 0000000000  |-----------------DDH-------------
PA 1        0000  |EAST 0000000000  |DDH 1 000:00:00:00   FR/S 0000
  TEMP      000   |--------UHF------|DDH 2 000:00:00:00   FR/S 0000
PA 2        0000  |MODE 0 0 0 000   |DDH 3 000:00:00:00   FR/S 0000
  TEMP      000   |296+000 295+000  |DDH 4 000:00:00:00   FR/S 0000
RFL PWR 00.0 00.0 |297+000 243+000  |SITE  000:00:00:00   FR/S 000
---------DEU-------------------------------RECORDERS---------FM------
  MF   DISP   PWR |OPS MODE TK %TP DIR SP MTN  TEMP |SS        +000
1 000  0000   000 | 1  0000 00 000 000 0  0000 000  |MODE   0000000
2 000  0000   000 | 2  0000 00 000 000 0  0000 000  |ANT SEL 00
3 000  0000   000 |P/L 0000 00 000 000 0  0000 000  |MODE      000
4 000  0000   000 |BFS SPCS 00   SM SPCS 00  TECS 00|SYS SEL 00
FAULT  00000000000000000000000000  GPC 00000  TIME 000:00:00:00.00
```

Figure 1. Typical Flight Controller Screen

The concepts developed from studying the Space Shuttle systems management were for more automated systems management functions. A systems hierarchy similar to the personnel hierarchy used for Space Shuttle flight control was envisioned for Space Station Freedom. This systems management hierarchy for Space Station Freedom would have multiple levels. The top level would be Integrated System Management (ISM). ISM would perform systems management at the highest level across all systems. This level of management is similar to the management performed by the astronauts on the Shuttle and controllers in the Flight Control Room. Each system would have its own System Management Application (SMA). Each subsystem within a system would have its own System Operations Application (SOA). Management at the lower two levels is similar to what is performed by the controllers in the Multipurpose Support Rooms and the other support rooms. It was recognized that systems management relies upon knowledge of the systems and their operation. Figure 2 shows the system management hierarchy for Shuttle and Space Station Freedom. As experience with Space Station operations is gained, the applications performing systems management would be refined to incorporate the new knowledge.

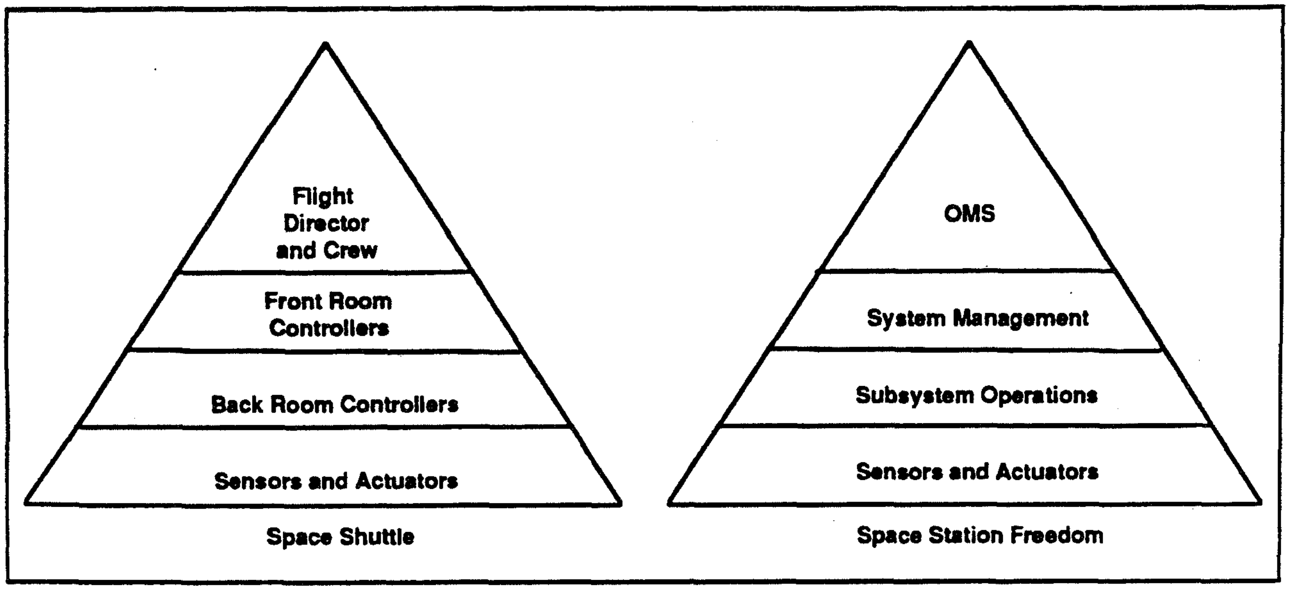


Figure 2. System Management Hierarchy

After the initial concepts were developed, the Operations Management System (OMS) working group was formed to define requirements for the top level of system management (originally called ISM) now called the OMS. Because of its position in the systems management hierarchy, the OMS would integrate operations across all the Station systems and elements. It would include onboard automation (the Operations Management Application (OMA)), ground based automation (the Operations Management Ground Application (OMGA)) , onboard manual operations (the flight crew), and ground based manual operations (the ground controllers). Figure 3 shows the division of the OMS. The OMS would perform high level functions including global planning, system/payload testing, command management, inventory management, maintenance, and training. The intention of the OMS was not to eliminate the work of humans, but to enhance it.

The OMS has been baselined as part of the Space Station Freedom program. Requirements of the OMS incorporated most of the SMA and SOA ideas as tier II and tier III system and subsystem managers.

Once the initial concepts and requirements were documented, the development of prototypes to further develop the concepts was started.

Demonstration Prototype

This effort involved the development of several prototypes of the systems management functions. These prototypes included the Integrated Status Assessment (ISA), Planning Support Environment (PSE), the Procedures Interpreter (PI), On-Orbit Maintenance (OOM), and Communications and Tracking System Management (C&T-SMA). These prototypes were used to demonstrate new concepts, educate the users about emerging expert system technology, and to further develop requirements for the OMA through comments and feedback from the user community. The prototypes were also used to assess proposed designs for OMS implementation. The ISA and PI prototypes were further developed to test OMA concepts in a test bed environment while the others were used only for gathering initial requirements. The following paragraphs describes the development of the ISA.

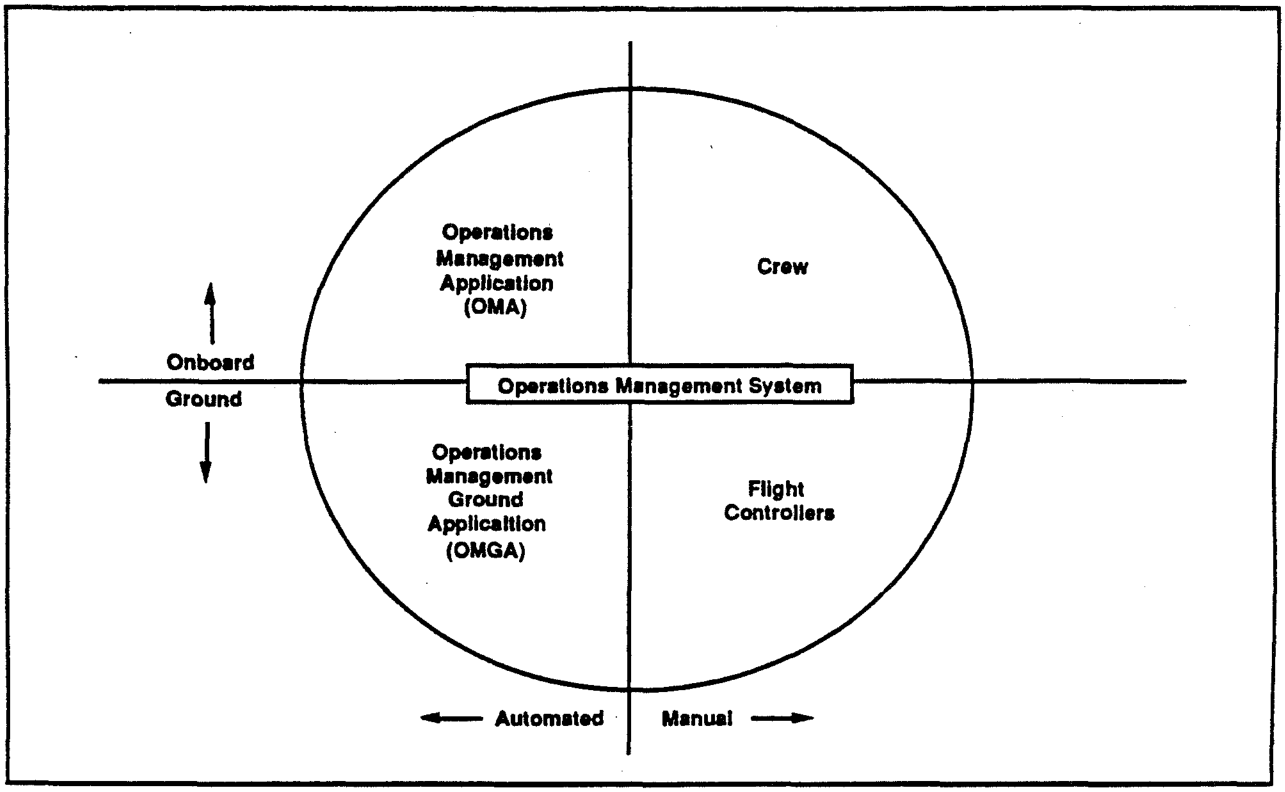

Operations Management Application (OMA)
Crew
Onboard
Ground
Operations Management System
Operations Management Ground Applicaltion (OMGA)
Flight Controllers
Automated
Manual

Figure 3. Operations Management System Components

The ISA project was a one person task that included attending working group meetings, requirements development, and briefings and demonstrations as well as prototype development. The main purpose of the ISA task was to introduce technology and educate the user community, develop requirements and not to do expert systems research.

The ISA prototype was developed to illustrate several functions. The ISA demonstrated the concept of gathering data from the various Space Station Freedom systems. It displayed the data in a coherent integrated manner with a user friendly graphical interface. In failure situations, the ISA showed how expert systems and advanced user interfaces could be used to determine the cause of the problem with a trace of its reasoning and possible recommendations [2].

Because the task of assessing the status of space vehicles is a complex job that requires "expert" knowledge to find heuristic solutions to problems, an expert system approach was chosen to prototype the ISA system. For the initial demonstration prototype, the ISA system was hosted on a Symbolics™ 3600 series computer and written in LISP and OPS5.

The knowledge engineering process took several months for the initial domain of the ISA prototype. The expert spent about a half day per week critiquing and suggesting changes to the system. More of the expert's time would have been useful but this was not possible. After most of the knowledge engineering process was completed, an elaborate user interface was added to the system. Next many other operations people were shown the prototype and their ideas and knowledge was later used to further refine the system. Requirements learned from this process were input to the OMS working group and later turned into Space Station Freedom requirements.

™Symbolics is a trademark of Symbolics Incorporated

The initial domain for the ISA prototype was the communications and tracking KU band system. This included the power busses, cooling loops, Tracking Data Relay Satellite System (TDRSS), and the interface to the DMS. This area was chosen because experts were available in this area and because it contained the intersections of several systems; a fault in one system could cause other systems to malfunction.

The ISA prototype is a rule- and model-based expert system that demonstrates Space Station Freedom fault detection and isolation. The ISA consists of a knowledge base, an inference engine, and a user interface. The knowledge base consists of facts and rules. The facts contain a high level qualitative model of Space Station Freedom. The rules consist of generic fault isolation knowledge and system specific knowledge to determine the source of faults. The inference engine controls how the knowledge base interacts with itself. The user interface gives the user overall control and allows the developer to examine and easily modify the knowledge base. The user interface gives the user overall control and allows the developer to examine and easily modify the knowledge base. Figure 4 contains a diagram of the system.

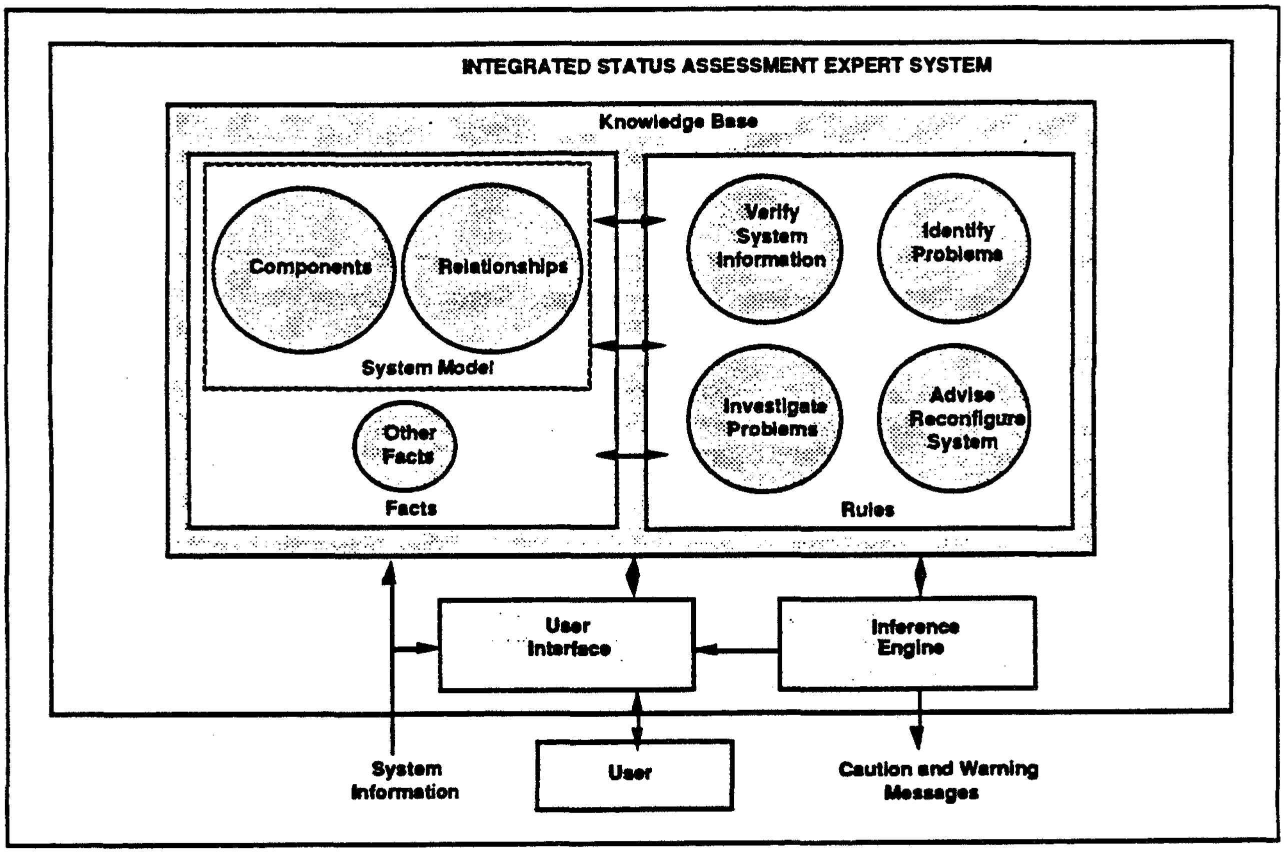


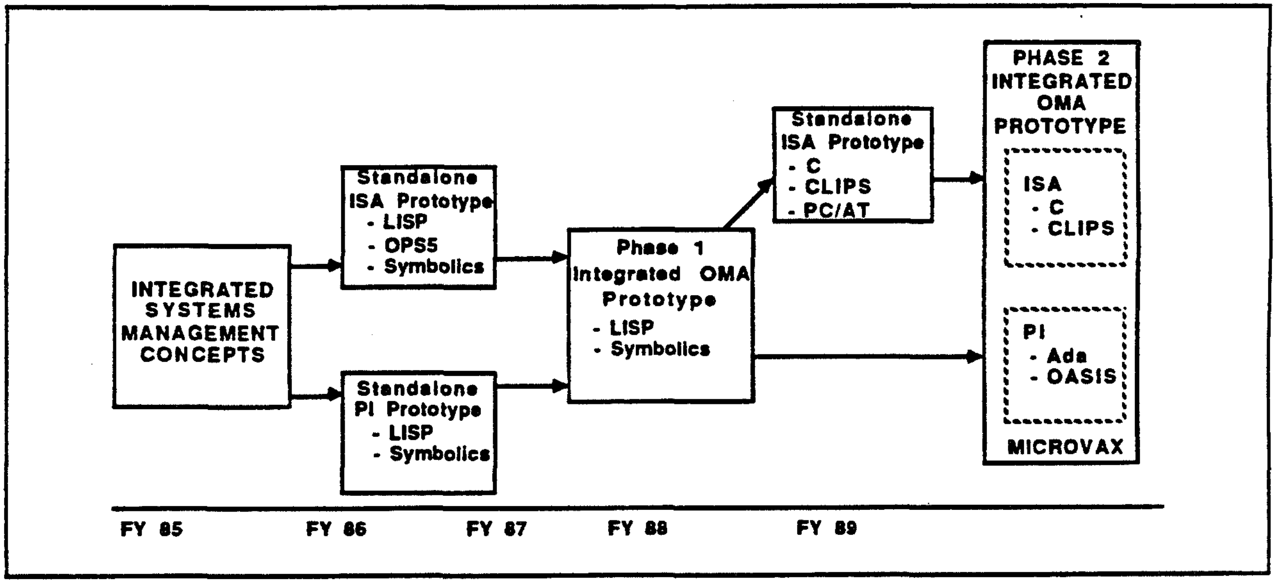
Figure 4. ISA Prototype Components

The ISA operates in an update, run, and react cycle. New operational data from a simulation file is input into the ISA and these values are used to update the model. If some of the operational data is off nominal, them the ISA rules are run on the model to isolate the source of the fault [3]. Once the fault is isolated, additional rules may react to safe Space Station Freedom. After the rules are through firing, the ISA can accept new data from the simulation file to update the model again.

At the end of the first demonstration phase of development, the ISA prototype was built with portions of the Communications and Tracking System, the Electrical Power System, the Data Management System, and the Environmental Control and Life Support System modeled. It has been demonstrated to many groups at NASA including the astronauts, flight controllers, engineers, and many different Space Station contractors. At each demonstration new ideas to improve the prototype and drive new requirements were solicited. These ideas and requirements were incorporated into NASA documentation and the OMS definition. The OMS definition document was incorporated into the Space Station Architecture Control Documents (ACDs).

Transition to a Test Bed Environment

The ISA prototype has been moved onto the Data Management System (DMS) Test Bed Through the OMS integration effort. The OMS integration effort is bringing many Space Station Freedom prototypes and simulations together to form a field environment for testing and developing OMS concepts. Moving applications in the OMS integration effort allows them to be run in as close to a real world environment as is possible. This is a necessary transition environment to test operations concepts with the applications. For the ISA prototype, this took place in two main phases: integration of the Symbolics LISP based prototype and integration of a rewritten C based prototype. Figure 5 shows the evolution path for the ISA prototype.



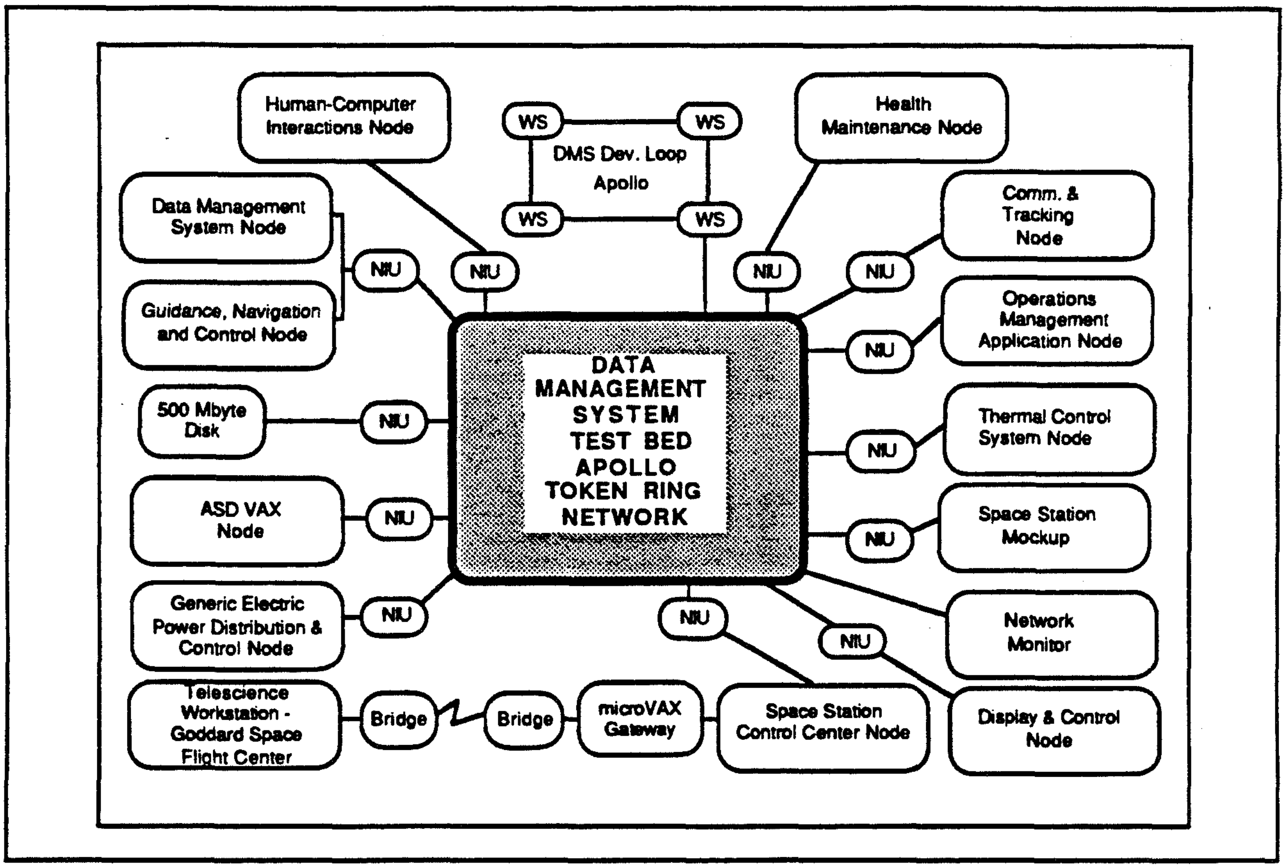
INTEGRATED SYSTEMS MANAGEMENT CONCEPTS
Standalone ISA Prototype - LISP - OPS5 - Symbolics
Standalone PI Prototype - LISP - Symbolics
Phase 1 Integrated OMA Prototype - LISP - Symbolics
Standalone ISA Prototype - C - CLIPS - PC/AT
PHASE 2 INTEGRATED OMA PROTOTYPE
ISA - C - CLIPS
PI - Ada - OASIS
MICROVAX
FY 85
FY 86
FY 87
FY 88
FY 89

Figure 5. Evolution Paths for ISA and PI

OMS Integration

There are many Space Station Freedom systems being modeled and simulated in test beds across the country. Many of these test beds are tied together by the Data Management System (DMS) Test Bed as part of the OMS integration effort. The OMS integration effort provides an environment to test and develop OMA concepts.

As part of the OMS integration effort, the OMA node is the focal point on the DMS network for demonstrating integrated operations of Space Station Freedom systems, with system simulations being represented on various nodes on the Test Bed. Phase One of this integration demonstrated the OMA (represented by the integrated ISA and PI prototypes) controlling and monitoring the Guidance, Navigation, and Control Emulation Laboratory in the execution of a reboost procedure [4]. Phase Two of the OMS integration effort adds four more simulations to the scenario: the Operations Management Ground Application; Generic Electric Power Distribution and Control; Communications and Tracking; and Thermal Control. Future plans include the incorporation of payload and life sciences nodes at the Johnson Space Center and the Marshall Space Flight Center and a data generating node from the European Space Agency. The DMS Test Bed is evolving to be closer to a real Space Station Freedom environment with real world scenarios. Figure 6 shows the DMS Test Bed configuration. In the future the DMS Test Bed will change to real Space Station Freedom like hardware using real DMS communications services.



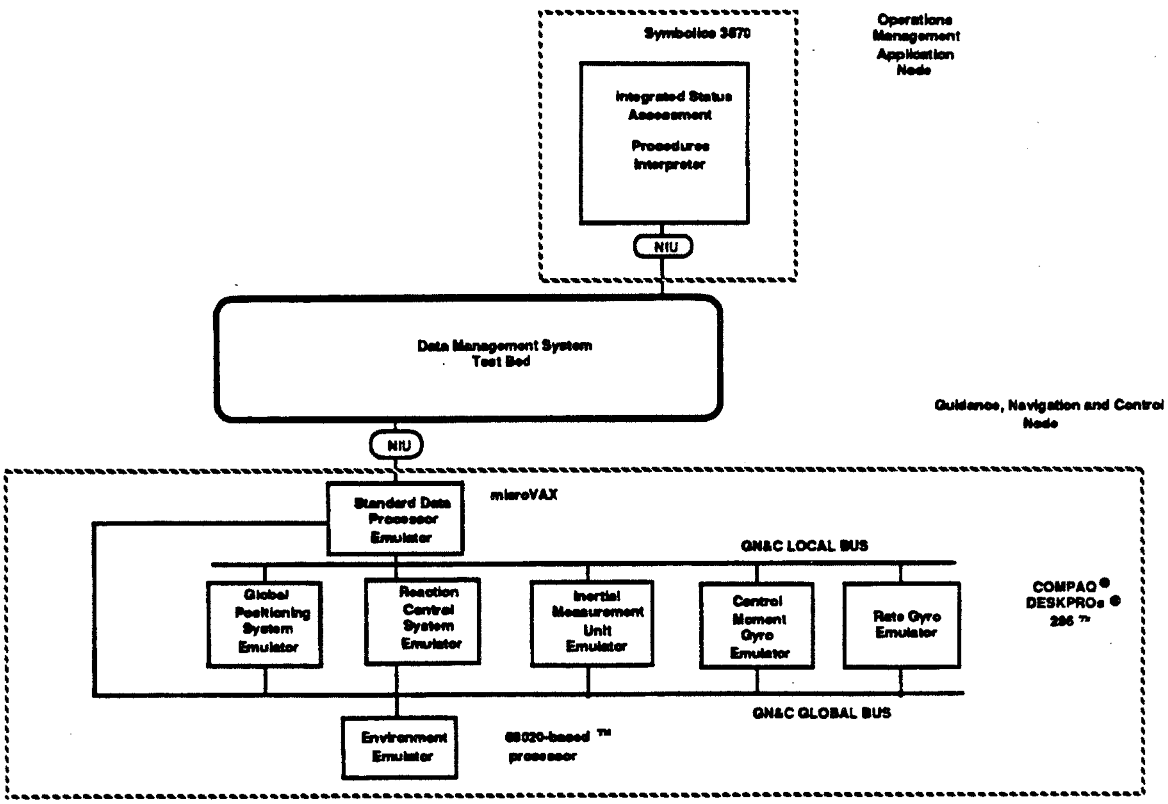
Human-Computer Interactions Node
WS
WS
DMS Dev. Loop Apollo
WS
WS
Health Maintenance Node
Data Management System Node
Comm. & Tracking Node
NIU
NIU
NIU
NIU
Guidance, Navigation and Control Node
Operations Management Application Node
NIU
DATA MANAGEMENT SYSTEM TEST BED APOLLO TOKEN RING NETWORK
500 Mbyte Disk
NIU
Thermal Control System Node
NIU
ASD VAX Node
NIU
Space Station Mockup
NIU
Generic Electric Power Distribution & Control Node
NIU
NIU
Network Monitor
NIU
Telescience Workstation - Goddard Space Flight Center
Bridge
Bridge
microVAX Gateway
Space Station Control Center Node
Display & Control Node

Figure 6. DMS Test Bed Configuration

Phase One OMA Integration

The first phase of the integration effort was to integrate the Symbolics based ISA that performed fault diagnosis with the Symbolics based Procedures Interpreter (PI) that executed and monitored procedures. This proved to be an easy task because of the nature of the object oriented Flavors system on the Symbolics. Messages were sent from one prototype to the other to pass information. When the PI needed expert system advice on a problem it would send ISA a "run" message. When the ISA rules fired and concluded that an action was needed by the PI, it would send PI the appropriate message.

Once these two prototypes were integrated to form the OMA prototype, the next step was to integrate with the DMS Test Bed and the Guidance Navigation and Control (GN&C) Integration Laboratory. Figure 7 shows the final configuration for Phase One of the OMA integration. The DMS Test Bed has three software communications services available to users: the Network Operating System (NOS) for low level communication; the Data Acquisition and Distribution Services (DADS) for requesting point to point cyclical data sets; and the Ancillary Data Service (ADS) for requesting a single predefined data set. The DADS and ADS are built on the NOS. Because the Symbolics only had the NOS software available to it, special code had to be written to communicate with the GN&C Laboratory.


Symbolics 3670
Operations Management Application Node
Integrated Status Assessment
Procedures Interpreter
NIU
Data Management System Test Bed
Guidance, Navigation and Control Node
NIU
Standard Data Processor Emulator
microVAX
GN&C LOCAL BUS
Global Positioning System Emulator
Reaction Control System Emulator
Inertial Measurement Unit Emulator
Control Moment Gyro Emulator
Rate Gyro Emulator
COMPAQ®
DESKPROs ®
286 ™
GN&C GLOBAL BUS
Environment Emulator
68020-based ™
processor

® COMPAQ and DESKPRO
™ 68020

Figure 7. Phase I OMA Integration

The Phase One demonstration showed the execution of a reboost procedure by the PI with faults inserted by the GN&C system. The ISA responded to the faults by advising that the reboost procedure be aborted when appropriate. While the phase one demonstration lacked the depth needed to completely test the ISA, it did integrate it with other systems.

Phase Two OMA Integration

Because the Symbolics hardware did not match well with the proposed Space Station Freedom architecture and because it did not have full support of all the software communication services on the DMS Test Bed, the

®COMPAQ and DESKPRO are registered trademarks of COMPAQ Computer Corporation
™68020 is a trademark of Motorola Incorporated

ISA and PI prototypes were ported to a MicroVAX[®] computer. The ISA was ported to C and the C Language Integrated Production System (CLIPS), an inference engine written in C. The PI was ported to Ada and the Operations And Science Instrument Support (OASIS) software written by the University of Colorado. The two prototypes communicate through interprocess communication on the MicroVAX. Because of the portability of C and CLIPS it was possible to first port the ISA on a PC based system. The only difference between the PC based ISA and the MicroVAX based ISA is the PC version has its own user interface written in C and the MicroVAX ISA uses OASIS for the user interface. The following sections describe the porting of the ISA system.

LISP to C

All the LISP code from the demonstration prototype was rewritten in C for Phase Two of OMA integration. For the procedural code, this was a straight forward process of rewriting LISP functions in C. For the Object Oriented code this involved creating C data structures and functions to replace the LISP Objects (Flavors and Methods). For the PC version, Graphic libraries and machine level calls to mouse driver routines were used to create a friendly user interface similar to the Symbolics version of the prototype. The models used by ISA are generated using the mouse and pop-up menus and create both graphical schematics and data structures describing the system. The model data structures are translated into CLIPS facts for the rules to reason about. The rules are written in CLIPS and can be controlled through the user interface. Special CLIPS functions were developed to allow the rules to modify the screen as well as the model.

Three difficulties were encountered in rewriting the ISA in C. First, the software tools for C were not as good as the LISP tools on the Symbolics. The Symbolics software development environment contained an integrated editor, debugger, flavor examiner, and compiler. These tools were as a big advantage in writing the various iterations of the original prototype. Other tools were used for the C version that were less capable. The impact of this was that a larger percent of time was spent debugging code. This was not a major problem, however, because most of the software design was completed with the LISP version and most of the coding in C was translation and not totally new code. Currently there are many new software development environments coming on the market for C and other languages and in the future the lack of tools should no longer be a problem for rehosting code from LISP to C.

The second difficulty in recoding the ISA was in building the new user interface for the stand-alone PC version. The Symbolics had its own unique object oriented windowing software, but there was no tool available with the same kind of functionality for the PC. All window functions had to be built from scratch; that took most of the coding time. The user interface was not an issue for the MicroVAX implementation of the ISA because it used OASIS for displays. In the future the use of windowing standards such as X.11 and software libraries to support such standards will reduce the difficulty of this problem.

The third difficulty of the rehosting process was substituting VAX Mailbox interprocess communication for the Flavor Message passing to communicate with the PI prototype. VAX Mailboxes are a system utility available on the MicroVAX to allow two programs to communicate to each other through a buffer in memory. VAX Mailboxes are not as easy to implement in C as messages are in LISP.

OPS5 to CLIPS

All the rules were recoded form OPS5 to use the CLIPS inference engine. CLIPS is a forward chaining production system similar to OPS5 that is written in C. Because the two rule systems are very similar in nature the recoding of the rules was an easy process [5]. All the structure of the rules remained the same

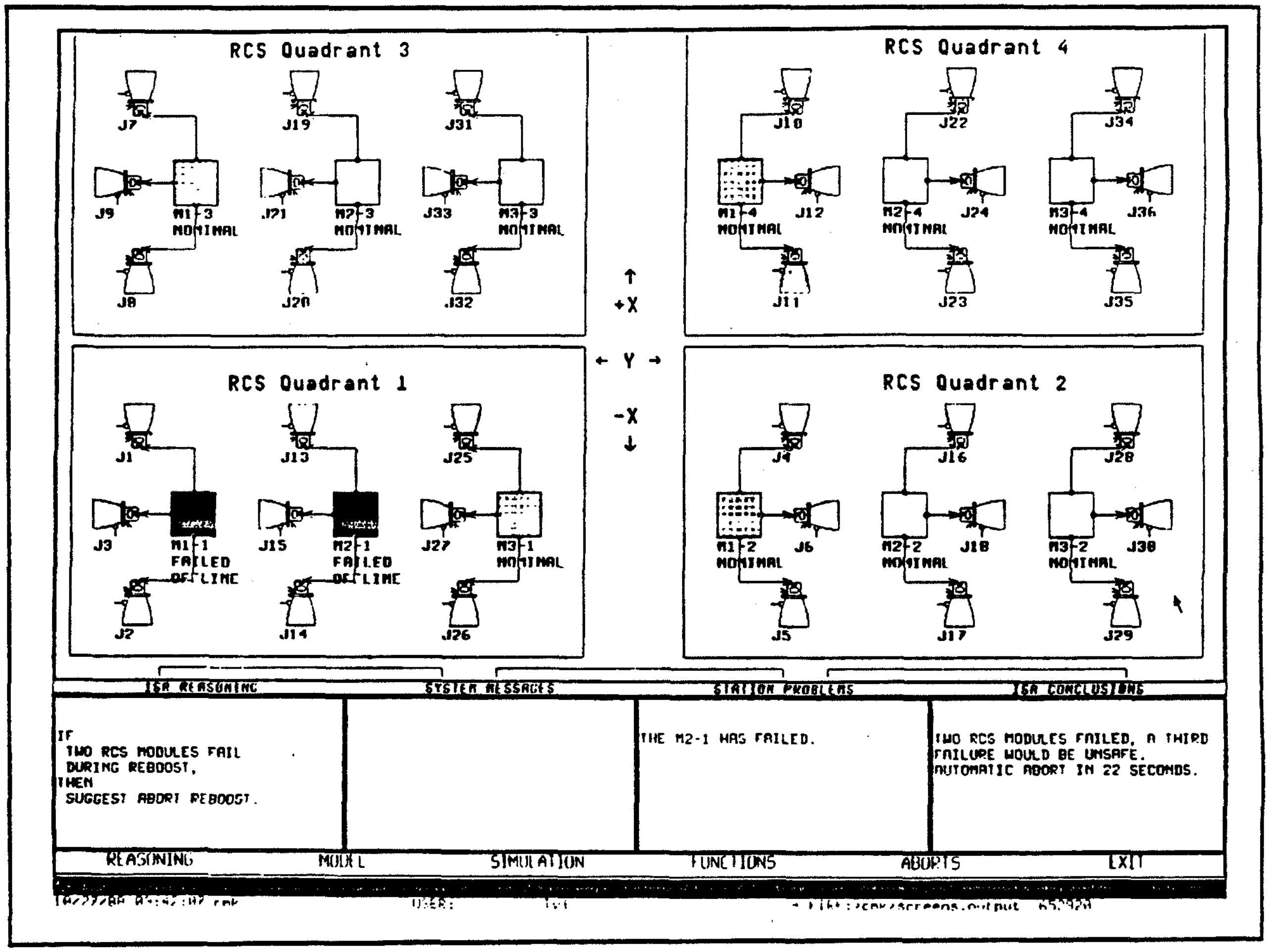
[®] MicroVAX is a registered trade mark of Digital Equipment Corporation with only minor syntax changes. The only difficulty was working around the lack of objects in CLIPS. This problem is currently being addressed by CLIPS developers for future releases of CLIPS.

Transferring System Models

A large portion of the knowledge in the ISA expert system is contained in the system models. These models contain all the static object knowledge used by the inference engine and all the graphic display information for schematics. Because the models are stored as text files, there was no change required to go from the LISP version of the ISA to the C version of the ISA.

Symbolics Graphics to PC Graphics

The large, high resolution black and white display of the Symbolics proved to be very useful to let the user view the status of systems. Figure 8 show the original Symbolics ISA screen. The PC based ISA has an Enhanced Graphics Adapter (EGA) which supports less resolution than the original so some detail was lost on the new display. The PC version also make use of multiple windows, panning, and zooming to show the user information. Figure 9 shows the ISA with EGA graphics.



```
IF
 TWO RCS MODULES FAIL
 DURING REBOOST,
THEN
 SUGGEST ABORT REBOOST.
```

THE M2-1 HAS FAILED.

TWO RCS MODULES FAILED, A THIRD FAILURE WOULD BE UNSAFE. AUTOMATIC ABORT IN 22 SECONDS.

Figure 8. Symbolics ISA Screen

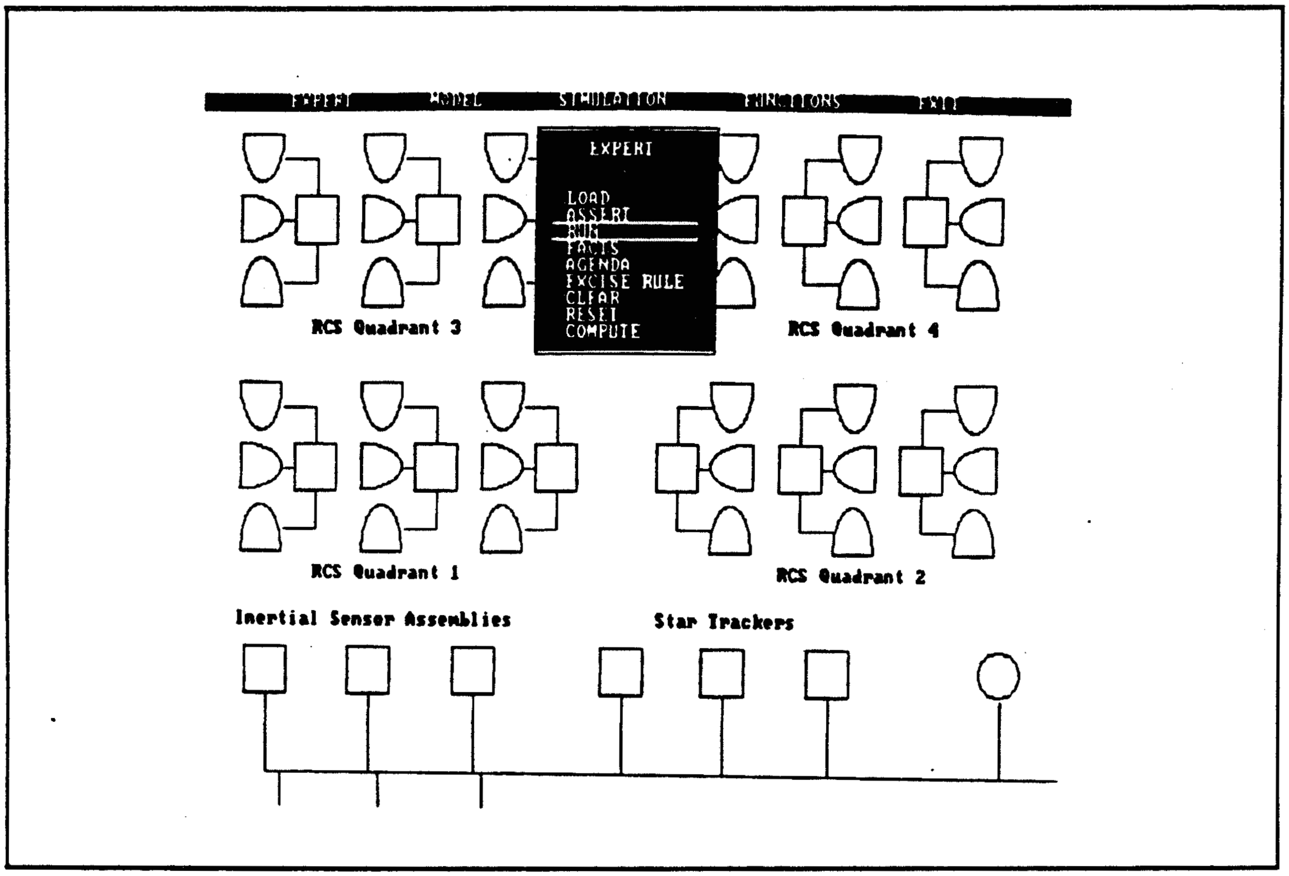


Figure 9. EGA ISA Screen

Phase Two Testing

Phase Two of the OMS integration effort adds four more simulations to the scenario: the Operations Management Ground Application; Generic Electric Power Distribution and Control; Communications and Tracking; and Thermal Control. Future plans include the incorporation of payload and life sciences nodes at the Johnson Space Center and the Marshall Space Flight Center and a data generating node from the European Space Agency. Phase Two will involve diagnosing failures in the Communications and Tracking System and in the Thermal Control System simulations.

While these additional simulations will add robustness to the OMS integration effort it will still lack the integrated closed loop simulation to back them up. What takes place in one simulation is not reflected in the others. The OMS integration effort is still just many separate simulations tied together on a communications link. To really test the ISA in its global fault detection capabilities, the simulations need to be more integrated. This is something that the OMS integration effort project needs to have, not only to test global fault detection capabilities but to test other OMA functions such as global resource management and station planning.

The Future of the ISA Prototype

The ISA system is still a prototype that will continue to evolve and remain a tool to help influence the design of the real OMA fault diagnosis function. The OMS integration effort has to have an integrated simulation capability. The remainder of the OMA functions need to be prototyped in an integrated environment. Deeper reasoning capabilities need to be built into the system. The following sections describe these future steps.

Better Simulation Capabilities

In order to test expert systems on the DMS Test Bed, there needs to be a Test Bed Simulation Coordination Node. This node will exist on the DMS Test Bed (probably as an additional process on existing hardware) and produce DADS data or an ADS data-set to coordinate the integrated demonstrations. Each node will have access to this data-set to get information about the current simulation being run. These are some of the features that such a node could provide: identification of the scenario being run; starting and stopping times of the current demonstration; current demonstration status; the ability to reflect changes on one node into another system; better coordination of integrated and multiple failures; and global resource levels. These features would help present a more coherent integrated simulation and move the Test Bed away from the awkward canned demonstrations we have today to a true testing environment.

Deeper Reasoning Capabilities

Rule based expert systems can perform good fault diagnosis for systems that fail where they are expected to fail. Unfortunately, systems don't always fail in a mode that was predicted. In the Apollo 13 mission to the Moon, two fuel cells were taken out in a mysterious explosion. There were no flight rules to handle such a failure. Because humans can reason much deeper that their written procedures they were able to bring the astronauts home safely. Model based expert systems that reason from first principals use the physical design instead of just rules to perform diagnosis [6]. These systems can diagnosis faults that were not planned for in a purely rule based expert system. The ISA currently uses high level models in its reasoning. To perform better fault detection and isolation, deeper models that contain behavior information will be used in the ISA.

C to Ada

All Space Station Freedom code will have to be written in Ada. Therefore the OMA fault diagnosis functions will have to be written in Ada. What limitations will this have on the system? Are the current generation of Ada compilers efficient enough to work within the hardware limitations and software requirements of Space Station Freedom and support expert systems? These are areas that need to be investigated.

Conclusions

The use of the OMA demonstration prototypes greatly helped illustrate the ideas developed in the concepts stage. These prototypes communicated these ideas to people much more effectively than the traditional concept and requirements documents alone did. Many useful comments and suggestions were gathered during the many demonstrations of the OMA prototypes. These comments and suggestions greatly enhanced the OMS requirements. The demonstrations also helped us gain many allies to support our ideas for a the implementation of the real system and also help give support to bring advanced automation and expert systems into the existing Space Shuttle program.

The rapid prototyping environment of the Symbolics proved to be very efficient for the stand-alone demonstration prototypes. Changes could be easily integrated into the ISA prototype in just a few minuets. Similar tools would have been very useful in later implementations of the ISA.

The use of C as the language for the integrated prototype was good because it allowed the ISA to be easily tied into CLIPS. The use of C and CLIPS allowed for the ISA to run on a multiple hardware platforms from a PC to a MicroVAX.

Making the software data driven from text files made the transition from LISP to C much easier and made it easy to introduce changes in both the knowledge base and the user interface. OPS5 and CLIPS both used text files for their rules and had a similar syntax. All the structure of the rules remained the same with only minor editing changes. The only difficulty was working around the lack of objects in CLIPS. A large portion of the knowledge in the ISA expert system is contained in the system models. These models contain all the static object knowledge used by the inference engine and all the graphic display information for schematics. Because the models are stored as text files, there was no change required to go from the LISP version of the ISA to the C version of the ISA.

LIST OF REFERENCES

[1] National Aeronautics and Space Administration (1986), *Space Station Systems Operations Concepts for Systems Management*, JSC-20792, Houston, TX: NASA Johnson Space Center.

[2] Marsh, C. (1988), *The ISA Expert System: A Prototype System for Failure Diagnosis on the Space Station*, The First International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, June 1-3, 1988, ACM Press, (60 - 74).

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

[4] Kelly, Christine M. (1988), *Automated Space Station Procedure Execution*, AIAA 26th Aerospace Science Meeting, Reno, NV, January 1988.

[5] Giarratano, J. C. (1989), *CLIPS User's Guide*, Artificial Intelligence Section, Lyndon B. Johnson Space Center, May 1989

[6] Davis, R. (1984), *Diagnostic Reasoning Based on Structure and Behavior*, Artificial Intelligence volume 24 (ISSN: 0004-3702), Elsevier Science Publishers B.V., Amsterdam, The Netherlands.

omit

B8 Session:
User Interface I

58-61

N96-12921

358509

p. 12

CLIPS Interface Development Tools and Their Application

Bernard A. Engel [1], Chris C. Rewerts, Raghavan Srinivasan,
Joseph B. Rogers, and Don D. Jones

Abstract

A package of C-based PC user interface development functions has been developed and integrated into CLIPS. The primary function is **ask** which provides a means to ask the user questions via multiple choice menus or the keyboard and then return the user response to CLIPS. A parameter-like structure supplies information for the interface. Another function, **show**, provides a means to paginate and display text. A third function, **title**, formats and displays title screens. A similar set of C-based functions that are more general and thus will run on UNIX and other machines have also been developed. Seven expert system applications were transformed from commercial development environments into CLIPS and utilize **ask**, **show**, and **title**. Development of numerous new expert system applications using CLIPS and these interface functions has started. These functions greatly reduce the time required to build interfaces for CLIPS applications.

Introduction

The Agricultural Engineering Department at Purdue University has been developing agricultural expert systems (ES) applications since 1984. Numerous applications have been developed since that time including GMA (Grain Marketing Advisor), DSS (Dam Site Selector), DBL-CROP (double crop soybean management ES), and MELON (muskmelon disease diagnosis ES). The use of these and other applications has been limited because of runtime licensing fees, inability to run on machines other than those running DOS, and difficulties with integrating applications with other software. In addition, the academic community has not been interested.in the paperwork associated with the licensing arrangements of most ES development tools. Many of the commercial tools require development and delivery on a single type of machine such as one that runs only DOS. In Indiana, the Cooperative Extension Service (a large potential user for most agricultural ES) have UNIX machines. Therefore, many of the agricultural ES that would be of interest to people in these offices will not run on their machines. The ES that are now being developed often require integration with other computer tools. Most commercial ES development tools do not provide adequate facilities for integration. As a result of these problems, CLIPS was examined as a potential development and delivery tool for agricultural expert systems applications.

One disadvantage of CLIPS, for our purposes, was the lack of a cost-free end-user interface for use with PC compatible machines. The interface available for the PC version of CLIPS requires the purchase of a screen-handling command library from a third-party vendor. End-users of ES produced with the interface provided would have to pay a fee for its use. Also, it was more a development interface than an end-user interface. To avoid such complications, we have built a set of interface functions and integrated them with CLIPS.

[1] The authors are: B.A. Engel, Assistant Professor; C.C. Rewerts, Research Assistant; R. Srinivasan, Research Assistant; J.B. Rogers, AI Systems Programmer; and D.D. Jones, Professor. Agricultural Engineering Department, Purdue University, West Lafayette, IN 47907

[2] ask©, show©, and title© Copyright 1990 Purdue Research Foundation.

User Interfaces

For a computer program to function, it must interface with an outside manipulator or controller. Some programs are controlled by other programs. An interface needed in such situations can typically be described outright, in well-defined terms. The interchanges will be predictable, because machines are involved. In a computer program developed for use by humans, the interface becomes a much different issue. The operation of the interface has a direct bearing on how well one can make use of the program. The user must be able to "run" the program while providing any needed inputs and making any requests for modification of runtime functions. There are many common programs with simple operations that function automatically when invoked, such as using *ls* or *dir* to list files in a DOS directory. However, the programs we refer to in the context of this paper are application programs requiring more user-machine interaction during program operation, such as an ES or simulation. In such cases, "the interface *is* the system for most users" [1].

The Need for an End-User Interface Package

In developing numerous application programs for distribution to a large audience of users with a wide range of computer backgrounds, we needed an interface package that could be incorporated into separate application programs. Of course, we were not the first to discover this need. In working with the design of several software projects, Faneud and Kirk [2] noted the following complaints:

a. Interface development was consuming a great part of the efforts of ES developers and represented a significant portion of the resulting code - as much as 60%.

b. ES developers were usually inexperienced at interface design, and generally had no interest in becoming experts in low level graphics or other interface tools.

c. There was no consistency of interfaces across applications.

d. It was difficult to provide multiple interfaces across applications.

Some of the benefits we hoped to gain by the development of a user interface that could be used by numerous applications included:

a. Users can employ a small number of computer concepts and syntactical rules, therefore they can concentrate on the task.

b. Program designers find it convenient to reduce the number of situations in which the user can make errors.

c. Different applications using the same interface package will have the same "look and feel" to the user. Thus, once a user becomes acquainted with an interface through the use of one application, the use of subsequent programs with the same type of interface may be made simpler.

Implementation and Development Background

This paper documents a simple user interface and its integration into CLIPS. Although all examples and most of the discussion of the user interface will revolve around its implementation within CLIPS, applications are not limited to ES. Most computer applications designed for a general end-user audience require an interface of one sort or another. With an ES, the operation typically starts with a question and answer session between the user and the program, much like a human expert would use to ascertain the definition of the problem to be addressed. A good interface package would allow the developer of the ES a straight-forward means to define how the ES should go about the task of this interchange. We will describe how the interface and CLIPS communicate, how the interface functions are used from within a CLIPS ES program, and how the interface presents its information and queries to the end-user.

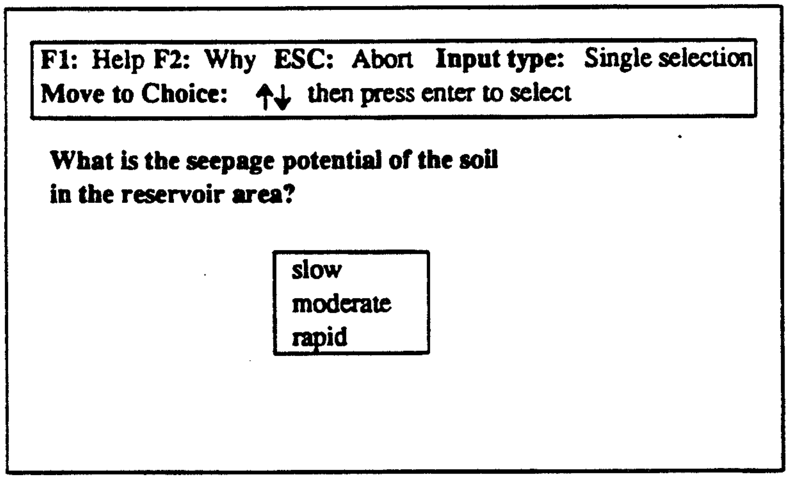
Appearance of Ask to the End-User

Our user interface is designed to use the graphics capabilities of the PC on which it is running, including high resolution graphics, if available. The interface presents itself in color, if available. When graphic capabilities are available on a given machine, provisions have been made to use graphic screens instead of, or in conjunction with, the textual screens used for "help", "why", or the question prompt.

The layout of the **ask** interface screen consists of three areas:

1. An information box,
2. The question prompt, and
3. An area for the input of the user's answer.

The information box (Figure 1), located at the top of the screen, informs the user of the "help", "why", and "abort" keys (F1, F2, and ESC, respectively). Based on the type of question, the information box tells the user what type of input is expected. In Figure 1, the input expected is a single selection from the three alternatives in the menu box. The second line in the information box gives brief instructions for selecting an answer.



F1: Help F2: Why ESC: Abort **Input type:** Single selection
Move to Choice: ↑↓ then press enter to select

What is the seepage potential of the soil in the reservoir area?

slow
moderate
rapid

Figure 1. Layout of an **ask** question screen.

If the user presses the F1 or F2 key, the **ask** program switches to a display screen to print the information provided by the knowledge engineer for the particular question property structure. "Help" or "why" information may be a graphic image, text, both, or neither. If the ESC key is pressed, the intent of the user to abort the program is confirmed with a dialogue box. If confirmed, both the operation of the interface *and* the operation of CLIPS is aborted.

The text of the question is printed in the question prompt area. Ask provides formatting to fit the text neatly on the screen. Below the question prompt is the user input area. It will be a menu box if the question type requires selection from a list of alternatives. Two types of menus are available in **ask**, one allows the user to select a single selection as an answer, and the other allows multiple selections. To select a single answer from the menu box, the user moves to a choice with the mouse or "up-down" arrow keys to highlight a choice. When the user presses the enter key, the highlighted selection is returned to CLIPS. To select multiple answers from a menu, the user may "mark" a highlighted selection with the "right" arrow key. All selections either marked or highlighted when the "enter" key is hit will be returned to CLIPS.

Menu selection is appropriate only when specific answers are expected. To obtain more open-ended responses, ask can prompt for input of text or a number. In this case, a simple prompt for the information is printed in the user input area. Ask can be given a range for numerical entry, and will constrain the user's entry as needed.

Errors

All interface functions are equipped with abilities to detect and report errors. An error is generated when the information being passed to a function is inconsistent with the expected format. (These errors will generally pertain to problems most likely to arise during development of an ES). The action taken by the functions in case of an error is to abort all processes and print a diagnostic statement.

Using Ask in an Expert System

Ask is invoked with a frame-like parameter structure that passes it the information it needs to operate. One of the first things that must be determined is the type of question screen it is to construct. As mentioned above, the four types of question screens **ask** generates are:

1. Multiple choice/single answer,
2. Multiple choice/multiple answer,
3. User input of text,
4. User input of numeric data.

For each question screen, a data structure for the **ask** function must be written. The data structures are stored in CLIPS as *facts*. The data structure will tell the **ask** function how to formulate the question, what kinds of extra information to provide to the user, how to retrieve the user's answer, and how to return the resulting answer to CLIPS.

Creating Instructions for Ask

We will refer to the above-mentioned data structures as "*question property structures*". CLIPS facts are stored as "*fields*", where each field is a word, number or "*string*" (a group of words or numbers contained in double quotes). When **ask** is given a question property structure from which to build a question prompt, it examines the fields one by one, looking for the information it needs.

There are two types of fields expected:

1. labels: Labels are key words used to identify the information that may follow in the next field(s). The **ask** function expects **exactly** thirteen labels.
2. values: Values are the actual information **ask** will use to construct and ask the given question. The **ask** function requires some labels to be followed by values, some label's values may be a certain type, and some may be ignored by the **ask** function (because they may apply elsewhere in the ES or are reserved for a future use).

As **ask** reads through the labels and values of a question property structure, it deduces what type of question it is to ask, based on the values. Table 1 is the list of labels and values, and how they are used by the **ask** function.

Constructing Question Property Structures

To use the **ask** function in a CLIPS program, question property structures must be entered as *facts*. Examples 1 through 4 demonstrate question structures in their format as *facts*. Each example will produce a different type of **ask** question screen.

Example 1.

```
(site-name
prompt
"What is the name of the site you wish to evaluate?"
expect
help
why
value
value-type
default
range
certainty-range
unknown
gprompt ghelp gwhy
)
```

The above example illustrates a question property structure. The first field, (in this case, *site-name*), can be any *word*, which is to say, any combination of legal characters, with no spaces. The purpose of this word is to label the question property structure, so that a rule could be constructed to look for a fact starting with the given word, which it could match and fire (this is explained further in the discussion of the example rule, below).

This example demonstrates the simplest type of question property structure, because it uses the least amount of information allowed: the labels, and a string value (the question) for the prompt. Values for all parameters except prompt are optional. Since no values are given for the *expect* label, ask deduced the question was to be answered by user input of text. To answer the question from the ask-generated interface, the user types in an answer, and presses the "enter" key.

Example 2.

```
(seepage-rate
prompt
"What is the seepage potential of the soil in the reservoir area?"
expect
slow moderate rapid
help
"A soil survey of the proposed reservoir site should provide
information concerning the seepage rate of soil at the site.
However, if the soil survey does not provide this information
answer the question as not being certain and additional
questions will be asked to evaluate the seepage rate."
why
value
value-type SINGLEVALUED
default
range
certainty-range
unknown
gprompt ghelp gwhy
)
```

There are three primary differences between this question property structure (Example 2) and the last example:

1. Three values are listed after the expect prompt, "slow", "moderate", and "rapid". When **ask** reads these values, it will set up a menu-type question, with the values to choose from.
2. Following the *help* label, is a value in the form of a string, "A soil survey of the proposed reservoir...". This is to be the help message displayed when the **F1** key is pressed when the question is asked. To include *why* information in a question property structure, use the same method as for *help*. To view *why* information while answering a question, the user presses the **F2** key.
3. The third difference is the value *SINGLEVALUED* following the value-type label. This tells the **ask** function to allow the user to select only one of the *expect* choices.

Example 2's question property structure produces a question with a menu-type answer selection (Figure 1). To answer the question, the user points to a selection with a mouse or uses the up/down arrow keys to highlight a choice, then the enter key to select the highlighted choice. The selection is returned to a **CLIPS** rule for processing.

Example 3 is a question property structure that will trigger **ask** to create a question that asks the user to input a number. This was done by setting the *value-type* value to *NUMERIC*. Since the expected answers on many questions asking for numerical input will fall within some range, it is logical to set the *range* values. In this case, if the user tries to enter a number outside the range of 1 to 10000, **ask** will inform the user of the range imposed and prompt the user to try again.

Example 3 offers two types of help to the user, text and graphic. The text can be seen following the *help* label. The name of the graphic image file, "area.hlp" appears following the label *ghelp*. :"Area.hlp" is the name of the graphic image file to be shown to the end-user if help is requested.

Example 3.

```
(surface-area
prompt
"What is the surface area of the reservoir, in acres,
if the water in the reservoir is at its normal depth?"
expect
help
"To determine the surface area of the proposed reservoir
at its normal depth, a survey of the area or a blown-up
USGS map of the reservoir site is needed. A planimeter
should be used to determine the area from the survey or map."
why
value
value-type NUMERIC
default
range 1 10000
certainty-range
unknown
gprompt
ghelp area.hlp
gwhy
)
```

Example 4.

```
(water-use
prompt
"What is the intended use of the water that will be impounded
in the reservoir?"
expect
water-supply recreation flood-control
help
"More than one of the expected values can be selected."
why
value
value-type MULTIVALUED
default
range
certainty-range
unknown YES
gprompt
ghelp
gwhy
)
```

Example 4 causes the ask function to generate a question menu that allows the user to choose more than one option, because the value for *value-type* is set to *MULTIVALUED*. Another feature of the question is that it offers the option "unknown" in the menu, as well as the listed *expect* options "water-supply", "recreation", and "flood-control". This is because the value "YES" appears after the label "unknown".

To answer this question with multiple answers, the user selects choices by highlighting a choice, and pressing the right arrow key. This "marks" the highlighted selection. (Inversely, if the left arrow key is pressed, a highlighted choice is "un-marked"). Other choices can be highlighted and marked. When enter is pressed, all choices that are highlighted or marked are returned to CLIPS as the answer.

Example 5. An Example Program Using Ask

```
(deffacts menus
(site-name
 ...rest of question property structure...)
(seepage-rate
 ...rest of question property structure...)
(surface-area
 ...rest of question property structure...)
(water-use
 ...rest of question property structure...))

(defrule interrogator-rule
        ?d <-(?question-name prompt $?question-prop-strct)
        =>
        (bind ?result (ask $?question-prop-strct))
        (assert (?question-name ?result))
        (retract ?d))
```

Points of Interest

Our example program consists of only four *facts* (Examples 1 through 4) and one *rule*. For this reason it is practical to put all the necessary information in one knowledge base file. The **CLIPS** command, *deffacts*, defines information to be loaded as facts. (There are other ways to load or enter facts into the **CLIPS** knowledge base, which we will not concern ourselves with here).

The "interrogator" Rule

The function of the only rule in our ES is to find the question property structure facts, call **ask** to get an answer from the user, and then *assert* that answer as a fact in the knowledge base (also called the *fact-list*). This rule is used in all knowledge bases that use the interface and will call the **ask** function for all question property structure facts.

The Show Function

The **show** function provides a means to display text to the user. Its primary use in an ES is displaying results to the end-user. The text may be stored in a **CLIPS** fact or in a text file. In either case, **show** is passed text, which it parses into lines to fit in a display box on the screen. The user pages through the text until all has been shown.

Example 6. A fact and rule used to invoke **show**

```
(results show "The numerical rating of the site for use as a
dam site is: -100.  The ratings range from -100 to 100 with
100 being the best possible rating of a site for the construction
of a dam and reservoir.")

(defrule show-results
   (declare (salience -1000))
   (? show $?x)
   =>
   (show $?x))
```

Example 6 shows a rule and a fact that would *match* the conditions of the rule. Presumably the fact shown was created during the end-users consultation with the ES. The `(salience -1000)` would give the rule a low priority to fire, thus effectively holding the showing of results until the end of the consultation. The rest of the rule matches a condition with the fact, setting the variable `$?x` to the textual contents of the fact. The action statement, `(show $?x)`, calls the **show** function and passes the fact's contents.

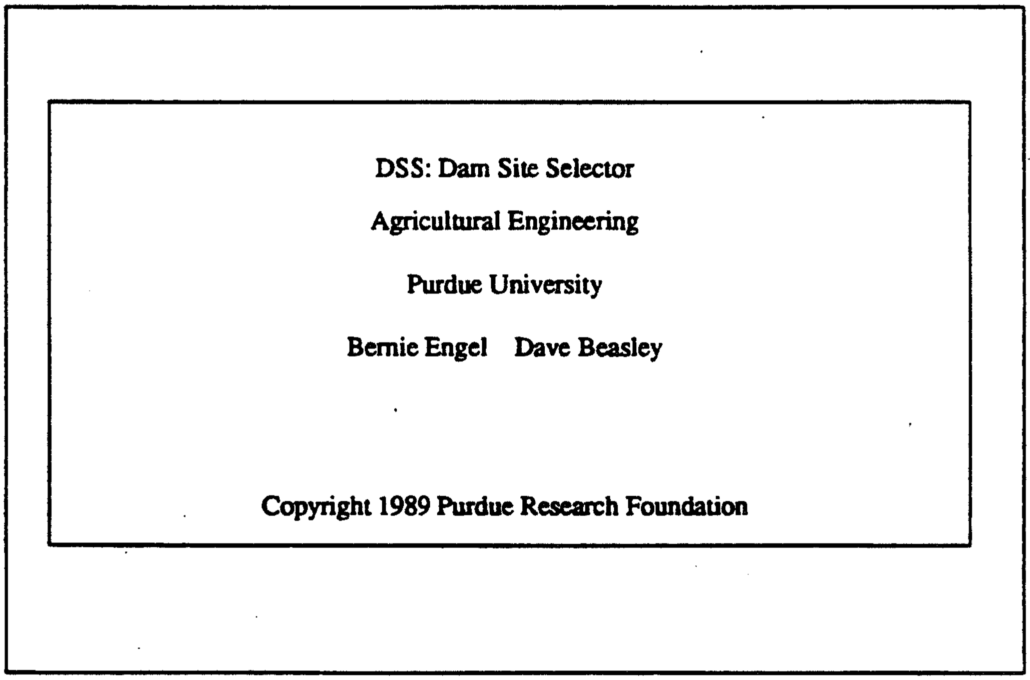
The Title Function

Another accessory interface function is **title**, which can be passed five strings of text to be displayed as a title screen. The first four lines are centered and displayed in a box drawn on the screen, and the fifth allows for the optional display of a copyright note at the bottom of the title box.

Example 7. A fact and rule used to invoke title.

```
(dss-title title "DSS: Dam Site Selector" "Agricultural Engineering"
"Purdue University" "Bernie Engel  Dave Beasley"
"Copyright 1989 Purdue Research Foundation")

(defrule display-title
  (declare (salience 1000))
  (? title $?x)
  =>
  (title $?x))
```

Example 7 illustrates that the **title** function is used much the same as the **show** function. One difference is how **title**'s text inputs are broken into separate strings, to indicate to the program what is to appear on each of the available title screen lines (Figure 2). The only other noticeable difference is *salience* which is set to *1000*, to insure that displaying the title screen is a high priority, since it should be the first thing the end-user sees.



DSS: Dam Site Selector

Agricultural Engineering

Purdue University

Bernie Engel   Dave Beasley

Copyright 1989 Purdue Research Foundation

Figure 2. A title screen produced by Example 7.

Programming Notes

The development of the interface functions was done on a PC-AT, using the "C" language. The source code of the interface programs and CLIPS was *compiled* and *linked* together to make a customized *executable* CLIPS program. The executable program runs on IBM PC-compatible machines. Knowledge engineers may then develop ES using the customized CLIPS shell, making use of the additional functions **ask**, **show**, and **title**.

General Interface

A more general purpose version of the interface was developed by re-writing portions of the PC interface functions. The general purpose interface will work on any machine that runs CLIPS. As stated earlier, one of the reasons for moving to CLIPS was because of its ability to run on a wide variety of machines. The general interface version uses numbered menus with items selected by typing the number associated with the menu item. It does not allow the use of graphics nor does it use boxes around text as the PC version. CLIPS knowledge bases function identical for either interface, allowing applications to operate on a variety of machines.

Interface Application

The interface functions have been used in the development and conversion of several ES. Four of the ES that were transformed from commercial development/delivery tool formats into CLIPS are DAM SITE SELECTOR (DSS) [3], DOUBLE-CROP [4], MELON [5], and the GRAIN MARKETING ADVISOR (GMA) [6]. DAM SITE SELECTOR logically rates potential dam sites and provides an explanation of the factors influencing that rating. DOUBLE-CROP assists with the decision making processes in managing double crop soybeans following winter wheat. MELON assists muskmelon producers with proper management of their crop and with diagnosis and treatment of diseases. The GRAIN MARKETING ADVISOR assists grain producers in the selection of the appropriate grain marketing strategy for their situation. These knowledge bases in their original format required a commercial runtime tool to operate. After the transformation process, these knowledge bases run without a commercial tool and will run on a wider variety of computers. Minor information is lost in the transformation process, but other information is gained [7]. Additional details describing the knowledge base transformation process are provided by Engel et al. [7].

Conclusions

A PC-based end-user interface package has been created and integrated into the CLIPS ES development and run-time tool. CLIPS lacks an easy-to-use end-user interface development tool commonly found in many commercial ES development shells. The end-user interface development package has successfully been used to add interfaces to several CLIPS ES, in transformed knowledge bases, and in the development of new CLIPS ES. A similar set of C-based functions that are more general and thus will run on UNIX and other machines have also been developed and tested.

Benefits gained by using the parameter-driven interface package include:

- Less programming time is needed to complete the development of an application.
- Developers need not worry about many of the details of screen control or other output device-dependent problems.
- Uniformity and modularity is improved across the various programs developed that utilize the interface package.

References

1. Kendall, Kenneth, & Kendall, Julie 1988. Systems Analysis and Design. Prentice-Hall, Inc. Englewood Cliffs, NJ.

2. Faneud, Ross, & Kirk, Steven 1988. A UIMS for Building Metaphoric User Interfaces. In James A. Hender (ed.), Expert Systems: The User Interface, Norwood, NJ:Ablex.

3. Engel, B.A. and D.B. Beasley. 1988. DSS: A dam site selector expert system. In D. Hay (ed.), Planning Now for Irrigation and Drainage in the 21st Century, American Society of Civil Engineers, New York, New York. p. 553-560.

4. Halterman, S.T., J.R. Barrett, and M.L. Swearingin. (1988). *"Double Cropping Expert System"*, in the TRANSACTIONS of the ASAE, 31(1):234-239.

5. Latin, R., G.E. Miles, J.C. Rettinger, and J.R. Mitchell. (1989). *"An Expert System for Diagnosing Muskmelon Disorders"*, in Plant Disease, vol. 73.

6. Thieme, R.H., J.W. Uhrig, R.M. Peart, A.D. Whittaker, and J.R. Barrett. (1987). *"Expert System Techniques Applied to grain Marketing Analysis"*, in Computers and Electronics in Agriculture 1:299-308.

7. Engel, B.A., C. Baffaut, J.R. Barrett, J.B. Rogers, D.D. Jones. 1990. Knowledge transformation. Applied Artificial Intelligence 4:67-80.

Table 1. The ask function question property structure requirements.

Property Label	Uses/functions/requirements
prompt	The *prompt* label must always be followed by a string value, which is the question to be asked of the user.
expect	These are the alternative responses that may be provided for the user to choose from. Ask will present the alternatives in the form of a menu. If the *expect* label is not followed by alternatives, ask assumes it is not to construct a multiple-choice answer in the form of a menu, but instead will assume that the format will be user input of a number or text.
help	An optional string following this label is additional information about the topic of the question that may be of help to the user to understand the question or explain how it is to be answered. The *help* label may be followed by nothing, or the help text string.
why	An optional string following this label informs the user why the information requested by the question is important. The *why* label may be followed by nothing, or the why text in quotes.
value	This property is ignored by the ask function. However, a single-field value may be stored in the slot following this label.
value-type	When the *expect* property is followed by alternative answers, ask will prepare a menu from which the user may choose among the alternatives. The *value-type* property allows the knowledge engineer to indicate whether the given question may be answered by only one or more-than-one of the alternatives. The *value-type* property may also be used to indicate a numerical input is to be expected. Possible values for the *value-type* property are *SINGLEVALUED*, which indicates only one selection is allowed; *MULTIVALUED*, which means the user may choose one or more alternatives; and *NUMERIC*, meaning the user is to input a number. If *NUMERIC* is specified, there should be no *expect* values. If this property is left blank, *SINGLEVALUED* is assumed.
default	This property is ignored by the ask function. A single-field value for *default* may be stored in the slot following this label.
range	When the knowledge engineer wishes the user to enter a numerical answer to a question and needs to restrict the range of values the user may enter, the *range* property should be used. The values for this property should be two numbers separated by a blank. The ask function will require the user's answer to be between the two numbers. If *value-type* is *NUMERIC* and no range is set, ask will allow any number between -1000000000 and 1000000000. If the desired type of question is not to be numerical input, the *range* values must be blank. Also, no *expect* values are allowed in a question where numerical input is desired.
certainty-range	This property is ignored by the ask function, but the label *certainty-range* must be here. Two numerical values may be stored in slots following this label.
unknown	This property can be used if the knowledge engineer wishes *unknown* to be included as one of the menu alternatives. If followed by the value of *yes*, the option *unknown* will be added to list of alternatives.
gprompt	If the question is to use a graphics prompt, then this label should be followed by the file name of the image.
ghelp	If the question is to use a graphics help, then this label should be followed by the file name of the image.
gwhy	If the question is to use a graphics why, then this label should be followed by the file name of the image.

S9-61

N96-12922

p-9

CLIPS: A Proposal for Improved Usability

358513

Charles R. Patton
Computer Sciences Corporation
16511 Space Center Blvd.
Houston, TX 77058

This paper proposes the enhancement of the CLIPS user interface to improve the over-all usability of the CLIPS development environment. It suggests some directions for the long term growth of the user interface, and discusses some specific strengths and weaknesses of the current CLIPS PC user interface.

Every user of CLIPS shares a common experience: their first interaction with the with the system itself. As with any new language, between the process of installing CLIPS on the appropriate computer and the completion of a large application, an intensive learning process takes place. For those with extensive programming knowledge and LISP backgrounds, this experience may have been mostly interesting and pleasant. Being familiar with products that are similar to CLIPS in many ways, these users enjoy a relatively short training period with the product. Already familiar with many of the functions they wish to employ, experienced users are free to focus on the capabilities of CLIPS that make it uniquely useful within their working environment.

To those without the benefits of such a background, however, the first meeting with CLIPS may have been more of a struggle than a triumph. Imagine the worst-case scenario for the aspiring CLIPS programmer. The inexperienced user may know little about rule based programming, so a fundamentally new programming style must be learned. The EMACS editor must be understood before any CLIPS code can be written. The nuances of the CLIPS language and its syntax must be mastered before the simplest program will compile. Testing a rule based system can be especially complex. A new operating system must be mastered. In short, the new CLIPS programmer must complete a lot of learning in a very short time.

Experience has taught us that modifications to the user interface of a software product can make that product both easier to learn and easier to use. A major goal of any changes to the CLIPS user interface would be to reduce the time required to learn the basics of the CLIPS development environment.

Additionally, enhancements to the CLIPS user interface could allow experienced programmers to develop software faster and more easily. Advances in user interface technology allow us to design interfaces specifically suited to multi-dimensional activities like developing rule-based software. Few managers would be opposed to improving the productivity of their programmers, provided the costs of the enhancements are not excessive relative to their benefits.

Another goal of this paper is to promote an awareness of usability issues among CLIPS users and developers. The purpose of these recommendations is to make the CLIPS community aware of some possible user interface enhancements for their development environment. The validity of the following usability recommendations will be established or refuted by CLIPS users. Certainly there are other ideas that will come directly from the users themselves, due to their extensive experience with the product. The user community can then discuss any possible enhancements with CLIPS developers, weighing matters such as costs, benefits, and priorities.

The CLIPS development group is constantly improving its product. As any product is made more powerful, however, it must also become more complex. Additional attention should be paid to the user interface of a product as its capabilities grow, because that product is making greater and greater demands upon the resources of its users. There is more to learn, more to do, and more to remember than there was before enhancements were made. For example, the object-oriented CLIPS system will be more complex than the current releases of this product. Enhancements to its user interface could reduce the amount of complexity presented to the user.

The development of the CLIPS window interface for the PC was a first step toward improving the usability of this product. The application of relatively new interface technologies such as the mouse pointing device and pull-down menus are distinct improvements over the basic command line interface. The window interface clearly saves typing time and reduces the cognitive load of the CLIPS user. While these steps are applauded, there are still aspects of the CLIPS user interface that demand improvement.

Proposed - A New CLIPS Development Environment

Certainly there can be no single CLIPS development environment. CLIPS is run on a variety of platforms in a number of different ways to solve a multitude of problems. Individuals have widely different programming styles that must be accommodated.

The idea behind this new development environment is to create a flexible user interface that can support the beginning user or be adapted to assist the experienced CLIPS programmer. Since the interface supports several different processes (editing, compiling, testing, etc.) a multi-window approach would be appropriate. Wherever possible, interface functions would be devised to reduce the cognitive load on the user.

Consider again the beginning CLIPS user. This person's primary activities are: writing simple programs, compiling them, and testing their functionality. A multi-window user interface can provide all of these capabilities at a glance, reducing the number of things that the user must remember how to access. (See Figure 1.) The user's CLIPS code would be available for editing in the window on the left. Interaction with the compiler and real-time testing would occur in the upper right window. A listing of the currently active facts (i.e. a "show facts" command) is displayed in the window on the lower right.

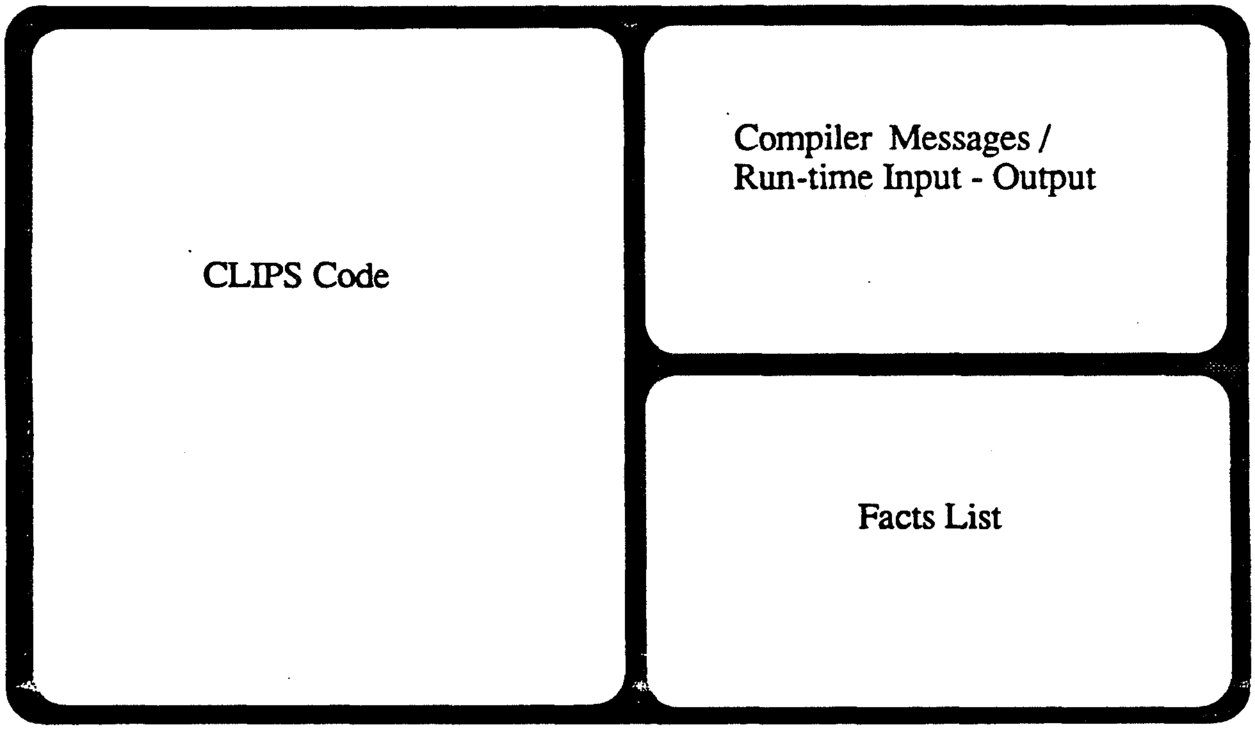


Figure 1: Example of a basic CLIPS development environment.

This display gives the programmer several interesting capabilities. It is possible to see and change the written code as it is compiled, reducing the time required to re-edit source code files. Program activities during testing can be traced back to the source code, speeding up the debugging process. The facts list would provide a constant display of the current facts that the system is using and generating. Here, then, most of the information that a beginning CLIPS programmer needs to know is available in one display. Less time is spent switching between modes and asking for information because it is all currently available on the screen. The user has fewer things to remember as the task is completed. The user can focus on the task at hand, rather than focusing on the processes involved in completing the task.

For CLIPS experts, the interface proposed in figure 1 would not be powerful enough to help them perform their tasks - in fact, it might even slow them down. Advanced users would require additional functionality, like the display shown in Figure 2. Notice that another window is available to display the source code from another program that references CLIPS rules as it runs. The facts file would support a initial list of facts to be used in testing a CLIPS module, while the current facts are again displayed in a facts list window.

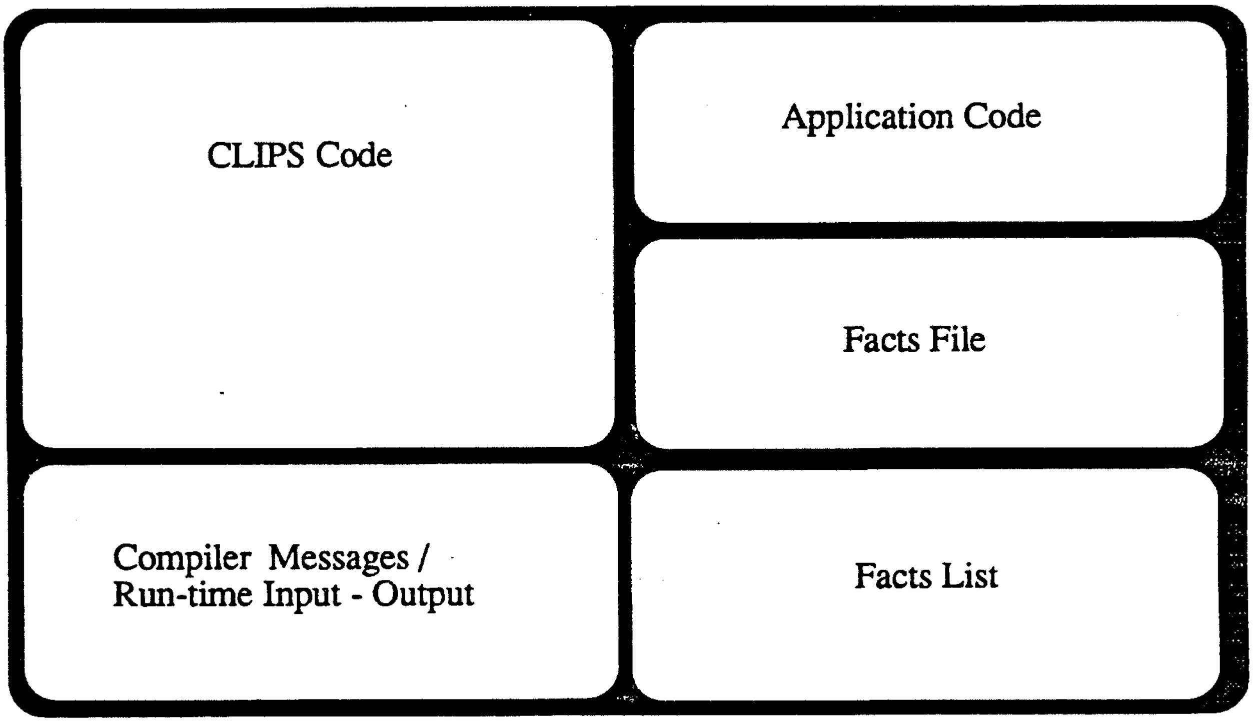


Figure 2: Example of an advanced CLIPS development environment.

Note: Please do not take figures 1 and 2 too literally. Window location and size would be under the user's control. The given arrangement is for the purposes of this discussion only.

Developing a generic user interface for CLIPS across its many platforms and operating systems would be technically challenging. Hardware constraints and portability requirements must certainly be considered. But as platforms become more powerful and as operating systems and as user interface management systems are standardized, ideas like this will become feasible.

Proposed - Changes to the CLIPS Development Environment

The following topics are presented as areas where the current CLIPS PC user interface might be improved upon. Specific recommendations and objective justification will be provided in further discussion of each issue.

Improving the format and content of compiler output.

Understanding compiler output is a critical aspect of learning a new computer language. No one really likes having their errors pointed out to them - especially by a machine. So it is important that compiler statements to the user be clear, accurate, and helpful.

```
a.   Compiling rule: grab-object-from-ladder
     Missing function declaration for defrule  <color highlight>

b.   Compiling rule:  drop-object-once-moved
     An argument in a function call must be a constant, variable, or expression

     ERROR:
     (defrule drop-object-once-moved " "
        ?f1 <- (goal-is-to-move  ?obj  ?place)
        ?f2 <- (monkey  ?place  ?on  ?obj)
        ?f3 <- (object  ?obj  ?  ?  light)
        =>
        (printout t "Monkey drops the  "  ?obj ."

c.   Compiling rule:   hold-object-to-move  +j +j +j +j
```

Figure 3: Examples of clear CLIPS compiler messages.

Figure 3 contains examples of some good CLIPS compiler messages. Notice in Example (a) that the system identifies the rule being compiled, and then follows the message with a statement of the problem. In the version of CLIPS used for this test, the error messages are printed in a separate color from the rest of the text. This is good for the on-line user with a color monitor, but notice that the effect is lost on the printout. The difference between the two types of statements could be further displayed by the use of italics or by flagging the error message with asterisks (**).

In Example (b), the compiler has printed the rule in question, up to the point of the error. This is a good practice, since it clarifies the position of the problem within the rule.

In Example (c), the +j symbols indicate that the rule has been compiled successfully. This allows the user concentrate on other rules that have syntactical problems.

```
1. Compiling rule: grab-object Function retract expected argument #1 to be of type
   number or variable

2. Compiling Region...
   Compiling rule: grab-object-from-ladder
                                                  Expected ')' to finish rule or  '(' to be
   gin new action
                Error:
                     defrule grab-object-from-ladder ""
                                                             ?f1 <-(goal-is-to han
   ds  ?obj)
            ?f2 <- (object ?obj ?place ceiling light)...

3. Compiling rule:  unlock-chest-to-hold-object  +j +j +j

   Expected left parenthesis to begin defrule or deffacts statement
   Compiling rule: hold-chest-to-put-on floor  +j +j +j +j

   Found unrecognized construct...
```

Figure 4: Examples of unclear CLIPS compiler messages.

Figure 4 contains examples of compiler statements that are less clear, less readable, or potentially misleading. In Example 1, the rule name and the error message are not separated, making reading and interpreting the message more difficult.

Example 2 illustrates a very useful feature of the CLIPS compiler - the regional (incremental) compile. A specific section of a CLIPS program can be highlighted and compiled within the editor. This speeds up the compiling process, and allows users to complete and compile "one rule at a time". Notice, however, how difficult it is to distinguish between the error message and the display of the rule due to the awkward spacing of the statements. Ideally, this message would be formatted much like Example (b) in Figure 3.

Example 3 can be difficult for a novice CLIPS user to interpret. What the compiler is trying to say is that 2 rules: *unlock-chest-to-hold-object* and *hold-chest-to-put-on-floor* have compiled correctly, and that two rules (one after *unlock-chest* and one after *hold-chest*) have failed to compile. The rules are not named because they were never recognized as rules by the compiler. While there are some cues in the messages that rules were not compiled, they are not powerful ones. Redundant cues would assist the novice user without distracting the experienced user.

The CLIPS Editing Environment

Experienced programmers and computer users generally have their favorites among the wide variety of editors and word processors that are currently available. CLIPS currently allows the user to choose any standard text editor for preparing code, which permits an individual to select the preferred editing environment.

Many programmers are particularly fond of the EMACS editor, while others do not like it at all. For beginners in the CLIPS environment, EMACS is a poor choice since it requires the user to learn and remember a specific set of commands as they try to learn and remember CLIPS syntax. Doing both of these things at once is a particularly heavy load for the new CLIPS programmer. If a more modern, direct manipulation style editor were offered as an option for beginners, their training time could be reduced. Also, a custom CLIPS editor could have built-in functions that relate specifically to programming in CLIPS, significantly speeding up the typing / coding process. Specific examples of these custom functions will be discussed later.

A Command Storage Buffer and Function Key

One of the most common errors committed by CLIPS users occurs when a relatively long command is typed on the command line. If a typographical error occurs early in the command, and it is not detected immediately, the user is forced to delete the entire line and type the entire command over. This can be quite frustrating, particularly when a long command is in error only because the initial parenthesis is missing.

It would be feasible to store the contents typed on the command line in a buffer associated with a PF key. Essentially, this would permit the user to "edit" and "paste" the contents of the buffer onto the command line. It would also be useful to store a stack of recent commands, allowing users to retain several frequently used commands. These commands could then be pasted on the command line and executed with two keystrokes whenever the user desired. Similar features are available on the DOS command line using the PF3 key.

(Parenthetically Speaking)

On a randomly selected page containing seven CLIPS rules there are 61 sets of parentheses. These represent 122 characters, 244 keystrokes, and about 8% of the characters on this particular page.

Since the CLIPS programmer may spend as much as 10% of his typing time addressing parentheses, some specialized functions to assist in this area might prove quite useful. The "action" menu in the CLIPS PC window is an excellent example of such a function. It will automatically format an "assert" or "retract" command for the user. This is a particularly useful function that would benefit even the experienced CLIPS programmer.

A similar function available in a CLIPS editor would be very useful, reducing the emphasis on typing parentheses and other symbols. An editor function that would place parentheses around a selected block of text would be helpful, too. This idea is closely related to the Command Storage Buffer and Editing issues addressed earlier.

On-line Help.

A strength of the CLIPS PC window user interface is the existence of its on-line help system. One feature of the help system that improves usability is its multi-level nature. Separate help is provided for the PC Window interface, CLIPS, and the help system itself. Since users ask questions at several different levels, this system is more likely to meet the user's needs in many situations.

The help system is well organized for letting the user "browse" through the information provided. This is a strategy that many users employ when learning a new system. By definition, however, a browsable help system generally does not respond well to ad-hoc requests. For example, the CLIPS user who desired information about the "retract" command would have to know (or find out) that this information resides under the menu items "using CLIPS" and "additional commands". A cross-referenced help system could provide help for both user strategies. Ideally, the browsable format would be retained and the system could also provide ad-hoc information in response to a command such as (help retract).

Remember that users often turn to on-line help for a quick answer to a specific question. By the time the CLIPS on-line help system is loaded and the user has mastered its tree structure, the original question may well have been forgotten. It is possible that the user may give up on the help system and turn to another source for assistance.

Dynamic Pull-down Menus and Mouse.

Application of a mouse and menu interface for CLIPS PC was a bold stride toward increasing the usability of the product. Selection by pointing and clicking with the mouse is almost always easier for the novice user. As users become more expert with a system, they tend to learn the keyboard equivalents for commands and spend less time using the menus and mouse.

The implementation of menus and mouse for CLIPS PC is based on the earliest level of technology. Compared to current products, the CLIPS window process is awkward and slow. Windows must be deliberately opened and closed, and selections are an active, very deliberate process. While errors may be less frequent under such conditions, user speed is drastically reduced. Professional programmers tend to prefer the potential for speed in their user interface as opposed to restrictive efforts intended to prevent errors. This would lead the CLIPS PC interface in the direction of the more dynamic mouse and menu technologies available today.

Conclusion

This paper has reviewed the usability of the CLIPS PC window system, pointing out some of its strengths and weaknesses and making some recommendations for possible improvements. It has suggested that the user interface in general move in the direction of a multi-window display. More important than any specific recommendation, however, is the suggestion that the CLIPS user interface be enhanced as its user community directs.

It should be pointed out that CLIPS platforms other than personal computers have had little or no attention paid to the attributes of their user interfaces. This paper has described some basic usability problems and solutions for one platform in an effort to promote the discussion of usability issues for all CLIPS implementations.

N96- 12923

5/0-61

p-5

358515

HyperCLIPS: A HyperCard Interface to CLIPS

Brad Pickering
Randall W. Hill, Jr

Jet Propulsion Laboratory
4800 Oak Grove Drive MS 125-123
Pasadena, CA. 91109

Introduction:

HyperCLIPS combines the intuitive, interactive user interface of the Apple Macintosh®* with the powerful symbolic computation of an expert system interpreter. HyperCard® is an excellent environment for quickly developing the front end of an application with buttons, dialogs, and pictures, while the CLIPS interpreter provides a powerful inference engine for complex problem solving and analysis. By integrating HyperCard and CLIPS the advantages and uses of both packages are made available for a wide range of uses: rapid prototyping of knowledge-based expert systems, interactive simulations of physical systems, and intelligent control of hypertext processes, to name a few.

Interfacing HyperCard and CLIPS is natural. HyperCard was designed to be extended through the use of external commands (XCMDs), and CLIPS was designed to be embedded through the use of the I/O router facilities and callable interface routines. With the exception of some technical difficulties which will be discussed later, HyperCLIPS implements this interface in a straight forward manner, using the facilities provided. An XCMD called "ClipsX" was added to HyperCard to give access to the CLIPS routines: clear, load, reset, and run. And an I/O router was added to CLIPS to handle the communication of data between CLIPS and HyperCard.

Programming in HyperCLIPS:

Programming HyperCLIPS is only slightly more difficult than programming HyperCard and CLIPS separately. The three extra issues that one needs to understand are: how to use the "ClipsX" XCMD; how to use the I/O commands from CLIPS to get information to and from HyperCard; and when and how to pass control of the Macintosh between the CLIPS and HyperCard. The following examples should clarify these issues.

The ClipsX XCMD:

* Apple, Macintosh, and HyperCard are registered trademarks of Apple Computer, Inc.

Example 1: The use of clear.

```
-- in a HyperCard script
  ClipsX "clear"
  get the result
  if char 1 to 3 of it is not "V4." then
    -- this is probably an error
    -- so handle the error and then exit
  end if
  -- continue setting up CLIPS program
```

The "ClipsX" command handles four sub-commands as specified by the first parameter. The first of these commands is "clear". It is used to clear the CLIPS environment. This should be the first CLIPS command called from a HyperCLIPS application stack, so that any other CLIPS program in the interpreter will be excised. If the CLIPS interpreter has not been loaded then it will be loaded at this time. Many things can go wrong while loading the CLIPS interpreter: memory may become full; the file containing the interpreter may not be found; or an incompatible version of the interpreter may be loaded. So it is important to check for these errors. Any data from CLIPS may be retrieved using the HyperTalk function "the result". If everything executes as it should then the first line of the data return will be the version information. This example checks that version four has been loaded.

Example 2: The load and reset commands.

```
-- in a HyperCard script
-- assumes card field "program" contains
-- the following CLIPS program
-- (defrule start
--    (initial-fact)
--    =>
--    (fprintout t "Hello world." crlf))
  ClipsX "load",card field "program"
  ClipsX "reset"
  -- continue setting up CLIPS program
```

The second command typically used is "load". It takes a second parameter which is the text of the CLIPS program to load. The next command is "reset" which sets up the initial facts and activations in the CLIPS environment. Because of how the IO router system is set up, these routines return may return information about which rules were compiled, which facts were asserted, and which rules were activated. But this information is not usually of interest in a HyperCLIPS application so this example does not make use of the data return through "the result". It simply loads a program and makes it ready to run, assuming no errors will occur.

Example 3: The run command.

```
-- in a HyperCard script
-- assumes the CLIPS program from the previous example
-- has been loaded and is ready to run.
  ClipsX "run",empty
  get the result
  -- process the results returned from CLIPS
  get line 1 of it
  answer it with "OK"
```

The last of the four sub-commands to "ClipsX" is "run". This is the most often used command because it passes data and control to CLIPS. It takes a second parameter which is the text of the data you wish to make available to the running CLIPS program. This example passes "empty" as its second parameter because the program that is loaded does not need any extra data to do its computation. The "run" command starts the CLIPS intepreter which does not return until an error occurs or it runs out of rules to fire. In this case the interpreter will fire just the one rule and then return control back to HyperCard. Because of the way the IO router is set up, the message "Hello world." will be returned as the first line of the data returned through "the result". Processing the results usually involves parsing the data and presenting it in an appropriate fashion to the user. This example displays the message in a dialog box. The last line of the data passed back from CLIPS should say how many rules were fired. This information may be useful for debugging purposes but is of little use in the final version of an application.

The I/O router:

Example 1: Sending data back to HyperCard.

```
; in a CLIPS program
(defrule start
  (initial-fact)
  =>
  (fprintout t "Hello world." crlf))
```

This is the example that was used above and you probably already understand what happens, but it will now be explained in greater detail. The I/O router facilities of CLIPS allow the redirection of I/O from one physical location to another. In standard CLIPS, any data written to any of the logical names "stdout", "werror", or "wdisplay" will probably be written to the terminal. Whereas in a windowing version of CLIPS the data will probably be written to three different windows. This is managed by routing data sent to these logical names to different locations in each case. The HyperCLIPS I/O router handles data written to all of the standard logical names by collecting and buffering it and then passing it back to HyperCard as "the result" when the CLIPS interpreter returns. This means that in the example above the fprintout statement, which writes a message to "stdout", will make the message "Hello world." available to HyperCard when the run command completes.

Example 2: Receiving data from HyperCard

```
; in a CLIPS program
; assumes a HyperCard call such as ClipsX "run","broken"
; also assumes that this rule is on the activation list so
; that it will be fired when the run command is called
(defrule get_engine_state
  ?fact <- (get_state)
  =>
  (retract ?fact)
  (bind ?state (read))
  (assert (engine_state ?state)))
```

Receiving data from HyperCard is also handled through the I/O router system. The standard version of CLIPS normally reads data from the terminal. The HyperCLIPS I/O router reroutes reads from the "stdin" logical name (the default read location) to get characters from a memory buffer instead of the terminal. When the "ClipsX" "run"

C-2

command is called the second parameter is used to fill in this buffer. This example will read the word "broken" from the buffer and then assert the fact "engine_state broken".

Passing control between CLIPS and HyperCard:

Example 0: Passing control to CLIPS

```
-- no example needed
```

HyperCard and CLIPS do not execute concurrently. Control must be explicitly passed between the two whenever either of them needs the functionality of the other. Control is usually passed to CLIPS when HyperCard needs a computation performed. This is done with the "run" command. The CLIPS program, though, must be ready to accept control. This means that there are rules on the activation list ready to fire. Initially rules are put on the activation list by the "reset" command, but there is another method to get CLIPS ready to accept control which will be explained next.

Example 1: Passing control to HyperCard

```
; in a CLIPS program
(defrule get_data
  ?f < - (phase get_data)
  =>
  (retract ?f)
  (fprintout t "need data" crlf)
  (assert (get_data_continue))
  (halt))

(defrule get_data_continue
  ?f <- (get_data_continue)
  =>
  (retract ?f)
  (bind ?data (read))
  (assert (data ?data)))
```

Control is usually passed back to HyperCard for one of two reasons: the computation is finished; or more data is needed to complete the computation. If the computation is finished then passing control back to HyperCard is trivial: there will be no more rules to fire so CLIPS will return automatically. The case of needing more data, though, is more complex. This example.shows how to give control back to HyperCard while making sure that the rule that reads the data will be ready to fire when HyperCard eventually returns control back to CLIPS. The important CLIPS function is "halt". It causes an error within CLIPS so that the interpreter will return to HyperCard, but it does not alter the activation list so that any rule that was ready to fire before the "halt" command will still be ready to fire after the "halt" command. In this way the CLIPS program is ready to accept control when HyperCard calls the "run" command with the data needed to continue the computation.

Technical difficulties implementing HyperCLIPS

Although HyperCard and CLIPS seem easily integrated through the use of their built in hooks for such reasons, there are some technical problems which make this task more difficult that it would appear. The problem is on the HyperCard side. HyperCard allows the addition of functionality in the form of XCMDs, but XCMDs have severe limitations: an XCMD cannot be larger than 32K bytes, and an XCMD cannot have global data. CLIPS breaks both of these rules and cannot, therefore, be implemented as a normal XMCD.

Both of these limitations are the result of the architecture of the Macintosh. A Macintosh application uses register A5 of the Motorola 68000 to point to the area of memory that contains the global variables and the jump table. The jump table is used to support intra-segment calls which are necessary because segments are limited to 32K and any application larger than this must be divided into multiple segments. Segments are limited in size by the longest possible branch instruction, which on the 68000 is +/- 32K. Jump instructions could be used to allow farther branches and larger segments, but this would make the code non-relocatable which is contrary to the Macintosh memory management strategy. While HyperCard is in control, register A5 points to HyperCard's global data and jump table. XCMDs cannot use this jump table or global data area, this leads to the limitations mentioned above.

The way to get around the two limitations mentioned above is obvious but tricky to implement: let the XCMD have its own jump table and globals area and make A5 point to this area while the XCMD is running. The difficulty in this is in setting up the jump table. This process is usually handled by the Segment Loader facility in the Macintosh Operating System. It interprets the information in CODE resource 0 of the application to form the jump table and globals area and then starts the program by jumping to the first entry in the jump table.

The implementation of HyperCLIPS is divided into two parts: an XCMD that duplicates the functionality of the Segment Loader and takes care of setting up the A5 register before calling the CLIPS interpreter; and a modified CLIPS interpreter stored in the format of an application file where the XCMD can find it. The only modifications to CLIPS are the code to handle the function dispatching and the I/O router to handle the communication of data.

Conclusion:

We have used HyperCLIPS to develop prototypes for device simulation and knowledge based training systems. In our experience we have found development time to be very fast. The CLIPS side of an application can be developed and debugged in the usual CLIPS environment and later be integrated with a HyperCard user interface. This final stage of integration is a little awkward because of the lack of tools for modifying CLIPS programs from within HyperCard, but we are adopting methodologies to make this step easier. Because HyperCard and CLIPS are interpreted languages, execution time for HyperCLIPS applications is rather slow. In the case of CLIPS, the results may be worth the wait, but HyperCard may need to replaced by a more efficient user interface engine in production quality applications. If a faster interface becomes necessary though, the substitution should be transparent to the CLIPS side of the application. Our future plans include looking for such an interface engine, possibly on a more powerful workstation.

omit

A9 Session:
Space Shuttle and Satellite Applications

A DYNAMIC SATELLITE SIMULATION TESTBED BASED ON CLIPS AND CLIPS-DERIVED TOOLS

Thomas P. Gathmann
Rockwell International
Satellite & Space Electronics Division
Seal Beach, CA

Problem

Current day spacecraft are complex machines and those on the drawing boards are increasingly more sophisticated and broader in scope. Gone are the days when a single engineer could fully grasp the intricacies of an entire satellite. Note that the recently launched Galileo spacecraft has several processors on-board the vehicle [1]. This fact, coupled with the increasing power of computing hardware and software tools and techniques, has introduced the possibility of realistic simulations being used for product definition, design, manufacture, and, even, performance analysis. The Strategic Defense Initiative Office (SDIO) is convinced of simulation capabilities since it has funded the National Test Bed (NTB) facility to evaluate the performance of all facets of the "Star Wars" concept.

Due to heated competition for the development and delivery of satellites, there is an increased reliance on simulation of components, subsystems, systems, and entire constellations of spacecraft. Given the wide variety of configurations and purposes of these satellites, flexible and convenient means for generating study and engineering data are necessary [2-4]. Monolithic simulations have become unwieldy and expensive to maintain. Configurable tools that can be quickly and accurately constructed are required. Rapid prototyping techniques have become more accepted within the aerospace industry for the production of deliverable software and also as a means to manage the software process [5].

We were motivated to define and build a sophisticated satellite simulation capability for the evaluation of a satellite operations automated environment called IntelliSTAR™ [6,7]. This architecture, and associated prototype, addresses the entire spacecraft operations cycle including planning, scheduling, task execution, and analysis. It is aimed at increasing the autonomous capability of current and future spacecraft. It utilizes advanced software techniques to address incomplete and conflicting data for making decisions. It also encompasses critical response time requirements, complex relationships among multiple systems, and dynamically changing objectives. Given the extreme scope of activities that are targeted, a sophisticated, flexible, and dynamic simulation environment was required to drive this prototype. In particular the derived requirements for evaluating the IntelliSTAR™ prototype include:

- provide realistic and dynamic environment
- easily reconfigurable
- multiple levels of fidelity

The overriding need of IntelliSTAR™ was a means for providing a valid evaluation of the concept (see Figure 1). This evaluation was planned to be accomplished through the injection of various scenarios describing mission and behavior types for the spacecraft to be controlled. Given this stimulus, the IntelliSTAR™ prototype provides measures of the plan and its status to satisfy the objectives for the satellite mission.

Approach

The testbed approach to simulation has risen to the top of the list of options due to the following attributes:

- flexibility to easily configure based on unique customer requirements
- modularity of the simulation components to allow the testing of portions of the overall system or varying degrees of fidelity for portions within the same simulation
- interoperability through the use of consistent user and integrator interfaces for reduced training.

Side benefits include the centralized storage and accumulation of metrics and related information of the simulation capabilities and past usage of the testbed.

Our approach to the development, utilization, and maintenance of a sophisticated satellite simulation testbed is the use of rapid prototyping and knowledge-based techniques coordinated with the use of existing simulation and communication resources. An architecture has been defined that provides the following attributes for a spacecraft simulation that addresses autonomy, surveillance, and survivability capabilities (see Figure 2):

- integrating architecture that supports the expansion of capabilities and resources
- high-level user interface for speci-

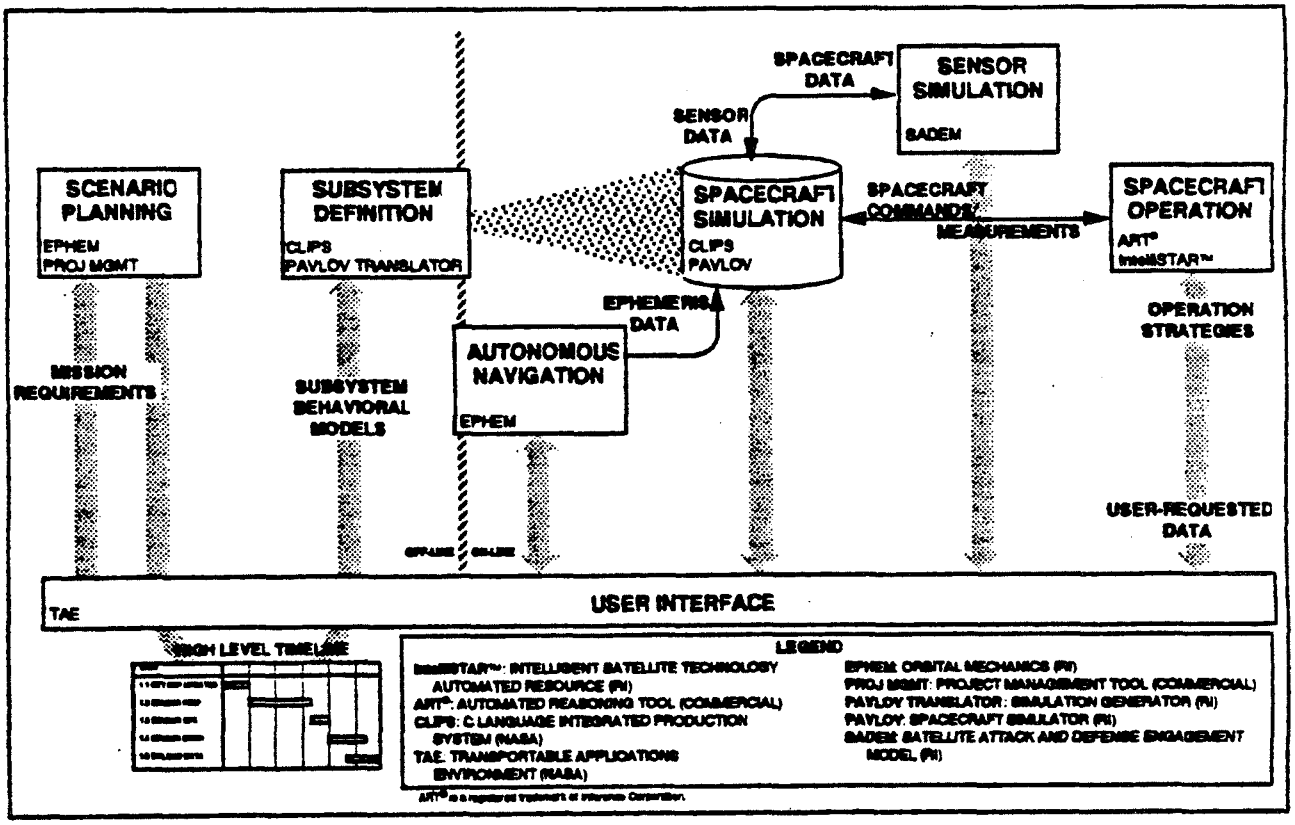

SPACECRAFT DATA
SENSOR SIMULATION
SADEM
SENSOR DATA
SCENARIO PLANNING
EPHEM
PROJ MGMT
SUBSYSTEM DEFINITION
CLIPS
PAVLOV TRANSLATOR
SPACECRAFT SIMULATION
CLIPS
PAVLOV
SPACECRAFT COMMANDS/
MEASUREMENTS
SPACECRAFT OPERATION
ART®
IntelliSTAR™
EPHEMERIS DATA
AUTONOMOUS NAVIGATION
EPHEM
OPERATION STRATEGIES
MISSION REQUIREMENTS
SUBSYSTEM BEHAVIORAL MODELS
OFF-LINE
ON-LINE
USER-REQUESTED DATA
USER INTERFACE
TAE
HIGH LEVEL TIMELINE
LEGEND
IntelliSTAR™: INTELLIGENT SATELLITE TECHNOLOGY AUTOMATED RESOURCE (RI)
ART®: AUTOMATED REASONING TOOL (COMMERCIAL)
CLIPS: C LANGUAGE INTEGRATED PRODUCTION SYSTEM (NASA)
TAE: TRANSPORTABLE APPLICATIONS ENVIRONMENT (NASA)
EPHEM: ORBITAL MECHANICS (RI)
PROJ MGMT: PROJECT MANAGEMENT TOOL (COMMERCIAL)
PAVLOV TRANSLATOR: SIMULATION GENERATOR (RI)
PAVLOV: SPACECRAFT SIMULATOR (RI)
SADEM: SATELLITE ATTACK AND DEFENSE ENGAGEMENT MODEL (RI)
ART® is a registered trademark of Inference Corporation.

Figure 1. The Satellite Autonomy Application Evaluation Testbed Maximizes the Use of Existing and Modified Tools While Providing Flexibility and Realism.

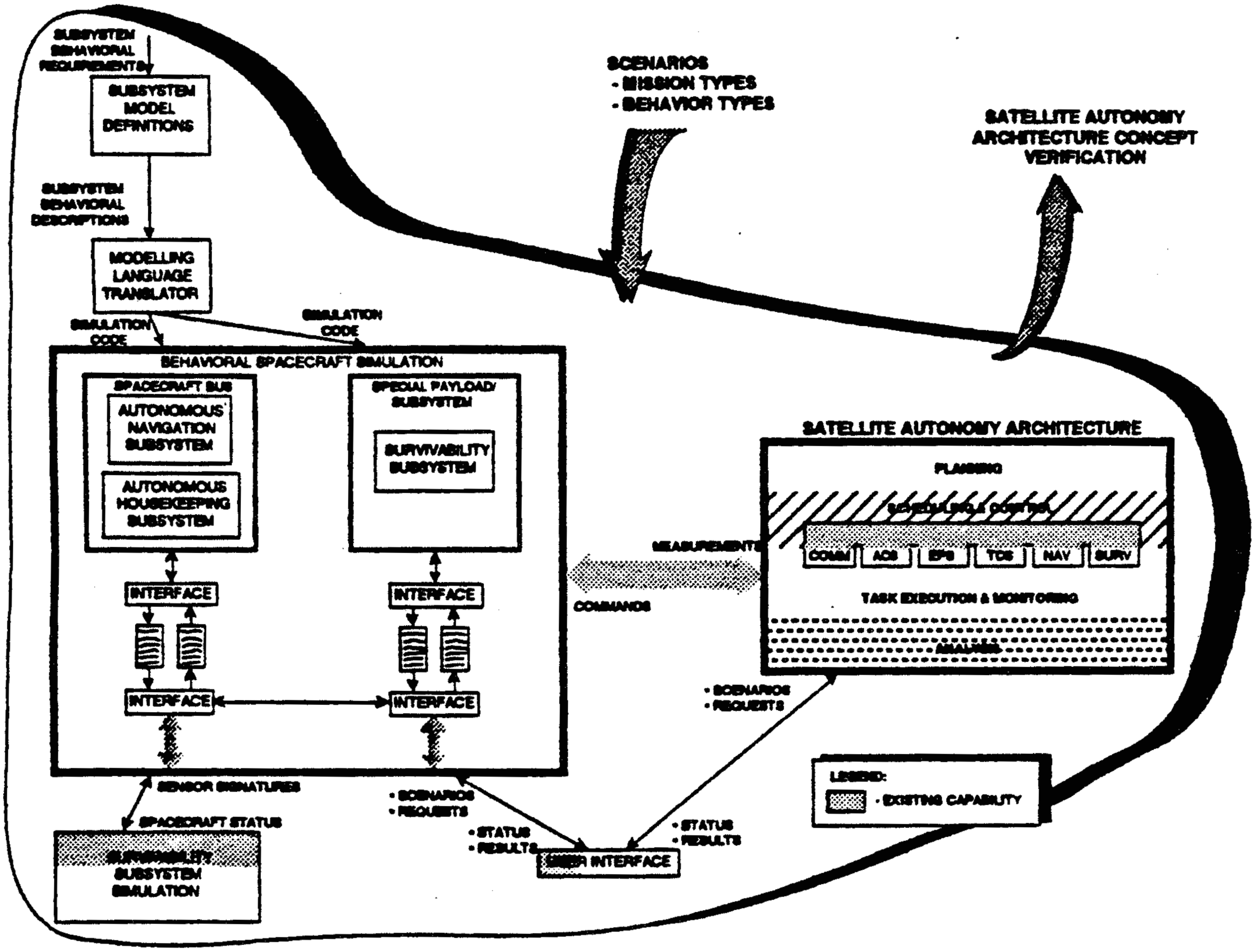

SUBSYSTEM BEHAVIORAL REQUIREMENTS
SUBSYSTEM MODEL DEFINITIONS
SUBSYSTEM BEHAVIORAL DESCRIPTIONS
MODELLING LANGUAGE TRANSLATOR
SIMULATION CODE
SIMULATION CODE
BEHAVIORAL SPACECRAFT SIMULATION
SPACECRAFT BUS
AUTONOMOUS NAVIGATION SUBSYSTEM
AUTONOMOUS HOUSEKEEPING SUBSYSTEM
SPECIAL PAYLOAD/ SUBSYSTEM
SURVIVABILITY SUBSYSTEM
INTERFACE
INTERFACE
INTERFACE
INTERFACE
SENSOR SIGNATURES
SPACECRAFT STATUS
SURVIVABILITY SUBSYSTEM SIMULATION
• SCENARIOS
• REQUESTS
• STATUS
• RESULTS
USER INTERFACE
• STATUS
• RESULTS
• SCENARIOS
• REQUESTS
SCENARIOS
- MISSION TYPES
- BEHAVIOR TYPES
SATELLITE AUTONOMY ARCHITECTURE CONCEPT VERIFICATION
SATELLITE AUTONOMY ARCHITECTURE
PLANNING
SCHEDULING & CONTROL
MEASUREMENTS
COMMANDS
COMM
ACS
EPS
TCS
NAV
SURV
TASK EXECUTION & MONITORING
ANALYSIS
LEGEND:
▨ - EXISTING CAPABILITY

Figure 2. An Integrating Architecture Has Been Defined and Developed to Provide Validation for a Satellite Operations Prototype.

fying simulation requirements and features in the form of a modelling language

- automated translator from the modelling language to CLIPS code which can be executed
- separability of generic spacecraft features from specialized components, subsystems, and payloads
- interface to an existing survivability simulation
- interface to an existing intelligent satellite operations framework
- interface to a graphical user interface

Architecture Description

We are using CLIPS as our basic programming language to create the modelling language, language translator, and simulation itself. The modelling language allows an engineer to specify the behavior of a system or subsystem in high-level terms that could be directly derived from specifications. The translator takes the modelling language constructs, verifies their consistency, and creates CLIPS knowledge bases which can be executed. The simulation uses the CLIPS forward-chaining mechanism as the driving force behind a system that is scalable to real-time events. Time can be specified directly or used in relative terms to compress or expand time to meet user requirements.

Satellite Modelling Language (SML)

The modelling language was created to provide a higher level interface to the identified end-user, a spacecraft design engineer. This interface allows the engineer to input requirements and features in a format which is familiar. This promotes a more rapid acceptance and utilization of the testbed resource resulting in increased productivity and the exploration of a larger number of engineering options.

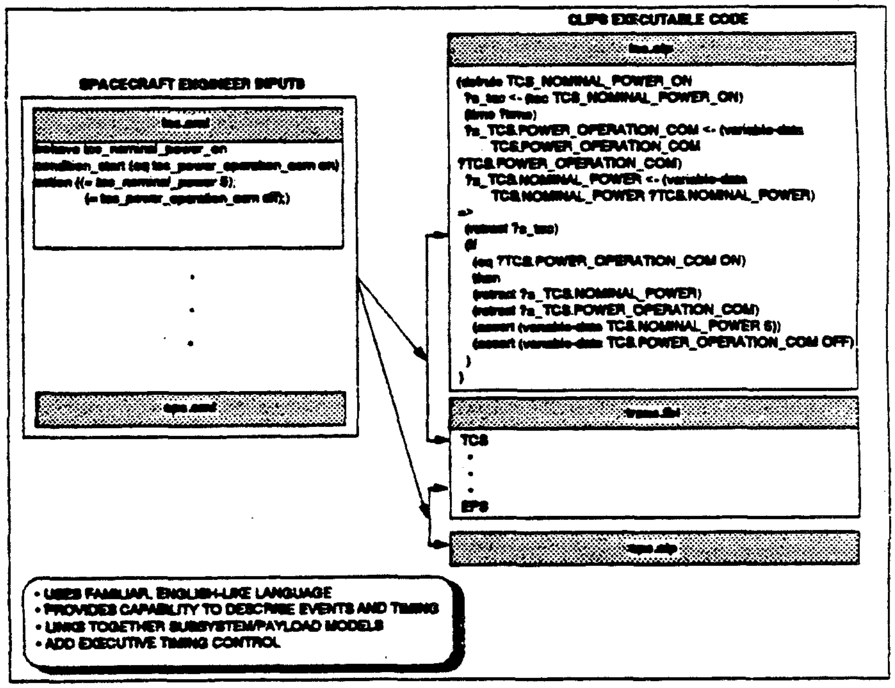
SML provides context-relevant and English-like language constructs to the spacecraft engineer. Through these constructs, the capability to describe events and timing is provided. This is accomplished through the use of three main structure types: templates, objects, and rules. Templates define conglomerations of objects, objects relate to physical or functional entities, and rules describe the behavior of the objects for various conditions.

The simulation itself uses CLIPS' forward-chaining technique to create a reactive and dynamic model of a spacecraft in its orbiting environment. Since spacecraft typically operate in a data- and situation-driven environment, CLIPS is a perfect match. Processes on a satellite are usually invoked on either a time or event basis. The stimuli cascade through many devices and components to achieve the necessary and required states. Side effects of component actions are relied upon heavily on spacecraft. These factors closely match the advantages of a system built with CLIPS.

Modelling language translator

The modelling language translator accepts the simulation specification from the engineer and converts it into CLIPS knowledge bases which can be executed (refer to Figure 3). This circumvents the need for the spacecraft engineer to become familiar with a new, and probably very different, software language. Also, since the CLIPS simulation code is automatically generated, the proper syntax and semantics are maintained within the knowledge bases. CLIPS is being applied in a manner much like an information compiler.

The translator accepts the SML constructs and converts them into CLIPS-acceptable syntax. Templates and objects are converted to facts while behavior rules translate into CLIPS rules. The CLIPS rules handle all the



```
behave tcs_nominal_power_on
condition_start (eq tcs_power_operation_com on)
action ((= tcs_nominal_power 5);
        (= tcs_power_operation_com off);)
```

```
(defrule TCS_NOMINAL_POWER_ON
 ?s_tcs <- (tcs TCS_NOMINAL_POWER_ON)
 (time ?time)
 ?s_TCS.POWER_OPERATION_COM <- (variable-data
     TCS.POWER_OPERATION_COM
?TCS.POWER_OPERATION_COM)
 ?s_TCS.NOMINAL_POWER <- (variable-data
     TCS.NOMINAL_POWER ?TCS.NOMINAL_POWER)
=>
 (retract ?s_tcs)
 (if
  (eq ?TCS.POWER_OPERATION_COM ON)
  then
  (retract ?s_TCS.NOMINAL_POWER)
  (retract ?s_TCS.POWER_OPERATION_COM)
  (assert (variable-data TCS.NOMINAL_POWER 5))
  (assert (variable-data TCS.POWER_OPERATION_COM OFF))
 )
)
```

```
TCS
 •
 •
 •
EPS
```

- USES FAMILIAR, ENGLISH-LIKE LANGUAGE
- PROVIDES CAPABILITY TO DESCRIBE EVENTS AND TIMING
- LINKS TOGETHER SUBSYSTEM/PAYLOAD MODELS
- ADD EXECUTIVE TIMING CONTROL

Figure 3. The Translator Capability Permits an Incremental Construction of a Spacecraft Simulation.

bookkeeping involved with the behavior such as retracting facts after they are no longer required and asserting the pertinent facts.

The translator permits the incremental construction of a complete simulation capability. In practice, the modules are aligned with the subsystem designs. For instance, the Thermal Control Subsystem (TCS) templates, objects, and behavior rules are all defined within a single file. The translator maintains a list of all possible constructs and allows the linking of these in any manner specified by the user. The linking procedure also adds the executive timing control to the executable simulation.

Satellite simulation

The satellite simulation generation methodology is represented in Figure 4. Two parallel development paths have been identified for the creation of a realistic and dynamic evaluation environment for the IntelliSTAR™ prototype. One path concentrates on the creation of behavioral models while the other generates functional simulation capabilities. Behavioral models take the "black box" approach to testing. Functional models are analogous to the "white box" approach. This approach is justified by remarks such as the following:

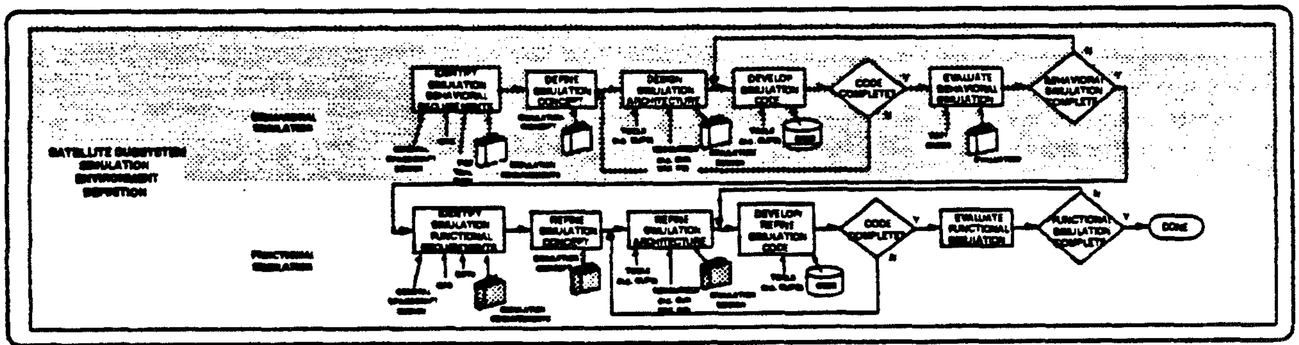

SATELLITE SUBSYSTEM SIMULATION ENVIRONMENT DEFINITION
BEHAVIORAL SIMULATION
IDENTIFY SIMULATION BEHAVIORAL REQUIREMENTS
DEFINE SIMULATION CONCEPT
DESIGN SIMULATION ARCHITECTURE
DEVELOP SIMULATION CODE
CODE COMPLETE?
Y
N
EVALUATE BEHAVIORAL SIMULATION
BEHAVIORAL SIMULATION COMPLETE?
FUNCTIONAL SIMULATION
IDENTIFY SIMULATION FUNCTIONAL REQUIREMENTS
REFINE SIMULATION CONCEPT
REFINE SIMULATION ARCHITECTURE
DEVELOP REFINE SIMULATION CODE
EVALUATE FUNCTIONAL SIMULATION
FUNCTIONAL SIMULATION COMPLETE?
DONE

Figure 4. An Incremental Approach to the Construction of a Realistic Evaluation Environment Encourages Both the Up-front Definition and Near-term Capability Generation.

> *"Black box testing is not an alternative to white box techniques. It is, rather, a complementary approach that is likely to uncover a different class of errors than white box methods."* [8]

Behavioral models permit the description of the inputs and outputs of a function (or process or subsystem or ...). These models permit an empirical or high-level description of an entity. These models can be constructed quickly with readily available information and allow various levels of detail.

Functional models require an extensive evaluation of the theories and principles behind the operation of an entity. These models result from the classical design phase of an engineering process. Functional models have typically been developed in a monolithic mode. Good examples of functional model implementations are the current Computational Fluid Dynamics (CFD) codes being constructed.

The combination of these two simulation methods allows the generation of realistic environments quickly while not negating the growth path to more robust and in-depth simulation. In fact, the overall evaluation architecture permits the injection of models of varying fidelity levels into the same simulation. Behavioral and functional model can co-exist in the architecture. This provides a flexible medium for testing of the

IntelliSTAR™ prototype. In addition, the evaluation environment is not strictly tailored to that prototype, but also permits the construction of any satellite models.

The test architecture encourages a modular generation and management of its constituent parts. A conscious design decision was made to make the generic satellite bus characteristics separable from the specialized subsystems or payloads that comprise a spacecraft. By doing so, a generic capability for simulating spacecraft was created. This model will continue to evolve and the available "library" of models will increase as this effort proceeds. In fact, a major satellite effort at our division is contemplating the use of this capability because of the attractiveness of minimal cost to tailor the system for their purposes. Our research can continue in parallel with this satellite application since models can be interchanged with little effort.

Interfaces

Three types of interfaces currently exist to the simulation environment. These include one to the IntelliSTAR™ prototype, one to an existing survivability simulation, and the last to a user interface capability. The mechanism used for all three interfaces is the same; the results of a generic, distributed process communications project are utilized.

The interface to the IntelliSTAR™ prototype is implemented to allow the evaluation of this satellite operations concept. The interactions between the prototype and the simulation are of two types: continuous and requested. The first type, continuous, contains the telemetry stream content from the spacecraft to the controlling entity (i.e., IntelliSTAR™). The information flow is handshaked between the two portions but the interface is not truly synchronous. IntelliSTAR™ provides an execution time frame to the simulation and the simulation responds for that amount of time or at some smaller increment. The response time is solely determined by the simulation with only the upper bound specified by the prototype.

The second type of interface to IntelliSTAR™ is closer to being of the synchronous variety. A request is made of the simulation for information and the simulator responds with the derived data. The prototype may or may not wait for the results of its query before proceeding with its processing.

An interface with an existing survivability simulation (SADEM - Satellite Attack and Defense Engagement Model) was constructed. SADEM is constructed in an object-oriented and distributed environment. SADEM schedules a communications event to the spacecraft simulation at either a time or based on some condition. Currently, this interface is only one-way due to a limitation in the SADEM development environment.

The last interface is to the user interface module. This interface allows the control and execution monitoring of the simulation. Individual measurements being generated by the simulator may be presented with user-specified limits. Graphical representations of the data are allowed.

Conclusions

The simulation environment allows the integration of several levels of fidelity and the configuration of many diverse components. The modelling language translator assures the consistent generation of syntactically and semantically correct spacecraft simulations. The "garbage in, garbage out" syndrome of many simulations is minimized through the active application of knowledge about spacecraft in general. This approach, and associated testbed development, enables the creation of a sophisticated and consistent satellite simulation environment used for the design, manufacture, and analysis of satellites and their related operations environments.

References

[1] *"Galileo Represents Peak in Design Complexity".* Aviation Week and Space Technology, Oct. 9, 1989.

[2] Mitchell, Robert R., *"Expert Systems and Air Combat Simulation".* AI Expert, September 1989.

[3] Rao, Nageswara S.V., *"Algorithmic Framework for Learned Robot Navigation in Unknown Terrains".* IEEE Computer Magazine, June 1989.

[4] Brown, Marc H., *"Exploring Algorithms Using Balsa-II".* IEEE Computer Magazine, May 1988.

[5] Boehm, Barry W., *"A Spiral Model of Software Development and Enhancement".* EEE Computer Magazine, May 1988.

[6] Gathmann, T.P., L. Raslavicius, and J.M. Barry, *"A Unified Concept for Spacecraft System- and Subsystem-level Automation".* 24th Intersociety Energy Conversion Engineering Conference (IECEC-89), Aug. 6-11, 1989.

[7] Gathmann, T.P., and L. Raslavicius, *"Satellite Autonomy Generic Expert Systems (SAGES)".* AIAA Computers in Aerospace, Oct. 4-6, 1989.

[8] Pressman, Roger S., *"Software Engineering - A Practioner's Approach".* McGraw-Hill Company, 1987.

Analysis of MMU FDIR Expert System

Dr. Christopher Landauer
Computer Science and Technology Subdivision
The Aerospace Corporation

April 30, 1990

Abstract

This paper describes the analysis of a rulebase for fault diagnosis, isolation, and recovery for NASA's Manned Maneuvering Unit (MMU). The MMU is used by a human astronaut to move around a spacecraft in space. In order to provide maneuverability, there are several thrusters oriented in various directions, and hand-controlled devices for useful groups of them. The rulebase describes some error detection procedures, and corrective actions that can be applied in a few cases.

The approach taken in this paper is to treat rulebases as symbolic objects and compute correctness and "reasonableness" criteria that use the statistical distribution of various syntactic structures within the rulebase. The criteria should identify awkward situations, and otherwise signal anomalies that may be errors. The rulebase analysis algorithms are derived from mathematical and computational criteria that implement certain principles developed for rulebase evaluation. The principles are *Consistency*, *Completeness*, *Irredundancy*, *Connectivity*, and finally, *Distribution*.

Several errors were detected in the delivered rulebase. Some of these errors were easily fixed. Some errors could not be fixed with the available information. A geometric model of the thruster arrangement is needed to show how to correct certain other distribution anomalies that are in fact errors.

The investigations reported here were partially supported by The Aerospace Corporation's Sponsored Research Program. The author would like to thank the members of the Vehicles Project at Aerospace for a continual stream of hard questions, and Chris Culbert of NASA JSC for providing the rulebase and the challenge to analyze it.

1 Introduction

This paper describes the analysis of an application rulebase for fault diagnosis. The rulebase describes fault detection procedures, experimental procedures to isolate the faults to particular components, and corrective actions that can be applied in a few cases.

The rulebase analysis algorithms are derived from mathematical and computational criteria that implement certain "correctness" principles developed for rulebase evaluation. The principles are *Consistency*, *Completeness*, *Irredundancy*, *Connectivity*, and finally, *Distribution*. Several errors were detected in the delivered rulebase.

An alternative to the systematic analyses above is a model-based validation, which uses several explicit models of system behavior to analyze the behavior of the rulebase that purports to describe the system. This technique is complementary to the systematic criteria, and tends to find different kinds of errors. In fact, each different style of analysis finds somewhat different errors, and it is the recommendation of this paper that many different V&V analyses be performed on any critical rulebase. A geometric model of the thruster arrangement could be used to show how to correct certain other distribution anomalies that are in fact errors.

1.1 Manned-Maneuvering Unit

The Manned Maneuvering Unit (MMU) is essentially a backpack unit for moving a human astronaut around a spacecraft in space. In order to provide maneuverability, there are several thrusters oriented in various directions, and Hand Control Devices for useful groups of them. The thrusters use Nitrogen Dioxide (NO_2) gas for motion.

The FDIR rulebase (see [Lawler,Williams]) is concerned with the problem of fault diagnosis, isolation, and recovery (FDIR) for the MMU. Its purpose is to determine whether the MMU has a fault, to isolate the fault to a particular subsystem when possible, and to take corrective action when that is possible. The rulebase has 104 rules, written in the expert system shell CLIPS (see [Culbert]), the C Language Interface to Production Systems, developed at NASA's Johnson Space Center. No external functions are called (CLIPS allows externally provided functions to be invoked during hypothesis examination and conclusion generation), so the CLIPS code is self-contained. The MMU FDIR rulebase was kindly provided to us by Chris Culbert of NASA JSC, as was CLIPS.

The rulebase was analyzed according to many of the criteria discussed in the next section. There was no automatic version of any of the analyses, since the criteria are not yet implemented in programs. The criteria were applied by hand, using editors, pattern searching programs, and other text manipulation programs generally available under UNIX. For this rulebase, some extra semantic information is available, such as the symmetry between side a and side b. This information was very useful in the analyses.

2 Principles of Rulebase Correctness

This section describes the correctness principles used for the analysis (see [Landauer89], [Landauer90] for more discussion). The five principles are accompanied by mathematical and computational criteria that serve as specifications of analysis algorithms for rulebases. The *Consistency* criteria address the logical consistency of the rules, and can rightly be considered as "correctness" criteria. The *Completeness* and *Irredundancy* criteria preclude oversights in specifications and redundancy in the rules, and are more like "reasonability" criteria for the terms in the rules. The *Connectivity* criteria concern the inference system defined by the rules, and are like completeness and irredundancy criteria for the inference system (see [Bellman,Walter], [Bellman] for arguments that redundancy in rulebases is dangerous, not just wasteful). Finally, the *Distribution* criteria are "esthetic" criteria for the simplicity of the rules and the distinctions they cause, and the distribution of the rules and the values implied by them.

The approach taken in this section is to treat rulebases as mathematical objects and develop criteria for acceptability, both correctness criteria and "reasonableness" criteria. The criteria should identify inconsistent or awkward rule combinations.

2.1 Rulebase Definitions

A rulebase is a finite set R of pairs

$$r \quad = \quad (hyp, conc)$$

of assertions (or formulas), to be interpreted as

if hypothesis ***hyp***, then conclusion *conc*.

The first component (the hypothesis) of a rule r is written $hyp(r)$ and the second one (the conclusion) as $conc(r)$ when there is need to refer to them separately. Each of these parts is considered to be a Boolean function.

The set V of variables in a rulebase R is finite. A ***situation*** is an instantiation of all of the variables, with the further restriction that all the rules are true of all situations. Every variable is considered to be a feature of the situation, with a possibly unknown value in the appropriate domain. The rest of this section will explain what this restriction means.

Each variable v is considered to be a function applied to situations, so for a situation s, the expression $v(s)$ denotes the value of the variable v in situation s. More generally, for any expression e over a set W of variables contained in V, $e(s)$ denotes the value of the expression in situation s.

The set of situations is therefore a subset of the Cartesian product of all of the variable domains, but the particular subset is not precisely known, since it is limited by the rulebase to only those elements of the Cartesian product that satisfy the rules (i.e., the rules define the situations). There are connections between variables that allow some of them to be computed from others. The Cartesian product will occasionally be called the ***situation space***, to distinguish it from the set of situations. An element of the situation space may be called a ***prospective*** situation until it is determined whether it is actually a situation or not. So the syntactic restriction of having each variable value in the appropriate domain suffices to define the prospective situations, and the semantic restriction that all rules are satisfied defines those prospective situations that are situations.

A rulebase is applied to a situation to compute some variable values (not to set the values, but to find out what the values are), so that a situation has both provided variable values ("input" variables) and derived variable values, some of which are displayed ("output" variables). It is further assumed that the variable values not specified by the input are defined but unknown, and that the rulebase is expected to compute the output variable values.

Rules are implicitly universally quantified over situations. A variable v in the rulebase is a fixed component selection function v applied to a variable situation s. There are no explicit quantifiers, so all situation variables are free in the expressions.

2.2 Analysis Tools

This section describes several derived combinatorial objects and other analytical tools that are useful for analyzing a rulebase. They are primarily graph theoretical notions, including graphs and incidence matrices.

2.2.1 Incidence Matrices

The simplest incidence matrix of a rulebase is called simply *the incidence matrix* of the rulebase. It is indexed by $R \times V$, with entry 1 when variable v occurs in rule r (the occurrences must be free, which is easy now when there are no quantifiers).

It is often convenient to retain the number of occurrences of variables in rules. The *counting* incidence matrix RV of a rulebase is a matrix indexed by $R \times V$, with

$$RV(r, v) \quad = \quad \text{number of occurrences of variable } v \text{ in rule } r,$$

so it may have counts greater than one.

The only non-trivial operation that can be performed on this matrix is multiplication. Since there is only one matrix at present, it must be multiplied by itself. Since the coordinate index sets are not the same, either one of the matrix factors must be transposed (giving actually two different products). The only remaining question is what the products might mean. It turns out to be relatively easy to interpret both of them.

With this matrix RV, the (v, w) entry of the product, $(RV^{tr}\ RV)(v, w)$, is the number of pairs of instances of variable v and variable w contained in the same rule, and the (q, r) entry of the product, $(RV\ RV^{tr})(q, r)$, is the number of pairs of instances of rule q and rule r containing the same variable.

The two matrix products above give rise to two undirected graphs, the first one with variables as vertices, and edges for nonzero entries in the product $(RV^{tr}\ RV)$, and the second with rules as vertices, and edges for nonzero entries in the product $(RV\ RV^{tr})$. The first graph connects two variables if they appear together in a rule, and the second one connects two rules if they have common variables. More detailed graphs will be studied later on, but all will use the same basic construction.

There are several other incidence matrices that are useful for rule analyses, including a clause-variable incidence matrix CV, and a rule-clause incidence matrix RC, but they are analogous to the rule-variable incidence matrix RV and are not described in detail. For this purpose, a clause can be considered as a predicate expression, and C is the set of clauses.

2.2.2 Clause Graphs

The inference C graph has vertices for all clauses c, and an edge from clause c to clause d whenever there is a rule r with $c \in hyp(r)$ and $d \in conc(r)$. The inference R graph has vertices for all rules r, and an edge from rule q to rule r whenever there is a clause c which is in both $hyp(r)$ and $conc(q)$. These graphs are defined from the counting incidence matrices to have labels according to the appropriate counts.

2.2.3 Association Matrices

An association matrix is a covariance matrix computed from occurrence patterns across a set of possible locations. The counting incidence matrix product $(RV)(RV^{tr})$ counts variables in common to rules, measuring the occurrence pattern of a rule according to the variables it contains. Then the correlations can be computed from the covariances, in the usual way:

$$\begin{aligned} Corr(q,r) &= Covar(q,r)/(Stdev(q) * Stdev(r)), \\ Stdev(q) &= \sqrt{Covar(q,q)}, \\ Covar(q,r) &= (RV\ RV^{tr})(q,r)/|V| - Avg(q) * Avg(r), \\ Avg(q) &= \sum(\text{variables } v \in V)\ RV(q,v)/|V|. \end{aligned}$$

Here, the q row of the counting incidence matrix RV is the occurrence pattern for rule q, so $Avg(q)$ is the average number of occurrences of each variable in rule q, and $Stdev(q)$ is the standard deviation. There is no random variable here, so there is no point in using the "sample standard deviation". The correlation is a measure of similarity between rules, as measured by the variables in them. The correlation value is 1 if and only if the two rules use exactly the same variables with the same frequency of occurrence of each variable. It will be negative, for example, when the two rules use disjoint sets of variables, and -1 in rare cases only (not likely in a rulebase).

Similarly, the counting incidence matrix product $(RV^{tr})(RV)$ counts rules in common to variables, measuring the occurrence pattern of a variable according to the rules containing it. Correlations are computed as before. Other incidence matrices for variables in clauses and clauses in rules can also be used in this way.

The use of correlations is in detecting unusual ones. If clause b almost always occurs with c, then something should be noted when they do not occur together. If variable v always occurs with w, then there may be a good reason for combining the variables. There should also be some justification for unusual correlations or distinctions.

Two rules that use the same variables are not necessarily redundant. As an artifact of the balance criteria described later, it will often be the case that there are sets of rules all using the same variables, giving the rulebase a natural clustering into groups.

Since each covariance matrix above is symmetric and positive semi-definite (as are the corresponding correlation matrices), one can consider computing eigenvectors to determine an "information space", as is done in associative information retrieval systems (see [Landauer,Mah]). The general idea begins with an arbitrary rectangular matrix B, indexed by $R \times C$ (these indexes are just rows and columns for this discussion; any of the incidence matrices or their transposes can be considered). First the association matrix A (indexed by $R \times R$) is computed as the transpose product $(B\ B^{tr})$, then the eigenvectors of the resulting matrix A are found. The eigenvector computation is not too hard, since A is symmetric and positive semi-definite.

This process of determining an abstract space in which to interpret some kind of measurement data is a special case of Multidimensional Scaling, and the eigenvector computations are the same mathematical procedures used in factor analysis and principal components analysis in statistics and pattern recognition (see [Gnanadesikan]).

It often turns out that the number of dimensions is too large to make eigenvector computation desirable. In those cases, the similarity measurements contained in the correlation matrix can be used in a cluster analysis. Clusters are cheap eigenvectors, and most simple clustering methods can give useful information (see [Sibson]). If the rows of B are considered as vectors in an information space, then the clusters of rows are sets of row items using related information.

Correlations can be used to check for some variable or expression dependencies, and particularly, almost dependencies (if a variable v almost always depends on a variable w, then something should be noted when it does not). If two expressions are highly correlated, then their values are almost related by a linear expression. The converse is also true, but correlations do not help directly with non-linear (i.e., almost all) relationships. However, if arbitrary functional transformations of the expressions can be made before the correlations are computed, then the correlations will help again. The problem becomes one of finding out whether or not there is a functional relationship, and finding its form (at least approximately) if there is one. This process is related to dimensionality reduction methods, such as nonlinear scaling or projection pursuit (see [Gnanadesikan], [Huber]), and is an important model construction method.

2.3 Criteria for Rulebase Correctness

This section describes some principles of rulebase correctness, and ways to test them for a particular rulebase. There is no description of how to determine whether or not to test the principles, since that decision is rulebase dependent. A principle of rulebase correctness is a condition on a set R of rules that is required for the rulebase to be reasonable in some incompletely defined sense. This notion is not the same as a principle of modeling a process or a system by rules (that step is hard). It is a notion of how rules fit together into a rulebase.

The five principles so far identified are:

- Consistency (no conflict),
- Completeness (no oversight),
- Irredundancy (no superfluity),
- Connectivity (no isolation), and
- Distribution (no unevenness).

These principles are implemented by many criteria for rulebase correctness. The criteria are separated into classes, according to the principles they implement. The criteria address logical consistency of the rules, completeness of specification of the rules, redundancy of the rules, connectivity of the rule and inference system, simplicity of the rules and the distinctions they cause, and the distribution of the rules and the values implied by them.

The first three principles, Consistency, Completeness, and Irredundancy, are not discussed in detail in this paper, since they are relatively easy to explain (see [Landauer89], [Landauer90] for the full discussion). The Connectivity and Distribution principles are discussed in detail in the next sections.

The Consistency principle leads to criteria that involve some kind of lack of conflict among rules. The idea is that the situations should be well defined, as should all the interesting variable values. The criteria will not be listed here, as they correspond to easy syntactic checks.

The Completeness principle leads to criteria that involve some kind of universal applicability of the rulebase. Defaults are usually used to guarantee certain kinds of completeness. All detectable places where defaults will be used should be signaled, since some of them may only indicate undesired incompleteness in a rulebase, instead of one expected to be fixed by the use of defaults. These criteria will also not be listed here.

The Irredundancy principle leads to criteria that insist that everything in the rulebase is there for some good reason. The variables make a difference, the rules make a difference, and there are no extraneous variables or rules.

2.3.1 Connectivity Criteria

These criteria collect rules together, involving either the entire dynamic process of inference, or the resulting graphs.

Criterion: recursion is dangerous

> The inference R graph should have no cycles.

Similarly for the inference C graph.

Dangling hypotheses and conclusions can be found very easily by looking for vertices in the clause graph (the inference C graph) that have no out-edges or no in-edges.

The rest of the criteria require the deduction graph to be nice in some sense. Disconnected components of the graph have no interaction, so they can be analyzed separately. There is some evidence to the effect that they should be described in different rulebases, instead of combining all the rules into one rulebase.

It is easy (though not necessarily fast) to check a finite directed graph for connectivity and for cycles.

The inference C graph has vertices for all clauses $c \in C$, and an edge from clause c to clause d whenever there is a rule r with $c \in hyp(r)$ and $d \in conc(r)$. A vertex with no out-edges is a clause c with no rule r having $c \in hyp(r)$ and $conc(r) \neq \emptyset$ (so c should involve only output variable values, or else it should not be in $conc(q)$ for any rule q, so that no inference chain can conclude that c holds). A vertex with no in-edges is a clause c with no rule r having $c \in conc(r)$ and $hyp(r) \neq \emptyset$ (so c should involve only input variable values, or else it should not be in $hyp(q)$ for any rule q, so that no inference chain can require that c holds).

The inference R graph has vertices for all rules $r \in R$, and an edge from rule q to rule r whenever there is a clause c which is in both $hyp(r)$ and $conc(q)$. A vertex with no out-edges is a rule r with no clause in $conc(r)$ and in $hyp(q)$

for any rule q (so any clause in *conc*(r) should only involve output variables). A vertex with no in-edges is a rule r with no clause in *hyp*(r) and in *conc*(q) for any rule q (so any clause in *hyp*(r) should only involve input variables).

2.3.2 Distribution and Simplicity Criteria

This section describes some of the simplicity and distribution criteria that can be used to signal possible problems with a rulebase. All of the criteria involve the way the rules divide up the set of situations. None of them is a mathematical correctness criterion; only a kind of "esthetic" criterion.

Criterion: simple distinctions

For every rule r,
the set of situations satisfying *hyp*(r) is simple.

Each rule r provides a distinction in the set S of situations between those situations s for which r acts and those for which it passes. When the boundary between those sets is too complicated, the expressions used in the hypothesis of r are awkward (and vice versa). It is sometimes necessary to use awkward phrases or distinctions in the rules, but some justification should be provided. Note that some awkwardness can be removed by using more than one rule in some cases.

Criterion: compact variable distribution

For every variable v,
the set of rules accessing v should be a small part of the entire rulebase.

This criterion affords a kind of modularity. The references to any one variable should be well-localized. A weaker form of the criterion would only require localization for the variables that occur in rule hypotheses. In any case, some variables (such as system health) must occur in many or all rules, but their wide distribution should be justified.

The other criteria describe various distributions as even. In this context, "even distribution" is less stringent than uniform distribution, and it really only means "not very non-uniform"; it represents a kind of balance condition. Cases of uneven distribution should be justified. It is clear that rulebases containing rare special cases will not satisfy these criteria. Part of the purpose of these criteria is to call such cases to the attention of the rulebase designer. The situations satisfying a given rule hypothesis should be evenly distributed in the variable domains. The rules accessing a given variable should be evenly distributed among its possible values.

Finally, The set of rules should be evenly distributed among variables. This criterion would prevent a larger number of rules from accessing (or just reading) one variable than for another. During rulebase development, some aspects of situations are not fully implemented in the rules, so some variables have very few references. This criterion signals those variables for further work (or justification).

The most blatantly non-uniform distributions are caused by unusual special cases. For example, if two variables always occur together except in one rule, or if two variable values are always correlated except in one rule, then the exception is an anomaly. In either case, some justification is required, either that there is a real difference for that one rule, or that there is a reason to have two variables where one might suffice.

The criterion examines the distribution of the rules over V. For a given variable v, the number of rules that access v is the column sum in column v of the counting incidence matrix RV. The row sum of row r of RV counts the number of variables mentioned in rule r. This count is related to the simplicity of *hyp*(r).

2.3.3 Distribution Checking

Distribution checking is not a well-established analysis technique. This section describes a test for each of the distribution and simplicity criteria defined earlier.

Using prospective situations, simple distinctions means only that *hyp*(r) is simple in form. Without using the entire rulebase to determine the set of situations, this is about the only thing one can check along these lines.

Compact variable distribution is easier to check. The column sums of the incidence matrix RV count how many rules contain the column variable v. Then v has a compact distribution if the sums are small. Uniform variable distribution also uses column sums of RV, checking that the numbers for a given variable v are all about the same (it should be noted that these criteria are more or less opposing, in that one wants all the values small and the other wants most of the values zero).

The uses of association and correlation matrices are even less well established. The basic idea comes down to one question (expressed here only for variables, but equally applicable to clauses or rules or other constructions):

> If v and w are highly correlated,
> then why are they different?

Detecting unusual conditions requires some computational indication of what the usual conditions are. For any given computational definition of the usual conditions, the cases not satisfying it can be determined (it is only after some empirical examination that the usual conditions can be computed, and deviations from that can be deemed unusual). For example, if variables v and w almost always determine expression e, then the usual case has the value of e for a particular situation dependent on the value of the pair (v, w) for that situation. Then there is some function of the pair (v, w) that should be nearly the same as the value of e, almost all the time. With the assumptions, it is now sufficient to distinguish large differences from small ones. Of course, there is still the problem of distinguishing small fluctuations (changes that do not indicate a new trend) in the usual values from the first signs of a real change in the usual values.

2.3.4 Other Criteria

This section contains some analyses that should lead to some other criteria, though more work is needed on each of them.

The various uses of correlation matrices to analyze rulebases are not so well established that they can be elevated to rulebase criteria. The simplest analysis considers only the pairs of items that have very high correlations (close to 1 or -1). Highly correlated variables, clauses, and rules might benefit from being rearranged to reflect the information structure better. For example, if two variables are highly correlated (over their sets of instances in clauses or rules), it might be better to express them both as deviations of some kind from a common variable. For another example, if two clauses have a correlation of -1, then they occur in large disjoint sets of rules (or they contain disjoint sets of variables, depending on which correlation matrix is used), so they are nearly mutual negations, and it might be better to replace one of them with the negation of the other. In this case (and, indeed, in all cases of high or unusual correlations), the correlation information is a derived feature of the rulebase, and may explain some facet of the system being modeled that was not previously seen as significant (or even noticed). It might therefore be better to leave the rulebase as it is until a sufficient explanation is found.

The association matrices to be considered are computed from the rule-clause incidence matrix RC, the clause-variable incidence matrix CV, and the rule-variable incidence matrix $RV = RC * CV$.

The main intent of these considerations is to find goodness criteria that can be evaluated using these association matrices. Until such time as they can be properly formulated, however, there are still some interesting questions. For example, what does it mean for all the eigenvalues to be the same size? What does it mean for one eigenvalue to be much larger than the rest? The hardest problem is not computational, but interpretational: to explain the dimensions in the information space (the principal components). There is still some controversy in whether or not there is any meaning in these inferred axes, even though (or perhaps because) the technique has been used in statistical analyses for many years.

3 MMU Analyses

Many analyses were performed that are implementations of the criteria discussed in the previous sections. There was no automatic version of any of the analyses, since the criteria are not yet implemented. The criteria were applied by hand, using editors, pattern searching programs, and other text manipulation programs available under UNIX. For this rulebase, some extra semantic information is available, such as the symmetry between sides a and b. This information was very useful in the analyses.

1	thruster
40	thrusters

Figure 1: Some Term Frequencies

1	no-xfeed-fuel-reading-test-side-a-grt
1	no-xfeed-fuel-reading-test-side-a-lss
2	no-xfeed-fuel-reading-test-side-b-grt

Figure 2: Some Rulename Frequencies

3.1 Preliminary Analyses: Uninterpreted CLIPS

The first analysis used a simple editor script, with (almost) no CLIPS knowledge beyond what can be found by looking at a CLIPS rulebase (which looks vaguely like LISP, with terms grouped together using parentheses). From the original rulebase file, all strings were mapped to "..." to avoid any reliance on word meanings. The names were selected from the text (here is where the CLIPS knowledge was used, in that the minus sign "-" can be part of a name instead of a delimiter). The left parenthesis "(" was also kept to separate function names from other names. Then the names were extracted, sorted, and counted to make a reference file.

This simple form of analysis found the first two errors. Among the name frequencies are the lines in Figure 1. The isolated instance is a mistake. In rule "xfeed-fuel-reading-test-general", there is a clause error. The delivered rulebase has

```
(checking thruster),
```

but it should have

```
(checking thrusters)
```

instead.

The second error is an incorrect rule name. The second instance of rule name "no-xfeed-fuel-reading-test-side-b-grt" is wrong. It should be "no-xfeed-fuel-reading-test-side-b-lss". This error was found as a side (a,b) asymmetry in the name frequencies shown in Figure 2.

Another anomaly found in the frequencies is not an error. The frequencies shown in Figure 3 might indicate an inconsistent use of the terms "cea-a-b" and "cea-coupled" that should be the same term. However, the two terms do mean something different in the rulebase, so the anomaly is not an error.

3.2 Detailed Analyses: Partially Interpreted CLIPS

A more systematic analysis based on the existing criteria was also conducted by hand. It differed from the preliminary analysis primarily in the degree of knowledge of CLIPS that was used in the editing process. Using this knowledge is equivalent to interpreting some of the symbols found in the rulebase, for example, in order to distinguish CLIPS commands from MMU terms.

4	cea-a
2	cea-a-b
4	cea-b
2	cea-coupled

Figure 3: More Term Frequencies

```
1          (vda a b1 off) (vda a b4 off) (vda a ?n&~ b1&~ b2&~ b3&~ b4 on)
1          (vda a b1 off) (vda a b4 off) (vda a ?n&~ b1&~ b4 on)
1          (vda a b1 off) (vda a f2 off) (vda a ?n&~ b1&~ f2 on)
4          (vda a b1 off) (vda a f3 off) (vda a ?n&~ b1&~ f3 on)
```

Figure 4: Some Lines from "vda" clause file

Some rulebase properties were found to be useful in this analysis that were not described in the previous section, and had not been considered as criteria for rulebase analyses (see [Landauer89]). The new criteria found during the analysis involve symmetry between side a and side b of the MMU, and, more generally, the symmetry among the replicated thrusters. The question to be asked in this case is, Do the multiple versions of a replicated object occur the same number of times in the rulebase, and if not, then why not? These criteria are associated with the distribution principle, and they simply say that any problem symmetries should be reflected in the rulebase, so that they appear in the distribution summaries.

Another kind of symmetry question, which not only concerns replicated objects, asks how to use geometric models in a "good" way. For the MMU, the thruster geometry is important in checking that the combinations of thrusters specified by the rules for correcting attitude and position errors correspond properly to the motions required to correct those errors. Because this geometric model was not provided with the rulebase, that analysis was not done.

3.2.1 Analysis Preparation

This analysis began with a revised rulebase, in which the two errors found earlier were corrected. They are clearly syntactic errors, and were fixed without further analysis. The new rulebase file was edited, using a knowledge of CLIPS syntax to identify terms and clauses, and to separate hypotheses of rules from conclusions. Many different syntactic items were separated: rule names, strings, functions, terms, and clauses were all placed into separate files of code numbers (to remove traces of semantic information derivable from the names). Editor scripts were made to translate items to code numbers, then several versions of the rulebase were made by partial translation.

The "separated rule file" was made by combining all hypotheses in each rule into one line, and all conclusions in each rule into one line. A file was made from the separated rule file to show clauses appearing in rule hypotheses, and a large number of different "(or " clauses was noted. files were made to show clauses appearing in rule conclusions, (except "(printout " clauses, which were omitted from the analysis, since the proper spelling and explanations for detected faults were not part of this analysis), and clauses appearing in each rule. These were used for the matrix and graph analyses.

3.2.2 Amplifiers

By far the biggest number of different clauses occurs for the clauses having function "(vda ", which concern the valve drive amplifiers (VDAs), each of which is used to control a thruster. Many of these clauses are collected in triples with "(or ". A file was made that contains the "(or " expressions with "(vda " clauses. Some lines of the file are shown in Figure 4 (the first column is a frequency count). The question marks in these clauses indicate variable names. For example, in the last line above, the third clause means that some side a thruster other than "b1" or "f3" is on.

The thruster names were selected as thruster names by context, manually. Every one occurs in a "(vda " clause, and it appears that every name that occurs as the third entry in a "(vda " clause is a thruster name. The spellings of the thruster names determine the original grouping, as shown in Figure 5.

The arrangement of thrusters and their relationship to roll, pitch, and yaw, and to rotation and translation will have to come from a geometric model of their locations and directions. Such a model is necessary for validation of the thruster commands.

Examining the "vda" clause file leads to the first thruster anomaly. Some clauses have all four "f" or "b" thruster names, and some do not. It turns out that the clauses with all four thruster names also have both sides on, and the clauses without all four have only one side on if the same thruster group is used (e.g., both "f" or both "b"), and both sides if different thruster groups are used. Since a model of the thrusters was not available, this anomaly cannot be resolved (an anomaly is not necessarily an error, remember, just something strange in the rulebase).

b1, b2, b3, b4
f1, f2, f3, f4
l1, l3
r2, r4
u3, u4
d1, d2

Figure 5: Thruster Name Grouping

b1, b4
b2, b3
f1, f4
f2, f3

Figure 6: Thruster Name Co-Occurrences

A file was made from the “vda” clause file to show which thrusters in the above groups are associated with each other in the same “(or ” combination of “(vda ” clauses (the same line in the “vda” clause file). The “f” and “b” groups subdivide, as shown in Figure 6. For example, “b1” and “b2” do not occur together in an assertion unless it asserts that some thruster different from both is on in the third disjunct of an “(or ” combination.

A file was made from the “vda” clause file to show which thrusters can be associated with which sides (the side is the second “(vda ” clause entry, and the thruster name is the third). This association leads to side assignments for the thruster subgroups above, as shown in Figure 7. The “l”, “r”, “u”, and “d” thrusters can appear with either side, but the “f” and “b” ones cannot (e.g., “b2” never appears with side a). Each of these files was also checked for side a, b symmetry, and no anomalies were found.

The MMU FDIR report says (only indirectly) that there are 24 thrusters, which was originally interpreted to mean that there are six places (the labels “b”, “f”, “l”, “r”, “u”, “d” are interpreted to mean “back”, “front’, “left”, “right”, “up”, “down”), with four thrusters in each place; however, not all names occur in the rulebase, so there is a possible symmetry error in allowing “l1” and “l3” for both side a and side b instead of just for one of them, with “l2” and “l4” for the other. Similarly, “r1”, “r3”, “u1”, “u2”, “d3”, “d4” do not appear, and yet are probably required to make 24 thruster names in all.

3.2.3 Hand Controllers and other Clause Notes

Another large group of clauses are the “(rhc ” and “(thc ” clauses, which deal with rotational and translational hand controllers. A file was made from the separated rule file to contain all those clauses. The complete file is shown in Figure 8 (the numbers on the left are frequencies). It turns out that every “(rhc ” clause is paired with a “(thc ” clause, and vice versa (this property was found by observation, but it could have been found by examining the correlations between occurrence patterns of these clauses). These counts also demonstrate the symmetry among roll, pitch, and yaw on the one hand, and x, y, and z on the other.

There are two styles of motion: rotation and translation. The rotations can be roll, pitch, or yaw, representing (it is assumed) the usual notions of vehicle attitude. The translations can be x, y, or z, representing (it is assumed) some

side a
b1, b4
f2, f3

side b
b2, b3
f1, f4

Figure 7: Thruster Name-Side Associations

4	(rhc roll neg pitch none yaw none) (thc x none y none z none)
4	(rhc roll none pitch neg yaw none) (thc x none y none z none)
4	(rhc roll none pitch none yaw neg) (thc x none y none z none)
4	(rhc roll none pitch none yaw none) (thc x neg y none z none)
4	(rhc roll none pitch none yaw none) (thc x none y neg z none)
4	(rhc roll none pitch none yaw none) (thc x none y none z neg)
25	(rhc roll none pitch none yaw none) (thc x none y none z none)
4	(rhc roll none pitch none yaw none) (thc x none y none z pos)
4	(rhc roll none pitch none yaw none) (thc x none y pos z none)
4	(rhc roll none pitch none yaw none) (thc x pos y none z none)
4	(rhc roll none pitch none yaw pos) (thc x none y none z none)
4	(rhc roll none pitch pos yaw none) (thc x none y none z none)
4	(rhc roll pos pitch none yaw none) (thc x none y none z none)

Figure 8: Hand Controller Clauses

24	(side a off) (side b on)
24	(side a on) (side b off)
43	(side a on) (side b on)

Figure 9: Side Clause Combinations

unspecified Cartesian coordinate system. The hypotheses of a single rule have changes in at most one component of at most one style of motion, and the changes occur symmetrically among the components. The relationship between the component being corrected and the combination of thrusters used to correct it cannot be checked, because no geometric model is available. An internally consistent relationship could be derived from the rules, but would not necessarily be correct.

Another large group of clauses is the "(side " clauses; a "side" clause file was made to contain them. They always occur in pairs, one for side a and one for side b, The pairs are shown in Figure 9. The few rules that do not have these clauses in their hypotheses are mostly in the rulebase to control other groups of rules, or to print out the problem statements (the rulebase has five predefined scenarios; special rules print the corresponding problems and solutions). There are no clause combinations for the case in which side a and side b are both off.

Other coverage notes were found by examining other kinds of symmetry. The "(aah " clauses, involving the Automatic Attitude Hold (AAH) process, and the "(gyro " clauses, involving the gyroscopes, form another potential source of error, since they split the attitude control information in two ways. The statistics of these clauses and their cooccurrences were computed, and are shown in Figure 10 and Figure 11. First, some simple anomalies are obvious at this point. There is no clause "(gyro off)" in any rule. There is no applicable rule if "(gyro off) and (aah on)".

Finally, another anomaly that is certainly an error was found by trying to infer from the above tables how these clauses combine in threes. It can be explained more easily, however, by noting that there is no combination of clauses "(aah on)" and "(gyro movement roll pos)". In fact, examining the four rules containing "(gyro movement roll pos)" shows

53	(aah off)
20	(aah on)
49	(gyro movement none none)
4	(gyro movement pitch neg)
4	(gyro movement pitch pos)
4	(gyro movement roll neg)
4	(gyro movement roll pos)
4	(gyro movement yaw neg)
4	(gyro movement yaw pos)
73	(gyro on)

Figure 10: Clause Counts for Attitude Clauses

49	(aah off) (gyro movement none none)
4	(aah off) (gyro movement roll pos)
53	(aah off) (gyro on)
4	(aah on) (gyro movement pitch neg)
4	(aah on) (gyro movement pitch pos)
4	(aah on) (gyro movement roll neg)
4	(aah on) (gyro movement yaw neg)
4	(aah on) (gyro movement yaw pos)
20	(aah on) (gyro on)
49	(gyro movement none none) (gyro on)
4	(gyro movement pitch neg) (gyro on)
4	(gyro movement pitch pos) (gyro on)
4	(gyro movement roll neg) (gyro on)
4	(gyro movement roll pos) (gyro on)
4	(gyro movement yaw neg) (gyro on)
4	(gyro movement yaw pos) (gyro on)

Figure 11: Pair Counts for Attitude Clauses

that the clause "(aah off)" is used instead. The same error also appears in the count for "(aah on) (gyro on)", which is 20 instead of 24, and in the count for "(aah off) (gyro on)", which is 53 instead of 49. The numbers 49 and 24 are much more consistent with the hand controller counts than 53 and 20 are.

3.2.4 Clause and Rule Associations

In order to compute associations, several files were made for incidence matrices and counts. The incidence matrix for clauses vs. rule hypotheses is sparse, with an entry for each rule that consists of a list of the clauses in the rule's hypothesis. The clause count vector has an entry for each clause that contains the number of occurrences of the clause in rule hypotheses. The co-occurrence matrix for clauses is also sparse, with an entry for each pair of clauses that occur together in a rule hypothesis. The entry is the number of rule hypotheses in which the two clauses occur together. The clause pair count vector has an entry for each clause that counts the number of clause pairs in which it occurs. These two count vectors are different, with the second one always larger. If a clause c occurs in exactly one rule hypothesis h, then the corresponding entry in the clause count vector will be one, and if that rule hypothesis h has four clauses, then the entry in the clause pair count vector for c will be three (one for each of the other clauses in the rule hypothesis).

Files were made for the clause count vector, the clause co-occurrence matrix, and the clause pair count vector.

The data files were converted, by editing them systematically, into two programs to compute correlations. For each clause, the first program ("frac.c") computes and prints the fraction of its co-occurrences with each other clause (when they do occur together). For each pair of co-occurring clauses, the second program ("corr.c") prints the correlations.

Suppose that each clause c has frequency $f(c,r) = 0$ or 1 in each rule r. Suppose also that there are nr rules and nc clauses. The rules are considered as samples, so each clause is considered as having some kind of clause distribution over the rules (not necessarily a random distribution), and various statistical measures can be computed. The clause count for clause c is $s(c) = \sum_r f(c,r)$, so its average frequency across the rules (the fraction of rules it is in) is $avg(c) = s(c)/nr$ and its variance is

$$var(c) = \frac{s(c)(nr - s(c))}{nr^2}$$

(since $f(r,c)^2 = f(r,c)$). The clause pair frequency for clauses c and d is

$$m(c,d) \quad = \quad \sum_r f(c,r)f(d,r)$$

and the clause pair count is

$$p(c) \quad = \quad \sum_{d \neq c} m(c,d)$$

$$
\begin{aligned}
&= \sum_{r} f(c,r) \sum_{d \neq c} f(d,r) \\
&= \sum_{r} f(c,r)(h(r)-1)
\end{aligned}
$$

for all clauses c, where $h(r)$ is the number of clauses in rule r. The correlation is computed in the usual way:

$$corr(c,d) = \frac{m(c,d)/nr - avg(c)\ avg(d)}{\sqrt{var(c)}\ \sqrt{var(d)}}$$

for all clauses c, d, which can be simplified to

$$
\begin{aligned}
sv(c) &= \sqrt{s(c)\ (nr - s(c))} \\
corr(c,d) &= \frac{m(c,d)\ nr - s(c)\ s(d)}{sv(c)\ sv(d)}
\end{aligned}
$$

for all clauses c, d.

The program "frac.c" is made from the clause co-occurrence matrix file and the clause pair count file to print co-occurrence fractions. Lines of the form

```
c000 9
```

in the clause pair count file become lines of the form

```
double c000 = 9;
```

in the program "frac.c". Lines of the form

```
1 c000 c022
```

in the clause co-occurrence file become lines of the form

```
printf(”c000,c022 = %.4f\n”,1/c000);
```

in the program "frac.c". The program then simply prints out the computed fractions.

The program "corr.c" is made from the clause co-occurrence matrix file and the clause count vector file to print correlations. Lines of the form

```
1 c000
```

in the clause count vector file become lines of the form

```
double c000 = 1.0, stdc000 = sqrt(1.0 * (nr - 1.0));
```

in the program "corr.c". Lines of the form

```
1 c000 c022
```

in the clause co-occurrence matrix file become lines of the form

```
printf(”c000,c022 = %.4f\n”,(1*nr-c000*c022)/(stdc000*stdc022));
```

in the program "corr.c". The program then simply prints out the computed correlations.

Then a file was made that contains all correlations above 0.7, sorted in decreasing order by correlation. Many clause pairs had a correlation of 1.0; almost all of the clauses in those pairs occurred exactly once in the rulebase, a few occurred twice, and one pair occurred six times each, all in the same rules. No anomalies were detected.

The largest correlation less than one is 0.9259, between the clauses "(aah off)" and "(gyro movement none none)", since only 4 of the 53 instances for the former clause do not occur with the latter clause. This is an anomaly, and in fact, it is the same error as the one described above. The next highest correlation is between a clause "(failure ?)", which only occurs in the combination "(not (failure ?))" (meaning that there is no asserted failure), and the two clauses "(xfeed-a closed)" and "(xfeed-b closed)" (separately). Only one rule contains the former clause without the latter clauses, which always occur together; these are the two mentioned above that occur in six rules. The extra rule containing the failure clause is a control rule that begins the tank and thruster test (most of the rules concern the electronics and not the propulsion system). This anomaly is not an error.

3.3 Discussion

The inference path analyses were not performed on this rulebase, due to their large computational requirements. It is expected that after the rulebase anomalies are corrected, an inference graph analysis will be performed.

It should be noted that many of the tests did not identify any anomalies. This situation is not a problem; because the theory is to apply as many tests as practical, there will often be tests that do not find errors. Moreover, in the rare cases of a correct rulebase, none of the tests will find any errors.

Special purpose tests, using special purpose criteria, will always be useful in analyzing a complex rulebase. The important point here is to make the special test usefully special, instead of making it the most general test possible. Some criteria can be made more widely applicable, and some will remain special purpose.

In the case of the MMU analysis, the symmetry criteria can be applied in general to systems with replicated components, but the choice of symmetries for the thrusters is specific to the MMU. The unmodeled geometric relationships among the thrusters was a missing aspect of the MMU definition that would have greatly assisted the analysis. A geometric model would allow validation of the rules that relate the VDA effects with the thrusters that cause them, and the rules that group thrusters together.

Perhaps the most interesting result of this analysis is that the tests that discovered errors did not do so automatically. In some cases, it was not at all obvious that the data represented errors. Some thought about the data and interpretation of results was required. It is not likely that a completely automatic system will find all errors in a rulebase (even aside from the undecideability barrier). A certain care will also be necessary.

However, these analyses and tools to implement them make the process of discovering some kinds of errors much easier, and should thereby make the design process much more effectively free of such errors.

4 References

[Bellman]

Kirstie L. Bellman, "The Modeling Issues Inherent in Testing and Evaluating Knowledge-based Systems", *Expert Systems and Applications*, Pergamon Press (to appear, 1990)

[Bellman,Walter]

Kirstie L. Bellman and Donald O. Walter, "Analyzing and Correcting Knowledge-Based Systems Requires Explicit Models", *Proc. AAAI 1988 Workshop on Verification and Validation of Knowledge-based Systems*, AAAI (1988)

[Culbert]

Chris Culbert, "CLIPS Reference Manual" (Version 4.2), NASA Johnson Space Center (April 1988)

[Gnanadesikan]

R. Gnanadesikan, Methods for Statistical Data Analysis of Multivariate Observations, Wiley (1977)

[Huber]

Peter J. Huber, "Projection Pursuit" (with discussion), *The Annals of Statistics*, Vol. 13 No. 2, pp. 435-529 (1985)

[Landauer89]

Christopher Landauer, "Principles of Rulebase Correctness", in Kirstie L. Bellman (ed.), *Proc. IJCAI 89 Workshop on Verification, Validation, and Testing of Knowledge-Based Systems*, Detroit, Michigan, 19 August 1989, AAAI (to appear, 1990)

[Landauer90]

Christopher Landauer, "Principles of Rulebase Correctness", *Expert Systems and Applications*, Pergamon Press (to appear, 1990)

[Landauer,Mah]

Christopher Landauer, Clinton Mah, "Message Extraction Through Estimation of Relevance", Chapter 8 in R. N. Oddy, S. E. Robertson, C. J. van Rijsbergen, P. Williams (eds.), Information Retrieval Research, "Proc. of the Joint ACM and BCS Symp. on Research and Development in Information Retrieval", Cambridge University, June, 1980, Butterworths, London (1981)

[Lawler,Williams]

Dennis G. Lawler, Linda J. F. Williams, "MMU FDIR Automation Task", Final Report, Contract NAS9-17650, Task Order EC87044, McDonnell-Douglas Astronautics Co. (3 February 1988)

[Sibson]

R. Sibson, "SLINK: An optimally efficient algorithm for the single link cluster method", *Computer Journal*, Vol. 16, pp. 30-34 (1973)

Satellite Simulations Utilizing CLIPS

by

Barbara Pauls
Mark Sherman

Rockwell International
Satellite & Space Electronics Division
P.O. Box 3644
Seal Beach, CA 90740-7644
MS: SJ62

Simulations provide necessary testbeds for system designs. Currently we are developing software whose main requirement is to produce CLIPS executable simulation code of a user prespecified system. This process minimizes the amount of engineering effort required to specify a system thereby reducing cost and providing the capability to quickly revise system definitions. Modeling satellite systems is the primary objective toward which testing has, and is, being conducted using satellite specifications. This paper describes the modeling software being developed, its formatted input and the CLIPS system simulation it produces.

Introduction

The main purpose behind our current satellite simulation efforts is to provide a testbed for autonomy research. The method currently being developed is to produce realistic and dynamic behavioral models reflecting current-state satellite systems. Future uses of the simulation method being developed may include the testing of more advanced and fault tolerant system designs.

The ability to easily add, delete, change and replace satellite subsystem definitions is required to support current research. Unfortunately, CLIPS, and expert system languages in general, are not common knowledge to most satellite engineers. To ensure efficiency, the approach used allows the specifications to be written in a 'higher-level' language. Such a modeling language has been defined and is referred to as Satellite Modeling Language (SML). The SML allows the user to specify the satellite system at any level desired. The satellite model can be defined at the system level, subsystem level or lower. Environmental affects on the satellite can also be defined using SML.

To convert SML code to CLIPS executable simulation code, a language translator was created. Consistent format of outputted code is automatically provided by the translator. The language converter can also implement necessary error checking. Currently the amount and type of error checking done by the SML translator is at a minimum. Future translator versions will include increased error checking capabilities of input modeling code. The language translator itself was written in CLIPS code. Being basically a sequential process difficulties arose forcing a language compiler to perform as an event driven process. However the experience of writing the translator in CLIPS provided understanding of CLIPS requirements needed to output simulation code.

By implementing the definition process in this manner, as shown in Figure 1, a basic structure evolved in each simulation model. This basic structure provides a certain degree of quality assurance, yet does not restrict the way in which a user defines a system. The specifications can be broken down into as many levels and/or modules as the engineer desires.

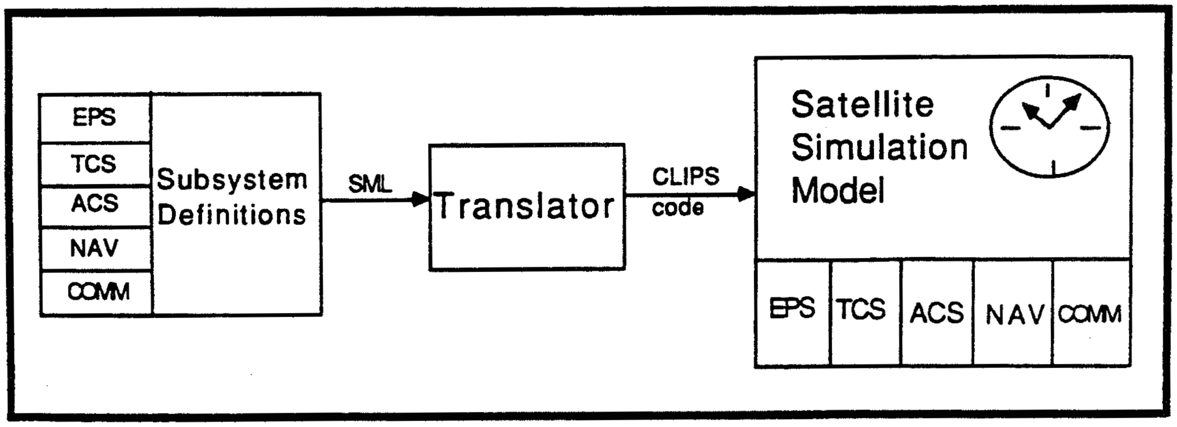


Figure 1. Simulation Definition Process

Satellite Modeling Language

SML consists of three main structures; templates, objects and rules. A template contains a generic set of attributes. The attributes are represented by simulation variables which describe the object. An object is created from a defined template and more than one object can be created from the same template. Objects can be specified at the time the template is defined or created separately. The SML rules define the simulation laws which all objects function under.

Object Definitions

Template specification contains a template name, optional object name(s), a list of attributes and their corresponding values. The SML 'define' command specifies a template and can create related objects. The syntax for the 'define' command is as follows, where optional fields are surrounded by square brackets []:

```
   define template [object1 object2 ... objectN]
        ( attribute1 = value1;
          attribute2 = value2;
                     :
          attributeN = valueN; )
```

Objects to be created with equivalent attributes are listed after the template name or can be specified by the SML 'create' command after the template has been defined. The syntax used to 'create' an object follows:

```
   create template object1 [object2 ... objectN]
        ( [attribute1 = value1;
           attribute2 = value2;
                   :
           attributeN = valueN;] )
```

Attribute values assigned when the template was defined may be changed for new objects. However, the 'create' command can not refer to any new attributes not defined in the corresponding template. If a template attribute is not listed in the 'create' command it retains the original value given in the template definition. The attribute value can be of any data type.

A one-dimensional array of attributes may also be specified. The array index is defined with square brackets and for every array element there must be a corresponding value, separated by commas. Examples of a template definition and object creations are given in Figure 2.

```
      define eps_template
         (nominal_power           = 0;
          batteries_enabled       = 0;
          batteries [1, 2, 3]     = off, off, off;
          main_bus_voltage        = 0;
          power_op_command        = off;
          enable_batteries_command = off;
          battery_on_command [1, 2, 3]  = off, off, off;
          battery_off_command [1, 2, 3] = off, off, off;)


      create eps_template EPS
         (nominal_power = 18;
          enable_batteries_command = on;)
```

Figure 2. SML Template and Object Examples

Rule Specifications

SML rules define and constrain simulation model behavior. Each rule is assigned a rule name in the 'behave' field and has a condition and an action section. The condition section of a rule is broken into five fields; 'priority', 'from', 'to', 'condition_start' and 'condition_end'. All five fields are optional. The action section of a rule must exist. Once the condition is met the action field is executed. The syntax for a rule is as follows:

```
 behave    rule-name
[priority  (priority-level)]
[from      (start-time)]
[to        (end-time)]
[condition_start  ( condition1
                    condition2
                        :
                    conditionN )]
[condition_end  ( condition1
                  condition2
                      :
                  conditionN )]
action     ( action1;
             action2;
                :
             actionN; )
```

The 'behave' field identifies the name of the rule and is required. The 'priority' assigns a priority value which is applied towards the order of rule execution and is restricted by CLIPS salience values to range between 0 and 10,000. Both the 'from' and 'to' fields are time oriented and have simulation default values which are currently provided by the interfacing process that uses the simulation as a testbed. Future versions may provide the capability to allow the user to specify default simulation times. When a time is specified in the 'from' field the condition is true if the current simulation time is greater than or equal to the specified start time. When a time is specified in the 'to' field the condition is true if the current simulation time is less than or equal to the specified end time.

When a 'condition_start' field exists and all conditions are met the rules action is fired. When a 'condition_end' field exists and all the corresponding conditions are true the rules action is not fired even if all time and start conditions are met.

The logical keywords 'and' and 'or' are used to connect rule conditions. The logical keyword 'not' is used to negate a condition. Legal SML comparison symbols are =, /=, <, <=, > and >=.

The 'action' field of an SML rule must exist and is executed when the corresponding conditions are met. Each action assigns or modifies values of object attributes. Currently SML input is constrained by the translators capabilities to use prefix notation in the action fields. The envisioned final translator version will allow infix notation in SML input. The rule examples given in Figure 3 depict future versions of SML input. Legal SML arithmetic operators are +, -, *, / and **. Currently only CLIPS functions are available in the SML input. User defined functions can be added to CLIPS and then used in SML input.

Comments may be inserted throughout SML code. Code between an exclamation character, !, and an end-of-line character is interpreted as user comments.

```
behave  EPS_NOMINAL_POWER
from  (10)
to   (950)
condition_start  (eps.power_op_command = on)
action  (eps.nominal_power = 18;
         eps.power_op_command = off;)


behave  RECORDER_1_COMMANDED_ON
to   (400)
condition_start  (comm.recorder_on_command.1 = on)
action  (comm.recorder_status.1 = on;)


behave DECREASE_AREA_A_TEMP   !environmental affect
condition_end  (not (acs.gyro_heater = on))
action  (tcs.area_a_temp = tcs.area_a_temp - .3;)
```

Figure 3. SML Rule Examples

Translator Description

The translator takes input files containing SML code and generates output files containing CLIPS code and an integrator symbol table. The translator requests names from the user for the input, output and integrator symbol files. Currently the translator converts three types of SML commands into CLIPS code; behave, create and define.

The input file can contain one or more SML commands. Any combination or order of SML commands is allowed. The output file has CLIPS code translated from an input file containing the SML commands. For each SML behave name there will be a CLIPS rule with the same name. An example of an SML behave command translated to a CLIPS rule is shown in Figure 4. The integrator symbol file contains a list of SML behave names, a list of variables that have been defined, and a list of variables not defined. The list of variables not defined may be defined in another input file that is yet to be translated. It is the responsibility of the simulation integrator program to report any undefined variables.

The translator was written in CLIPS to better understand the requirements of translation into CLIPS code. The translator is more of a sequential process than an event driven process. Many challenges were presented when a sequential process was coded in an event driven environment. Sequential coding was accomplished by using control flags. The translator was written to take advantage of event driven processes as much as possible.

The CLIPS translator code is stored in eight different files. The behave, create, and define files parse the SML commands and build the related CLIPS code. The read, and write files deal with input and output files. The index and field files parse a line from the SML file. The main file of the translator obtains user inputs, starts the translator and terminates the translator.

The translator relies on CLIPS being case sensitive. By converting the SML code into upper case and using lower case for the translator variables, duplicate fact names are reduced. The only exception to this rule is when a CLIPS function is used by an SML command thus requiring conversion to lower case.

SML

```
behave tcs_nominal_power_on
priority (  2)
from     (  0)
to       (250)
condition_start (tcs.power_op_command = on)
condition_end   (tcs.power = off)
action ((tcs.nominal_power = 5);
        (tcs.power_op_command = off);)
```

CLIPS

```
(deffacts TCS_NOMINAL_POWER_ON-time
   (TCS_NOMINAL_POWER_ON-from-time 0)
   (TCS_NOMINAL_POWER_ON-to-time 250))

(defrule TCS_NOMINAL_POWER_ON
   (declare (salience 2))
   ?a_toc <- toc TCS_NOMINAL_POWER_ON)
   (time ?time)
   (TCS_NOMINAL_POWER_ON-from-time ?from-time)
   (TCS_NOMINAL_POWER_ON-to-time   ?to-time)
   (TCS.POWER ?TCS.POWER)
   ?a_TCS.POWER_OP_COMMAND <- (variable-data
      TCS.POWER_OP_COMMAND ?TCS.POWER_OP_COMMAND)
   ?a_TCS.NOMINAL_POWER <- (variable-data
      TCS.NOMINAL_POWER ?TCS.NOMINAL_POWER)
=>
   (retract ?a_toc)
   (if (and
       (>= ?from-time ?time)
       (<= ?to-time   ?time)
       ( eq ?TCS.POWER_OP_COMMAND ON )
       (not
       ( eq ?TCS.POWER OFF )
       )) then
      (retract ?a_TCS.NOMINAL_POWER)
      (retract ?a_TCS.POWER_OP_COMMAND)
      (assert (variable-data TCS.NOMINAL_POWER 5))
      (assert (variable-data TCS.POWER_OP_COMMAND OFF)
   )
)
```

Figure 4. Sample SML Behave Translation

Translation of SML Define and Create Commands

Figure 5 shows the translation of the SML define and create commands into CLIPS code. Each part of the define and create command is broken up into individual pieces (i.e. template, object, attributes) during the reading of the command. Each piece is tagged with the template name for latter use in generating CLIPS code. The generation of CLIPS code from the define command is delayed until after all the create command CLIPS code has been generated. This is because the create and define command can come in any order and the create translation needs the pieces of the define command. After all the create commands have generated their CLIPS code, the define command can then generate CLIPS code. Once the define command has generated the CLIPS code all the pieces related to the define command can be deleted.

```
                                                              SML
define tcs_template
   (power_op_command    = on;
    power               = off;
    nominal_power       = 5;)

create tcs_template tcs
   (power_op_command = on;)
```

```
                                                            CLIPS
(deffacts TCS_TEMPLATE
   (variable-data TCS.POWER_OP_COMMAND OFF)
   (variable-data TCS.POWER OFF)
   (variable-data TCS.NOMINAL_POWER 5))

(deffacts TCS
   (variable-data TCS.POWER_OP_COMMAND ON)
   (variable-data TCS.POWER OFF)
   (variable-data TCS.NOMINAL_POWER 5))
```

Figure 5. Sample SML Define and Create Translation

Translation of SML Behave Command

Figure 4 shows the translation of the SML behave command. For every SML behave command the translator produces a maximum of one CLIPS deffacts statement and one CLIPS simulation rule. If any time conditions are specified in the SML rule 'from' and 'to' fields, a deffacts statement is created which asserts minimum and/or maximum time values specifically corresponding to the simulation rule. These are then tested in the CLIPS rule against the simulation time.

In order to assure that all CLIPS rules are executed once per simulation second, the left hand side (LHS) conditions of the CLIPS rule must always be true. Therefore only necessary facts are referenced on the LHS using binding variables whenever possible. The SML specified conditions are then tested on the right hand side (RHS) of the CLIPS rule using an 'if...then' structure.

Each SML behave command consists of six specific parts. The 'priority' part translates to a declaration of rule salience. The 'from' and 'to' parts define a check on the simulation time facts done on the RHS of the CLIPS rule. The 'condition_start' and 'condition_end' part also define the 'if...then' check done on the RHS of the rule. The SML 'action' part translates to retract and assert statements in CLIPS code.

Simulation Integration

The integrator program accepts input from the integrator symbol table. The integrator symbol table is created by the translator program. The integrator symbol table, see Figure 6, contains a list of all SML rule names, a list of SML variable names, and a list of undefined SML variable names. The list of SML defined and undefined variable names have been provided for future enhancements. The output of the integrator program is the dynamic CLIPS code, see Figure 7. The dynamic CLIPS code file contains any simulation control code needed to run the simulation model.

```
                                                   INPUT
*** NEW ***
TCS_NOMINAL_POWER_ON
    :
*** NEW ***
    :
NAV_PAYLOAD_ELECTRONICS_SDTBY

                                                   OUTPUT
TCS_NOMINAL_POWER_ON
    :
NAV_PAYLOAD_ELECTRONICS_SDTBY
```

Figure 6. Sample Integration Symbol Table

```
(defrule tic
   (not (tic-done))
   ?a_tic <- (tic)
   ?a_time <- (time ?time)
   (time-max ?time-max)
=>
   (retract ?a_tic)
   (bind ?num (+ ?time 1))
   (if (<= ?num ?time-max) then
      (retract ?a_time)
      (assert (time ?num))
      (assert (toc TCS_NOMINAL_POWER_ON))
            :
      (assert (toc NAV_PAYLOAD_ELECTRONICS_SDTBY))
   else
      (assert (tic-done))
      (assert (get-tic))
   )
)
```

Figure 7. Dynamic CLIPS code

Simulation Model

All simulation code output from the SML translator is CLIPS executable. For every SML file input to the translator one corresponding CLIPS file is output. To execute the simulation all the translator outputted files and two other input files, one static and one dynamic file, are loaded into the CLIPS environment. The simulation static file contains the simulation time control rules and any other CLIPS rules needed that are not subsystem dependent. This additional code provided in the static file is user specified simulation requirements not supplied by the SML input. The dynamic file contains time rules which control simulation rule execution. This file is generated by the integrator program previously described. The remaining files contain SML translated commands. In our simulation model each subsystem was described in one SML input file and after translation each subsystems simulation code was contained in a unique output file.

As previous examples have shown, satellite simulations have been defined on the subsystem level using command and measurement attributes to describe each subsystem. Once these object attributes have been defined and created a time clock is introduced by the translator produced static file to control the simulation processing. The implementation of time restricts rule execution by allowing each CLIPS rule to fire only once per simulation second. Start and end times of the simulation clock are currently defined by a higher level process interfacing with the satellite model. The simulation can be defined as a stand-alone process if the start and end times are hard coded in the static file.

In order to simulate time the translator produces a predefined set of time rules which are based on a 'tic toc' process. A 'tic' fact serves as a timer interrupt and in our current simulations is produced by the higher level process interfacing with the satellite model. This interrupt could be produced by the CLIPS simulation model itself if it were to execute stand-alone. The CLIPS simulation always processes the timer interrupt using one rule. This rule retracts the 'tic' fact when it exists, validates that the current time is less than the maximum simulation time, increments time and asserts a 'toc' fact for every translated SML rule. Each 'toc' fact is retracted when its corresponding simulation rule is executed. When all 'toc' facts have been retracted the simulation model is hung until another timer interrupt, a 'tic' fact, is asserted.

Currently no user interface exists to run a stand-alone CLIPS simulation model. Any information to be displayed during runtime must be added to the CLIPS simulation code and no operator interrupt capability has been provided. However our current uses do not require a stand-alone interface.

Summary

The method which evolved from the basic satellite simulation approach provides the tools needed to minimize development effort and allow the subsystem engineers to quickly revise system definitions. The input and output requirements for any simulation are independent and in our approach we left such requirements to be implemented by the simulation coordinator. The simulation interface can be coded in CLIPS and put into the static file so as not to complicate subsystem engineer development. When a function is needed which is not provided by either SML or CLIPS it can be easily defined in CLIPS and then referenced in the SML descriptions.

Utilizing the CLIPS expert system language as the simulation code was quite advantageous. Coding the SML translator in CLIPS was a challenge, however, this approach did provide insight to CLIPS capabilities and functionality. For the satellite modeling effort CLIPS provided a more than suitable event driven simulation environment. Other advantages to utilizing CLIPS included low cost, high portability and easy integration with external systems. We believe the approach described allows the definition of a wide range of satellite architectures, satellite behaviors and environmental influences with minimal effort.

omit

B9 Session:
User Interface II

S14-61
ABS. ONLY

358572

IMPROVING THE HUMAN FACTORS OF SOFTWARE WITH CLIPS

P-1

Thomas J. Nagy
Management Science Department
George Washington University
Washington, D.C. 20052

ABSTRACT

The use of CLIPS has transformed a conventional graduate course on the human factors of software. Previously, the class centered on lectures and discussions of a mix of ideas for improving the user-friendliness of software. By using CLIPS, the course can focus instead on teaching students to build three rule-based projects in CLIPS for improving the human factors of software.

For the first project, students construct a friendly CLIPS front-end to existing software. For the second project, students build a CLIPS expert system to help comply with user-interface guidelines. Alternatively, students may build an expert system to assist in detecting discrepancies between user-interfaces and guidelines. For the third project, students use CLIPS to implement a GOMS Model Methodology to assess the human performance impacts of given user-interfaces.

Feedback on the projects from the students' colleagues and superiors in the workplace confirm the effectiveness of this CLIPS project-oriented approach to teaching the human factors of user-computer systems. Future refinements are described. Suggestions for those wishing to try this approach are outlined.

A MEMORY EFFICIENT USER INTERFACE FOR CLIPS MICRO-COMPUTER APPLICATIONS

aurthors

Mark E. Sterle
Richard J. Mayer
Janice A. Jordan
Howard N. Brodale
Min-Jin Lin

Knowledge Based Systems Laboratory
Department of Industrial Engineering
Texas A&M University
College Station, TX 77843
(4009) 845-8500

April 24, 1990

A MEMORY EFFICIENT USER INTERFACE FOR CLIPS MICRO-COMPUTER APPLICATIONS

ABSTRACT

The goal of the Integrated Southern Pine Beetle Expert System (ISPBEX) is to provide expert level knowledge concerning treatment advice that is convenient and easy to use for Forest Service personnel. ISPBEX was developed in CLIPS and delivered on an IBM PC AT class micro-computer, operating with an MS/DOS operating system. This restricted the size of the run time system to 640K. In order to provide a robust expert system, with on-line explanation, help, and alternative actions menus, as well as features that allow the user to back up or execute "what if" scenarios, a memory efficient menuing system was developed to interface with the CLIPS programs. By robust, we mean an expert system that (1) is user friendly, (2) provides reasonable solutions for a wide variety of domain specific problems, (3) explains why some solutions were suggested but others were not, and (4) provides technical information relating to the problem solution. Several advantages were gained by using this type of user interface (UI). First, by storing the menus on the hard disk (instead of main memory) during program execution, a more robust system could be implemented. Second, since the menus were built rapidly, development time was reduced. Third, the user may try a new scenario by backing up to any of the input screens and revising segments of the original input without having to retype all the information. And fourth, asserting facts from the menus provided for a dynamic and flexible factbase. This UI technology has been applied successfully in expert systems applications in forest management, agriculture, and manufacturing. This paper discusses the architecture of the UI system, human factors considerations, and the menu syntax design.

USER INTERFACE ARCHITECTURE

The UI architecture was developed as a result of the requirements of the ISPBEX system for memory efficiency and fast execution speeds. By designing menus that could be stored on the hard disk during program execution, main memory could be reserved for execution of the CLIPS program. Thus, more rules could be incorporated into the system, more facts could be maintained, and the program could run faster. This architecture, illustrated in figure 1, consists of two components, the FIFTH programming environment and the menu interpreter.

FIFTH Programming Environment

The FIFTH programming environment, was developed by Cliff Click and Paul Snow and is maintained by the Software Construction Company. FIFTH facilitates the compilation and debugging of text description menus by providing high level utilities to access the low level commands of Forth. The programmer uses a simple text editor to create the text descriptions which specify the size, content, and actions of each menu. This eliminates the need for complicated key sequences and instructions and thus allows rapid menu creation. The menus are compiled into microprocessor-like binary format instructions (e.g. push, pop) and stored on the hard disk in a file.

Menu Interpreter

The second component of the system, the menu interpreter, is written in C. The menu interpreter loads and interprets the binary instructions for a menu only when a call is received from the expert system. The menu interpreter is extendible because new commands can be easily defined and compiled into the C code providing a system that can meet the needs of the particular application. A menu selection history utility was also developed so the user could review or modify data that was previously entered into the system for a problem scenario.

Advantages of the UI Architecture

The design of the UI architecture allows the programmer to design a robust and user friendly system by providing a memory management method for developing the menus. Since a text editor can be used to create the menus, quick prototyping is simplified. The extendible menu interpreter allows the programmer to implement a system that can be tailored to the user's needs. All of this went into the development of a UI to meet the requirements for memory efficiency and fast execution speeds.

HUMAN FACTORS CONSIDERATIONS

Human factors design considerations were included in the development of the UI. Since this system would be utilized by Forest Service personnel with various levels of computer expertise (managers as well as technicians), it had to be easy to learn and convenient to use. One design consideration which simplified use of the system involved requiring the user to remember as few keys as possible. Another consideration included the use of help menus and explanation files. Also, backing up to allow the user to execute similar scenarios made using the system easier. Ten keys were selected for this implementation and were mapped to the functions described in table 1, titled Mapping of Keyboard Keys to Functions.

Cursor Movement, Option Selection, and Data Entry

The first set of keys shown in the table allow the user to move the cursor or leave a menu. The arrow keys are used to move the cursor to the appropriate selection so the user may select an item by pressing the enter key. Or the user may type the first letter of the selection to move the cursor to the item. If a selection is made by mistake, the user can deselect the item by moving the cursor to the selection and pressing the enter key. If the selection requires information to be typed in the space provided, the data can be entered after moving the cursor to the selection without hitting the enter key first. The data is checked for validity, type, and length which is defined in the menu description and if a mistake is made (syntax or out of bounds) an error message is displayed and the user is allowed to re-enter the information. When the user presses enter again, the cursor moves automatically to the next selection. The del key will cause the cursor to backspace and delete one character at a time. The menu can be designed so that the user goes directly to the next menu after making a selection (that is, after pressing the enter key) or the user can be permitted to review selections and leave the menu by pressing the end key.

Function Keys

The second set of keys are the function keys. Help and explanation menus can be accessed where provided by moving the cursor to an item and pressing the F1 key. The user may then press any key to return to the previous menu where the cursor will be on the same item. General information on use and movement of the cursor and function keys is provided from any position on any menu by pressing the F2 key. Pressing the F10 key will cause a question to appear asking if the user wants to end the ISPBEX session and exit to MS/DOS. Typing an *n* will simply return the user to the previous menu.

Viewing Results

The third set of keys shown in the table allow the user to view and leave a result file. To view a result file in which information determined by the application program during a session has been stored, the user can use the page up and page down keys to move through the text. The up and down arrow keys can be used to move up or down one line at a time through the file and the right and left arrow keys can be used to view files that are wider than 80 columns. The esc key will return the user to the previous menu from which the result file was accessed. No editing can be performed on these files as the information they contain is determined by the program.

Human Factors Design Benefits

Several factors were deemed necessary in order for the WCA to be successfully implemented. First, by keeping the number of keys required to a minimum and providing the user with individual selection help utilities, a system can be designed that is easy to learn as well as convenient to use. With this system, the user does not have to remember complicated key sequences or details about system implementation, and thus, first time or infrequent use becomes less trying. Also, the user can learn from the expert system because files are created by the system which give explanations and details for why certain results were suggested and the logic that went into making the decisions. Second, by backing up to previous menus, the user can execute similar scenarios and soon begins to understand the subtleties involved in the complicated reasoning processes that the experts used to make those decisions. Additionally, the user can save time with the history facility when executing problem scenarios that have similar data input to ones already evaluated because no time is lost due to re-entering all the data. These factors were considered necessary for the successful implementation of this system.

TEXT MENU SYNTAX

The text menu syntax was designed to help the programmer develop menus rapidly and easily. Figure 2, Text Menu Syntax, is an example of a text description for a menu showing the identification and location of an infestation of southern pine beetle, *Dentroctonus frontalis* Zimn. (Coleoptera: Scolytidae). It also includes the help menus associated with the input data. First, the main menu name is declared, followed by the help menu declarations. Next, the help menus are defined. Following the help menu definitions are the commands that are executed from the main menu upon making a selection. And finally, the main menu is defined. The following paragraphs will explain some of the syntax shown in figure 2 in more detail, how the menu is called from a CLIPS program, and the additional commands that are available.

UI Function Call

The main menu is identified by the name SPBDATA and is called by a CLIPS rule during program execution. The call looks like this:

(ui "spbis.mnu" spbdata history ?id-code)

The user defined function call to the menu interpreter is *ui*. The name of the file containing the compiled menu definitions is *spbis.mnu* in this example. The main menu name, *spbdata*, is next. Any number of menus (usually related functionally) can be stored in one file. The items entered by the user in the menu are stored in another file called *history* and will appear the next time this menu is accessed. A unique history file name must be used for each menu. A *NULL* can replace a file name if saving the information is not desired. The parameter, *?id-code*, is passed to this menu for display as the infestation number at the equals sign on the main menu shown in figure 2 . The number *0*, shown in the figure, represents the first parameter passed to the menu. Additional parameters would be numbered in increasing order (*1*, *2*, etc.).

Help Menu Definition

The help menus are defined within a set of brackets with the menu name following the closing bracket. The *0 0*, located after the opening bracket, refers to the minimum and maximum number of selections, respectively, that must be made before leaving this menu. This indicates no selections are to be made and the user can leave the menu and return to the main menu by pressing any key. *Menu-begin* and *menu-end* indicate the start and finish of the menu's display area. The ^ symbol specifies the border limits. Finally, *3 8*, followed by the command *display*, indicates the position on the CRT screen, row and column, to display the help menu.

Menu Selection Commands

Each item that can be selected from the main menu has three sets of brackets associated with it. The brackets contain menu commands, summarized in table 2, that can be executed from the menu. The following is a brief description of each of these commands.

The first set of brackets contain commands which are executed when the cursor is moved to that item and the enter key is pressed. The *prtmsg* command causes the message in the quotes to be displayed at the bottom of the main menu as shown in figure 3. The *1 20 readi* command specifies that an integer between 1 and 20 is to be entered for this item. Similarly, the *"0123456789WLD" reads* command restricts the user to entering an integer or the special characters WLD. Other possible commands available for defining input are: *read*, which reads any printable keyboard input; *reada*, which reads an alphabetic character; *readr*, which reads a real number; *readan*, which reads alpha-numeric input; *readdate*, which reads a date. All the numeric read commands have range checking. If an out of bounds number is entered an error message will tell the user to enter a value between the specified bounds. These commands help the user by checking the input to avoid errors in data entry and thus, data integrity can be maintained.

The second set of brackets contain commands that are executed when the the F1 key is pressed. The *exec* command causes a help menu containing detailed information about the item pointed to by the cursor to be displayed. The user returns to the main menu after leaving the help menu. A sub-menu can also be called with the *exec* command that allows information related to the items in the main menu to be entered and then returns the user to the menu it was called from.

The third set of brackets contain commands that will be executed upon leaving the main menu. Two commands, *readwrd* and *readstr*, will read and store the word or string entered by the user for a selected item. The *assert* command causes the string within the square brackets to be asserted to the fact base of the CLIPS program.

Other Menu Modifiers

Placing the *exit* command in the first set of brackets will cause the user to leave the menu if the cursor is next to that item when the enter key is pressed. Otherwise, the user is required to enter the minimum to maximum number of items specified by the numbers that precede the *menu-begin*. If these numbers are equal, but not zero, the user will leave the menu as soon as the minimum/maximum number of entries have been performed. If they are different, the user must press the end key to exit the menu.

The asterisk is used to specify cursor placement on the screen for an item. When enter is pressed on the selected item the line following the asterisk will be highlighted. The programmer can also use the ampersand/tilde combination to control cursor direction. This is especially useful when the user must fill in several items because it will allow the use of arrow keys to wrap around the menu.

Advantages of the UI Menu Syntax

Changes to menus during the prototyping phase of knowledge acquisition can be made quickly by the programmer or an expert who has minimal knowledge of programming and computers by using a simple text editor. Development time for the system is thus reduced. Utilities for error checking of user data entry are extendible and can be specifically tailored for the application and user's needs. This saves the user time because errors are caught immediately and there is no reason to rerun the entire program. Methods for cursor movement, data entry, and leaving a menu can be specified which make the system less cumbersome to use.

FUTURE ENHANCEMENTS

Three enhancements to the UI would improve the performance, maintainability, and versatility of the system. First, the FIFTH programming environment, currently written in Forth, and menu interpreter, which is written in C, should be rewritten in one language. This would provide a single environment for creating the menus and allow easier modification and enhancement of the UI system. Second, the current UI operates only on the IBM PC AT class machines running with MS/DOS and should be rewritten to port to other operating systems, such as UNIX. And third, the UI was designed specifically to run with CLIPS and should be rewritten as a stand alone package that can be used with other software systems.

SUMMARY

From a programmer's view point, there are four advantages gained by using this UI. First, because the menus are stored on the hard disk (instead of in main memory) during program execution, a more robust system can be implemented. Second, the compact size of the binary files leads to efficient memory usage. Third, since the menus can be built rapidly using a text editor, fast prototyping speeds up the knowledge acquisition phase and development time is reduced. And fourth, because new commands can be added to the text menu syntax, the system is extendible and can be tailored to the user's specific needs and requirements.

A user benefits from the use of this system in four ways. First, detailed help menus can easily be associated with any item and displayed using a common function key. Second, backing up and saving menu choices allows the user to repeat similar scenarios without having to re-enter all the information. Third, asserting facts from the menus provides for a dynamic and flexible factbase. And fourth, requiring the user to remember as few keys as possible makes learning and remembering how to use the system easier. This UI technology has been applied successfully in expert systems applications in forest management, agriculture, and manufacturing.

Mapping of Keyboard Keys to Functions

• *Arrow keys*	Moves the cursor up, down, right, and left.
• *Enter*	Causes the item next to the blinking cursor to be selected or deselected from a menu.
• *Del*	Allows you to backspace and deletes values already typed in.
• *End*	Allows you to leave a menu after the appropriate information is entered.
• *F1*	Causes help information to be displayed if available for the item that the cursor is on.
• *F2*	Causes explanations for the user keys to appear.
• *F10*	Allows you to end the expert system session and return to DOS.
• *Page up*	Causes the previous page of a result file to be displayed.
• *Page down*	Causes the next page of a result file to be displayed.
• *Esc*	Allows you to leave a result file.

table 1

Menu Commands

• *prtmsg*	Causes message to be displayed at bottom of main menu.
• *readi*	Specifies that an integer is to be entered for this item.
• *reads*	Restricts user to entering the special characters designated.
• *read*	Reads any printable keyboard input.
• *reada*	Reads an alphabetic character.
• *readr*	Reads a real number.
• *readan*	Reads alpha-numeric input.
• *readdate*	Reads a date.
• *exec*	Causes a help menu containing detailed information about the item pointed to by the cursor to be displayed.
• *readwrd*	Read and store word entered by the user for a selected item.
• *readstr*	Read and store string entered by the user for a selected item.
• *assert*	Causes the string within the square brackets to be asserted to the fact base of the CLIPS program.
• *exit*	In the first set of brackets causes the user to leave the menu when the enter key is pressed.

table 2

User Interface Architecture

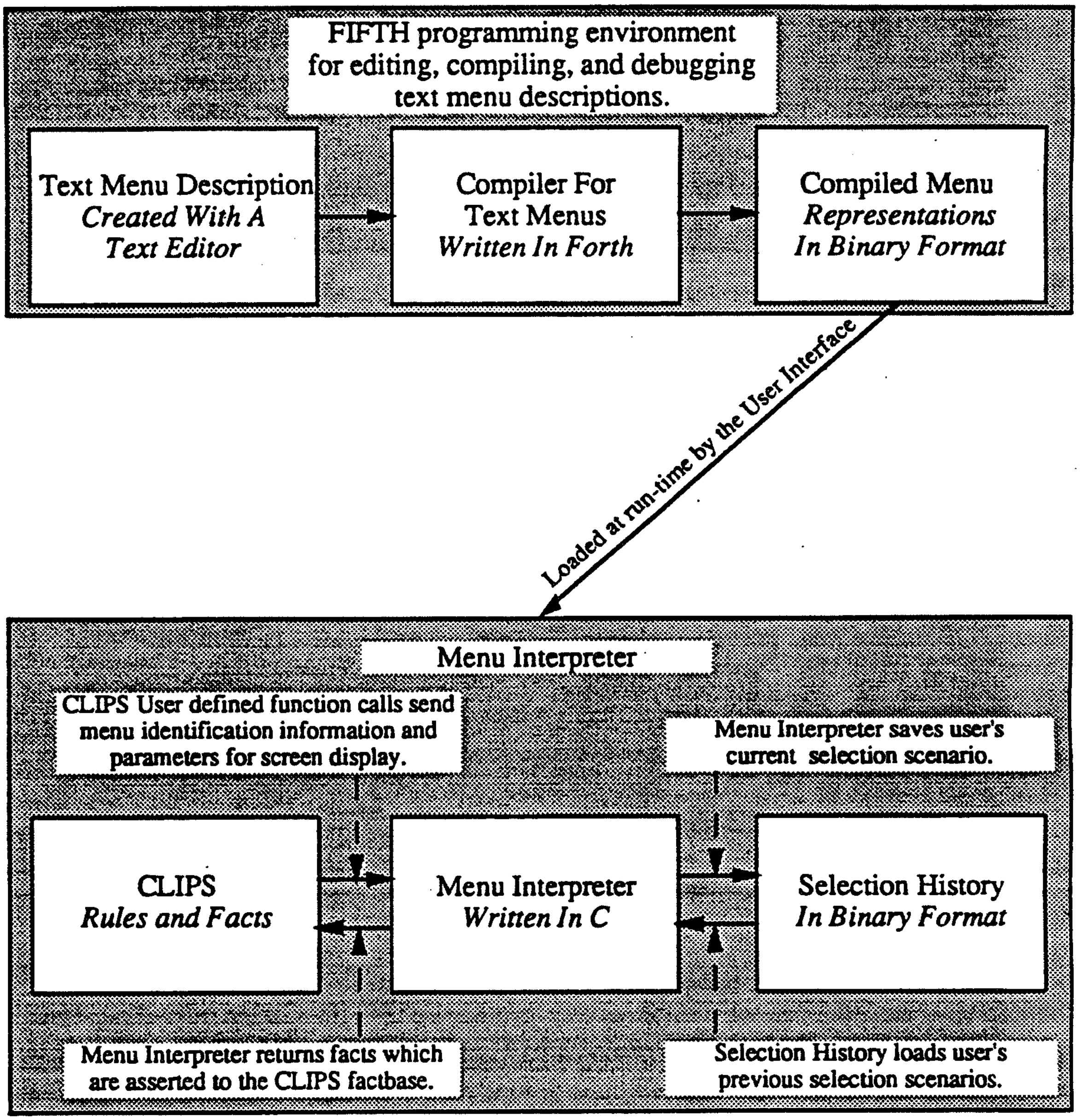

FIFTH programming environment for editing, compiling, and debugging text menu descriptions.
Text Menu Description *Created With A Text Editor*
Compiler For Text Menus *Written In Forth*
Compiled Menu *Representations In Binary Format*
Loaded at run-time by the User Interface
Menu Interpreter
CLIPS User defined function calls send menu identification information and parameters for screen display.
Menu Interpreter saves user's current selection scenario.
CLIPS *Rules and Facts*
Menu Interpreter *Written In C*
Selection History *In Binary Format*
Menu Interpreter returns facts which are asserted to the CLIPS factbase.
Selection History loads user's previous selection scenarios.

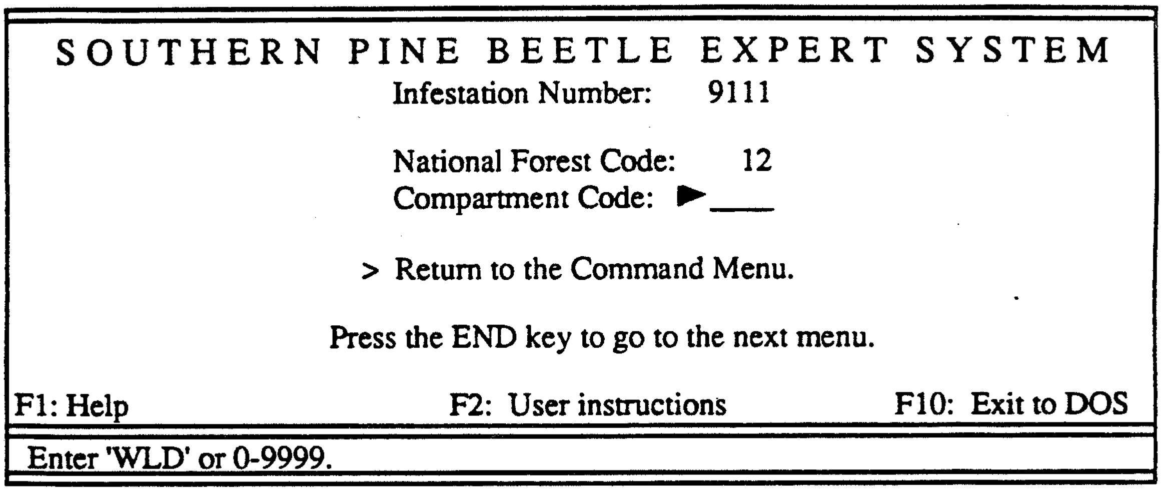
figure 1

Text Menu Syntax

```
SPBDATA
!00000898
define spbdata
var help1
var help2

{ 0 0
menu-begin
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          Enter number from 1 to 20 for the National Forest Code.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
menu-end
3 8 display
} define help1
{ 0 0
menu-begin
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Enter number from 0 to 9999 for general forest or the letters WLD for wilderness.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
menu-end
3 8 display
} define help2

{ " Enter 1-20 for National Forest. " prtmsg 1 20 readi }{ help1 exec }{ [national-forest readwrd ] assert }
{ " Enter 'WLD' or 0-9999. " prtmsg "0123456789WLD" reads }{ help2 exec }{ [comp readwrd ] assert }
{ [ backup command ] assert exit } { } { } 3 5
menu-begin
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   S O U T H E R N   P I N E   B E E T L E   E X P E R T   S Y S T E M
                              Infestation Number:     = 0

                              National Forest Code:    &__~
                              Compartment Code:    &____~

                          *> Return to the Command Menu.

                    Press the END key to go to the next menu.

F1: Help                         F2:  User instructions                 F10:  Exit to DOS
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
menu-end
1 1 display
end
```

figure 2

Example Menu Output



```
SOUTHERN PINE BEETLE EXPERT SYSTEM
              Infestation Number:    9111

              National Forest Code:     12
              Compartment Code: ►____

            > Return to the Command Menu.

          Press the END key to go to the next menu.

F1: Help              F2: User instructions          F10: Exit to DOS
Enter 'WLD' or 0-9999.
```

figure 3

5/6-61

N96- 12929

Prototyping User Displays Using CLIPS

p. 6

358576

Charles P. Kosta
Ross Miller

Center for Productivity Enhancement
University of Lowell
Lowell, MA

Dr. Patrick Krolak
Matt Vesty

Transportation Systems Center
Cambridge, MA

Abstract

CLIPS is being used as an integral module of a Rapid Prototyping System. The Prototyping System consists of a display manager for object browsing, a graph program for displaying line and bar charts, and a communications server for routing messages between modules. A CLIPS simulation of physical model provides dynamic control of the user's display. Currently, a project is well underway to prototype the Advanced Automation System (ASS) for the Federal aviation administration.

A prototype, as defined by *The American Heritage Dictionary*, is an original type, form, or instance that serves as a model on which later stages are based or judged.

LEVELS OF FUNCTIONALITY

The prototyping of user interfaces has evolved into four distinguishable levels. The first level is the "straw man" stage, when a basic screen design is developed that approximates how the interface should look. The purpose of this phase is to work out aesthetics issues only; it does not give any indication of the usability of the display. Using C or another script -like language, the second level prototypes static responses using limited scenarios. At this phase the objects can react to user input, but the responses do not deviate from an internal script. The third level incorporates a dynamic response from the system. During this phase the dynamic system attempts to mimic the real system as closely as possible in such areas as responding to user events and simulating (or generating) user scenarios. While using this level prototyping users should not be able to tell that they are using a prototype and not the real system. The highest level of prototyping contains everything in the previous three levels plus the ability to capture and report on usage metrics.

The function of prototyping is to demonstrate whether or not a model serves a useful purpose. At the first level, we are trying to find out if the screens are discernible; do they portray right meaning. The second level asks whether or not the prototype can respond in an intuitive manner. The third level utilizes scenarios that in turn simulate events to which the user must react. The highest level uses metrics to modify the behavior of the running system. It is important to note that the first three levels also have metrics, but they are not integrated into the prototype; they are external: surveys, video taped sessions, subjective comments of the user community.

USER DISPLAYS

Typically, static mock-up displays are the first prototypes created for most applications. They help determine spatial and size constraints for various data models. Dynamic displays are later generated to allow users to interact with the prototype.

Today's prototypes not only deal with data models, but with user models as well. For example, icons must somehow depict a similar meaning for all users. Supporting this trend is the rapidly increasing role that windowing systems are playing in today's computing environments. Specifically, the method in which information is distributed into windows and icons is important for users who are trying to understand the state of an active system.

New techniques are being developed daily that strive to go beyond the borders of windows of information into what have been termed widgets. Widgets are typically some graphical representation, in the form of an icon or window, that provide movements and actuators upon some object. An example of this type would be a sliding bar widget. In a similar manner to the sliding bars used on stereo equipment, the user can select the slide bar with the mouse and move it along the axis to set or adjust some scalar value. Widget complexity is limited only by the creator's imagination, and they can be as simple as a small radio knob dial or as complicated as the entire front panel of a virtual computer. In general, prototyping systems are becoming increasingly object oriented with data items taking on object properties. These properties can be linked to widget functionality on the display and when an object value changes the corresponding widget can be updated.

This paper will attempt to explain one particular system that was designed to elicit user requirements through the use of prototyping user interactions. The project is called User Requirements Prototyping System (URPS). URPS is positioned at the prototyping interactions (third) level on functionality. This does not mean that the two lower levels (static and responsive) are excluded — they are also available. What we have not included as yet is a method to obtain metrics from the running prototype.

OBJECT RENDERING

Information can be represented (rendered) in different manners. A temperature can be rendered as a number, a picture of a mercury thermometer that has more pixels filled as the temperature increases, or as a square block that changes from blue to red. Any one of these methods may be appropriate in a given situation. Any object can be rendered in some manner, although the method is usually based on object functionality as far as the user interface is concerned.

WINDOWING

It is important to consider the user model as a guide to object rendering. Current windowing systems allow the designer to choose different techniques for window (or object) management. The three main types are tiled, overlapping, and pop-up windows. Tiled windows are those that split up the screen into smaller tiles — no window ever covering up another — and is based upon the user's ability to deal strictly with base spatial concepts. Overlapping windows allow for the possibility of data being covered up and are usually equipped with the ability to resize, move, and place one window over another. In user models terms, overlapping windows represent the "desktop" paradigm. Pop-up windows are interesting in that they can represent a user model that goes beyond the "desktop" into models that are based on a virtual technical assistant working with the user's "desktop." In particular, current pop-up windows are used for displaying a message about the system that you must deal with immediately (like a high priority memo on your desktop); displaying a menu that represents either local or global choices about the window below it; and displaying pop-up windows that act like post-up notes from the system.

DYNAMICS

Allowing dynamic changes to happen on the display is useful. Most user design prototypes find it necessary to know if the user can use and interact with the data that is presented. Current techniques make use of C language (object-code linkability), specially designed scripting languages, or message passing constructs to facilitate dynamics. URPS takes a combined approach in the form of an expert system shell call CLIPS (C Language Integrated Production System). Event messages travel between objects via a FACT construct. Programmability is available at both runtime via CLIPS rules and link time via C code though CLIPS.

CURRENT SYSTEMS

There are many systems currently available for prototyping user displays. Two will be discussed briefly.

The first is a low cost solution available through COSMIC called TAE+ (Transportable Applications Environment Plus). TAE was developed by NASA Goddard as a tool for building consistent, portable user interfaces in an interactive alphanumeric terminal environment. TAE also supports rapid prototyping of user interface screens and interactions, and allows the direct reuse of those screens in the final applications. TAE+ now supports X Window and MOTIF widgets.

VAPS (Virtual Application Prototyping System) is a much more elaborate, commercially available package that runs on silicon graphic workstations. The user can build prototypes by interactively laying out the display and then attaching scripts to each object. The scripts are C functions that are modifiable by the user. VAPS supports a wide range of input devices, and a designer can first prototype a control panel using just graphics and a mouse. Later, a touch sensitive screen can be added. VAPS, a sophisticated product that can prototype very realistic screens, is a product of Virtual Prototypes.

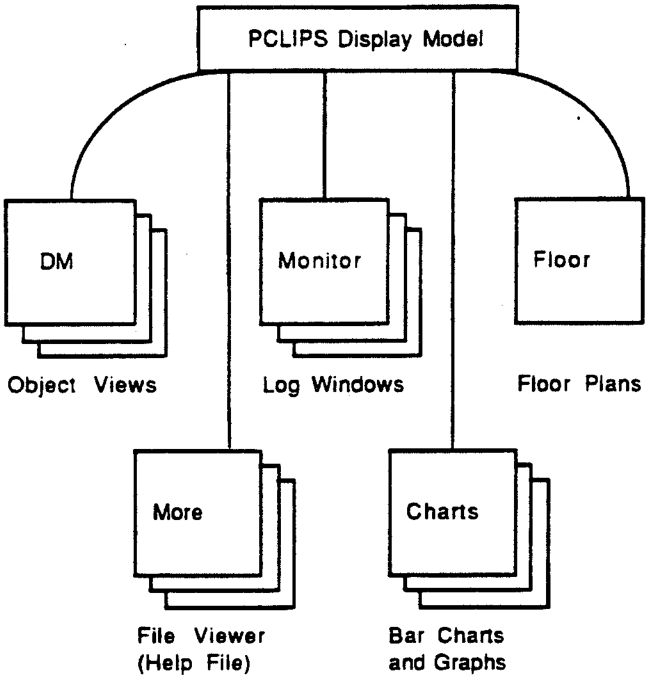


Figure 1. PCLIPS Display Model

DISPLAY MODELS

Rendering Models are based on the display devices available. These devices range from very low capability displays and very high level displays. To examine a few of these differences, three examples will be discussed here: the ANSI terminal, the IBM PC and the X Windowing System.

Using inverted text and special symbols whenever possible, the standard ANSI terminal can provide many rendering possibilities, although tiled windows seem to be the favorite on these systems. It is, however, possible to write, or use, a package can provide both overlapping and pop-up styles. Pictorially, widgets tend to be square and numbers are usually depicted with numerals. Artistically speaking it is possible to have icons that are intuitive.

The next step up from the ANSI terminal the IBM PC. The extended ANSI capabilities of the PC, along with the speed of the system, can support interesting pictorial effects. But one can always choose to tackle the graphic modes (using or buying a package). The biggest problem here is in choosing what level of graphics to support. Bit image graphics on the PC can provide a good medium for widgets; however, screen management is usually still up the programmer.

Lastly, the X Windowing System (and other windowing systems) provide window management features and widget management as well. A detailed explanation of the X Windowing System can be found in other places - it is referenced here to show that display models can vary greatly with device availability.

PROTOTYPING THE ISSS

The original work in this area was done to support the rapid prototyping of the maintenance and control consoles for the Federal Aviation Administration's (FAA) new air traffic control system, the Advanced Automation System (AAS). The purpose of the project is to develop a rapid prototyping system for a man-machine subteam to use in identifying user requirements in terms of the graphical interface. This information could then become the basis for a requirements document for the user interface.

The user displays were separated into functional groups where corresponding object structures and icons were created to represent the various objects. Functionally, the objects represented hardware and software objects that were in some state of usability. Widgets were built using the "traffic light" concept. Green means the object is functioning fine;

yellow means there is a degradation of the object; and red means that the object is dead. Blue is used to represent available but nonallocated resources.

CLIPS is being used as an event-based system. CLIPS is well qualified for this role due in part to the features of the production system model. It addition to events, CLIPS facts are being used to recreate the display model in the form of a fact base (knowledge base). These facts hold the object oriented system data about the actual objects and all the corresponding widget functionality. CLIPS rules function as receptacles for events that occur both by the simulation system and user's (display-based) events. See Figure 1.

PCLIPS is a parallel version of CLIPS that allows multiple CLIPS experts to communicate via a broadcasting function called *remote assert* (*rassert*). By using this method any number of CLIPS experts can be initiated. URPS presently has two: one that serves as a simulation of the prototyped system and another that maps simulation events to the user's screen. A display manager controls usage of the user's screen. Widgets communicate with the display manager in order to gain access to the display space and to update the data.

EVENT-BASED FUNCTIONALITY

There are two major types of widgets: an icon class made up of bit-image graphics and the other, an icon-which is surrounded by a colored box; both represent the state of the object. The box type is our GENERIC class. For this demonstration we have only one icon class; it is called TERMINAL.

```
(deffacts DisplayManager "Base Object Classes for Display Manger"
;
; template: (map-dm-icon <widget-class> <widget-state> <icon-filename>)
; template: (map-dm-state <widget-class> <widget-state> <box-color.)
;
   (map_dm_icon termina up "ik:i_terminal_ok")
   (map_dm_icon terminal down "ik:i_terminal_err")
   (map_dm_icon terminal degraded "ik:i_terminal_warn")
   (map_dm_icon terminal standby "ik:i_terminal_standby")
   (map_dm_icon terminal spare "ik:i_terminal_spare")
   (map_dm_state generic up GREEN)
   (map_dm_state generic down RED)
   (map_dm_state generic spare WHITE)
   (map_dm_state generic standby BLUE)
   (map_dm_state generic degraded YELLOW)
)
```

NOTE: The *generic_display_update* and *icon_display_update* use facts sent from CLIPS to the Display Manager to control widgets. *ask_for_something* receives events from the Display Manager.

```
(defrule generic_display_update "Catch all Generic Status Changes"
   (status ?type ?object ?state)
   (dm_object ?object ?)
   (map_dm_state generic ?state ?signal)
=>
   (rasser dm turncolor ?object ?signal)
)
;
(defrule icon_display_update "Catch only TERMINAL Status Changes"
   (status CC ?object ?state)
   (dm_object ?object icon)
   (map_dm_icon terminal ?state ?fname)
=>
   (assert dm chg-icon ?object ?fname)
)
;
; (Select . . .) facts are remotely asserted by the
; Display Manager when the user does something These
; are much like user events.

; Currently, the default action is to  open up a
; subview.  If the object SELECTed does not have a
; subview, then it does not have a "map_dm_windows"
; fact either.   Another rule with a lower salience
; catches lost User Events in case there is no sub
; view.
;
(defrule ask_for_something "Catch User Events"
 ?rl<-(select ?obj)
 (dm_window ?obj ?w ?h S?Window_Stuff)
 (map-dm-window ?obj ?x ?y)
=>
  (rassert dm open-window ?obj ?x ?y ?w ?h S?Window Stuff
(retract ?rl)
)
```

DISPLAY MODEL FUNCTIONALITY

Functionally, the display is separated into views. These views consist of collections of such widgets as object views, monitor logs, bar charts, and pop-up menus. Object views are windows controlled via remote asserts (*rasserts*) to the Display Manager Screen control, and pop-up windows are also controlled by Display Manager requests. Log windows, bar charts, and the floor plan are separately running programs that join the PCLIPS session upon start-up.

IMPLEMENTATION ISSUES

The Commodore Amiga was chosen as a platform for the following reasons: Low cost, useful resolution (640 X 400), choice of bitplanes, dynamically loadable icons, commercially available image-based tools, and multiprocessing capabilities. The first challenge was porting Clips 4.3 over to the Amiga -- no problem -- just a 5 week delay! The next challenge was in designing the actual display functions. Following this came the PCLIPS functionality; being able to allow multiple CLIPS experts to join together to form a PCLIPS Environment. This was accomplished via the recoding of a PCLIPS server which runs in the background. The server manages incoming requests to join a PCLIPS session and distributes remote *asserts* to all currently listed CLIPS processes. Once we had tools working we were then able to attack the problem of rapid prototyping the ISSS.

CONCLUSIONS

After weeks of designs and redesigns, we have found widgets, object oriented programming and image based icons to be important concepts in the development of new user interfaces. Widget technology is important for encapsulation of data and needs further study. Object Oriented approaches were definitely the way to go in our prototyping system. These approaches were used to determine the level of granularity for the prototype and also to specify functionality of object classes -- no one object was coded better or worse than another in the same class. Image based view facilitated the involvement of art-types who felt they had much more feedom with paint programs than when they were asked to layout displays based on geometrical (graphical) shapes.

Additionally, an interactive configuration tool was created to help in the layout of widgets within views, allowing objects to be positioned over bit-images (pictures). This is part of a far more interesting problem: whether to deal with image based objects or grahpical based (lines, cubes, geometry . . .) objects. One interesting group discussion led to the idea of rendering graphical objects on top of bit image backdrops.

omit

A10 Session:
Artificial Neural Systems and Fuzzy Logic

S/17-61

N96- 12930

p. 8

CLIPS on the NeXT Computer

Elizabeth Charnock & Norman Eng
Pacific Microelectronics, Inc.
201 San Antonio Circle C250
Mountain View CA 94040

358578

Abstract

This paper discusses the integration of CLIPS into a hybrid expert system-neural network AI tool for the NeXT computer. The main discussion is devoted to the joining of these two AI paradigms in a mutually beneficial relationship. We conclude that expert systems and neural networks should not be considered as competing AI implementation methods, but rather as *complimentary* components of a whole.

I. Introduction to NeuExpert

NeuExpert is the name of our system, which is the basis of this paper. NeuExpert was designed for the NeXT compu'er. NeXT was an ideal candidate for this type of development since it runs under Unix, and has an object-oriented programming environment as well as a nice large high-resolution monitor. The intent behind the design of NeuExpert was to make AI as accessible to the end user as possible, and, in particular, to remove some of the stigma associated with neural networks. The incorporation of neural networks was necessary since although neural networks are certainly not the answer to every problem, they *do* represent the resolution of most of the usual complaints against expert systems, namely a sometimes crippling lack of adaptability and flexibility.

The first natural combination of the two methodologies that comes to mind is a partitioning the knowledge space into areas owned by the expert system and areas owned by different neural networks, (neural *units* for short.) Since the granularity of the expert system is much less than that the neural network, this partitioning could be easily achieved by partitioning the knowledge space by *scale*. This can be conceptualized by considering the interval between 0 and 1, from which an arbitrarily large set of rational numbers can be extracted, nevertheless leaving behind an infinite amount of space occupied by the irrational numbers in that interval. Thus the first type of interaction that must occur between expert system and neural network is some arrangement regarding the ownership of different parts of the knowledge space.

II. Basic Integration Strategy

Starting from the previous observation, there are two possible ways to proceed: making the expert system and the neural networks compete for territory as two distinct species competing for turf, or creating an absolutely cooperative relationship in which both expert system and neural network would function as two organs in one body which perform different, but related functions.

We opted for the latter course because most actual knowledge in the brain seems to consist both of highly formalized components *and* completely unformalized hunches. Pursuing this line of thought seems to yield the following "common sense" model which classifies knowledge into three distinct categories:

1. knowledge which is of a *strictly procedural* nature, either because of a lack of real comprehension of what underlies the procedures, because of lack of any experience applying the knowledge, or because there *is* nothing at all underneath. For example, if I were to memorize bus schedules to various locations, the knowledge which I possess could be considered strictly procedural.
2. knowledge which can be considered largely *intuitive*; knowledge derived from extensive experience, or which only exists in the form of triggering associations between items in memory.
3. knowledge which lies in one of the first two categories, but is gravitating towards *joint membership*. This could occur because highly formalized knowledge from the procedural category has become endowed with the added knowledge acquired from experience, or because ideas that began as vague or indistinct associations have evolved into a more formalized representation.

This model clearly suggests a cooperative, rather than a competitive relationship between expert system and neural network. Based on this model, the following interactions between expert system and neural network were created:

a. neural network allocation for a specific rule node by the expert system based on rule usage. The neural network, or neural unit, can be considered "clamped" to that node which we will refer to hereafter as the "parent" node.
b. neural network "feeding," or "starvation," based upon rule usage in the expert system
c. neural network migrations to nodes having a very high positive or negative correlation with the parent node. Neural units which migrate in this way can be thought of as *associators*.
d. neural network migrations due to patterns of rule firings elsewhere in the system which are similar to patterns occurring in a group involving the parent node. Neural units performing this type of migration can be thought of as *concept generators*, since their task is to locate structural*[1] similarities in the information.
e. strong migrating units can actually cause the expert system to leapfrog from its appointed path, or "freely" associate, if permitted to do so by the user.

By *migration*, we mean that the attracting node becomes "close" enough to the parent node from the neural unit's perspective that the firing of the attracting node can cause activity in the neural unit. Conversely, the firing of the parent node can cause the attracting node to fire as a system "afterthought," which is displayed separately to the user.

These interactions allow for information in the third category to be appropriately hybridized. They create a true symbiotic relationship between expert system and neural network. However, the first category of knowledge clearly is a straight expert system application. Unfortunately at this time, the second category does require the training of neural networks. The third category, however, is the most important of the three since most applications that people wish to use expert systems for really fall into this category. The cold hard reality of category three is the reason for the inherent impossibility of "complete" knowledge acquisition.

III. How?

To accomplish this task, we require the basic inference engine machinery, a statistical "state keeper," a neural unit generator and supervisor, as well as an arbitrator to handle such conflicts as arise. The arbitrator keeps track of the current settings of the system parameters which affect all areas of interaction.

We alluded in the last section to statistical correlations. These correlations, along with an overall summary of rule usage, represent the backbone of our extended CLIPS structure. Although they are significant baggage to carry around, they serve three very valuable functions:
1) They provide a basis for optimization and learning *solely* on the part of the expert system
2) They provide the migration paths for the neural units
3) They are used to make the user aware of unusually strong correlations which can represent a bug in the knowledgebase, or a serious gap in the knowledge acquisition. Better still, they could actually be used to point out "new" knowledge in the form of genuine relationships between events which had not been previously noted.

The expert system learning is accomplished through the use of "dynamic" salience values - CLIPS salience values for rules which are updated based on rule usage, starting from the initial salience values (if any) declared by the user. This same mechanism allows the user to define different "experts" having different "experiences," by loading different salience values into CLIPS. In addition, this means that an expert system would learn to behave differently if it were placed in different environments.

To complete the expert system interface to the neural network, we endow the CLIPS structure with three additional properties which are generally associated with neural networks: firing thresholds, back-propagation, and rule learning methods which we will call filter functions.

The firing threshold construct is made possible by our single addition to the CLIPS fact: certainty factors. Rules can be thresholded to different values based on the summed certainty of the information which the rule is acting upon. Thus certainty affects the execution of rules since a rule will not fire if the overall certainty of information does not reach the necessary threshold for that rule.

A loose form of back-propagation has been implemented in the form of a "Reality Inspector," which in addition to providing an explanation facility, allows the user to replace an inappropriate answer with a "better" answer, and have the system readjust salience values appropriately. Since this is a potentially dangerous operation, large changes of this nature are discouraged. The filter functions determine the amount of activity which must occur for a given rule to have its salience value adjusted. We call them filter functions, because they filter out what the user defines as an "irrelevant" amount of stimulation, or lack of it.

To accommodate all of this, we have created a system in which information is cyclically evaluated first by the CLIPS inference engine, okayed or altered by the arbitrator which then checks for the existence of any powerful*[2] neural units which could force a different path from that which was agreed upon by the CLIPS engine and the arbitrator. After this last step is performed the information is passed to the object that handles the graphic display. All updating of system information is performed *after* the session (unless otherwise requested) in order to minimize the amount of time that the user must wait while the system updates.

IV. Computation

Unfortunately this task requires a large amount of computation, and eats a nice chunk of memory for storage. Using the NeXT somewhat minimizes the latter problem, since the optical disks utilized by NeXT are intended for storage of large amounts of information. The computation is a weightier problem. However, there too the use of the NeXT affords an advantage due to the presence of the DSP (digital processing chip) which allows for rapid array processing. Since space is limited, we will concentrate on the computation and maintenance of the correlation data.

The DSP requires all inputs to lie in the interval [-1, 1]. As we will see, this is sufficient for our purposes. The first step of encoding the data consists of constructing a rule network from the rules declared in CLIPS. Beginning from the first "layer" having more then one rule in the network, we determine the boundary of the knowledge space by arbitrarily assigning one of these rules a tag of -1, and another rule a tag of +1. The tag represents ownership of an interval around the tag. Additional rules are assigned tags which are equidistant from one another. For example, if there were only one additional rule it would be assigned a tag of 0. Proceeding to the next layer down, we repeat the process. Only this time, rules descending from a parent must share the portion of the interval which was allotted to the parent. This process continues until the all of the expert system rules have been similarly assigned a tag.

The mathematical set formed from this process is the interval [-1, 1] - {the set of boundary points between adjacent owned territory.} Physically, the set can be thought of as a dotted line with the number of gaps in any part of the line being proportional to the number of rules occupying that part of the interval. Each time the system is run to completion, this skeleton set is "filled in" to show which rules fired during that run.

The path taken by the system is like a mathematical footprint which describes the order of rule execution. All of the necessary information is derivable from this set. A new image of the system state is created for each complete system run. These images can be stacked on top of one another to create a pictorial as well as mathematical three dimensional system history.

Since one of the system's tasks is to alert the user to unusually high correlations between rule firings*[3], the system must be continually aware of the occurrence of these events. This is accomplished through inspecting different similarly sized peaks to see if the contexts of the rules firing matches up with the absolute number of firings. Clearly a peak located in a subpartition of another peak's interval is not of interest, since the first rule would be a direct ancestor of the second rule. If an unusual correlation is discovered it is reported to the user, who can then decide if some modification is merited.

V. The Neural Units

There is not sufficient space for an extended discussion of the neural units, however a few words on the subject are merited. Clamped neural units with a small number of hidden layers can appear either through system allocation, or by user request. The system will notify the user each time it adds a new neural unit. The user will then be requested to specify a set of inputs and outputs, and may then begin training. If the user chooses not to train the neural unit, the system will *not* remove the unit unless explicitly instructed to do so by the user, but instead will train it randomly. This is because the allocation of a neural unit is a *considered* action on the part of the system which is intended to inform the user that some system attention should be devoted to the area of the knowledge space where the neural unit was placed. The training of the neural unit is quite similar to the previously described "Reality Inspector," in order to minimize the difference between the two components.

Just as in the expert system case, very frequently used neural unit output nodes will spawn addition neural units following the conventions described above. The system will have a small selection of accepted learning methods from which the user may choose. The user does not directly control the migration of the neural units, although he/she can adjust some system parameters which will affect the neural units' definition of sufficient proximity for migration. Determination of "proximity" in the case of the so-called concept generators is a "hard-wired" behavior of neural units which are entirely concealed from the user.

The neural units described in this paper are clearly differentiated in purpose, as well as "physical" appearance: for example, neural units which have migrated to other nodes maintain an umbilical cord from the parent node. This differentiation results from a combination of the experiences which the expert system as a whole happens to have, as well as the placement of the individual unit. On a conceptual level this is quite similar to contemporary neurobiological models of neuronal differentiation in the brain.

In summary, the neural units augment the capabilities of the expert system by providing recognition of detail and imperfect instances, gap-filling in the knowledge acquisition, and the important power of association between vaguely similar items. As with human beings, there can be no guarantee that every association is a valuable, or relevant one. Yet it is inarguable that much of human memory and reasoning ability stems from the capability to recognize unformalized similarities between otherwise unrelated pieces of information.

VI. Other Supporting Features

In addition to a standard windowed environment, our system supports several user interface oriented features worthy of discussion since they augment the value of the system capabilities which were discussed in the previous sections.

We have added the construct of rule "groups." This is a construct which is completely external to CLIPS, but which is useful to represent system progress to the user. Our system allows the user to specify execution paths with the mouse that temporarily override all other existing salience values. The group construct is also used for many of the graphics representations discussed below.

Rules can also be defined as objects in a limited sort of way. Since our rules have eight attributes in all (name, definition, category, salience, threshold, object type, filter function and group,) it is inconvenient for the user to have to replicate this information for every rule in a large set of rules which the user wishes to have several identical attributes. Thus an object type is determined by a user defined ID name, the number of traits the user wishes the rules to have*4 and any default settings, such as a threshold value. In addition, each rule group must have a filter function associated with it that all of its rules possess. Object type is also completely external to the CLIPS engine.

Although both rules and facts have optional "category" slots for reference purposes, our system offers additional aid for knowledge extraction in the form of a "librarian" which maintains a record of the *context**5 of rule usage. This corrects for any errors or omissions made categorizing the information initially.

Another important system feature is the presence of the five different graphic visualization methods which are part of the knowledge debugging environment. These representations encourage the user to view data in different ways which accent different traits of the data. However, the most important function of these different modes is making the system as transparent as possible to the user. Doing so makes the task of using the system more interesting because it involves genuine comprehension of the underlying knowledge, as well as some degree of demystification about the inner workings of the system. In addition to switching between graphic modes the user may take "snapshots," from different perspectives which yield different simultaneous views. The user may collect these snapshots in a "photo album."

The different graphic visualizations are as follows:

1. a rule network, in which each rule is represented as a node connected to other nodes which can result from its firing. Under this mode, the user may view the correlations between different nodes which are illustrated as connecting lines whose width varies according to the degree of the correlation, and whose coloration varies according to whether the correlation is positive or negative. The user may also graphically view the different rule thresholds as well as the certainty with which each rule fired following the end of a completed run. Neural units are visible as entities, but without any detail.
2. a condensed overview of the entire system which, in a biological analogy, depicts the regions of greatest neural unit activity as having "denser tissue." Individual regions can be more closely examined with a "microscope"
3. a "fuzzy" view showing a partition of the knowledge space which accents the uncertainty of information using a fuzzy set representation.
4. a moving drivers' seat view graphically depicted by the entirely on the graphics screen filling with a graphic object representing the rule group currently being examined.
5. a text-based explanation mode

This diversity of viewpoint is necessary to ensure that the user is able to comprehend the information which the system is acting upon. We believe that one of the principal functions of an expert system should be helping its user to better understand the expert system's *knowledge*, not just the expert system. Thus, an expert system can be thought of chiefly as fulfilling a knowledge distribution function, while increasing its knowledge store both through its own experience and through modification by very "knowledgeable" users.

VII. Conclusion & Summary

We have discussed the integration of a CLIPS-based expert system and neural networks in a unified, cooperative system. Our conclusion is that these two AI methodologies should not be viewed as antithetical to one another, but rather as naturally symbiotic partners operating in complimentary portions of the knowledge domain.

Footnotes:

*1 structural in terms of data involving patterns of occurrence.
*2 A powerful neural unit is one that has been very well "fed" by the expert system.
*3 that is, non-trivial rule firings. The relationships between the firing of rules directly descended from one another is not of interest in terms of providing data for associations.
*4 For each new rule, only name and definition *must* be entered. Category and group are optional: if a rule is not assigned one, it does not receive any default value. If no threshold is defined, the rule is presumed to have none. If no salience is defined, an average salience value is assigned by the system. If no object type is defined, the rule is presumed to be of the generic type.
*5 By *context* we mean the knowledge category context. The system keeps track of what each rule was used for, based upon the category of the end result. This ensures that all of the system knowledge necessary to operate in some subdomain of the system is extractable.

5/8-63

N96-12931

358582

p.4

CLIPS:
A Tool for Corn Disease Diagnostic System and An Aid to Neural Network for Automated Knowledge Acquisition

Cathy Wu, Pam Taylor, George Whitson and Cathy Smith

Department of Mathematics and Computer Science
The University of Texas at Tyler
Tyler, Texas 75701-6699

ABSTRACT

This paper describes the building of a corn disease diagnostic expert system using CLIPS, and the development of a neural expert system using the fact representation method of CLIPS for automated knowledge acquisition. The CLIPS corn expert system diagnoses 21 diseases from 52 symptoms and signs with certainty factors. CLIPS has several unique features. It allows the facts in rules to be broken down to <object-attribute-value> (OAV) triples, allows rule-grouping, and fires rules based on pattern-matching. These features combined with the chained inference engine result to a natural user query system and speedy execution.

In order to develop a method for automated knowledge acquisition, an Artificial Neural Expert System (ANES) is developed by a direct mapping from the CLIPS system. The ANES corn expert system uses the same OAV triples in the CLIPS system for its facts. The LHS and RHS facts of the CLIPS rules are mapped into the input and output layers of the ANES, respectively; and the inference engine of the rules is imbedded in the hidden layer. The fact representation by OAV triples gives a natural grouping of the rules. These features allow the ANES system to automate rule-generation, and make it efficient to execute and easy to expand for a large and complex domain.

INTRODUCTION

Many criteria can be used to evaluate an expert system: the accuracy and efficiency, the ease of use, the ease of initial building and later expansion, and extra features such as the explanation facility and certainty representation. Diagnostic rule-based expert systems are among the most important and successful expert systems. However, the implicit nature of the domain knowledge has made it difficult to develop new expert systems on different domains even with the available rule-based expert system shell, because it requires explicit rules. Artificial neural system [4] has been used as an alternative approach to build diagnostic expert systems to overcome the knowledge acquisition bottleneck [1,3,5,7]. On the other hand, the neural systems lack a built-in explanation facility and a natural query system. Furthermore, the representation of the domain knowledge in a large single set of values makes the neural expert systems not suitable for a large and complex domain.

This paper descibes and compares the two corn disease diagnostic systems, one rule-based using CLIPS [2] and one neural network using ANES [6]. The paper also shows the automated knowledge acquisition scheme used in the ANES corn system with a direct mapping to CLIPS system. The fact representation method in both systems allows the rule-grouping and result to speedy execution, natural query system, and easy system expansion.

CLIPS CORN EXPERT SYSTEM

CLIPS, a rule-based expert system tool developed at NASA, is used to build the corn disease diagnostic system that identifies 21 diseases from 52 symptoms and signs. The facts are broken down to <object-attribute-value> (OAV) triples. Each object in the OAV triples has two components: <plant_part> and <pathogen_type>. There are five plant_parts, namely, seedling, whole_plant, leaf, stalk_or_root, and tassel_or_ear; and three pathogen_type, fungus, bacterium, and virus. The attribute is the <descriptor>, which can be a symptom or sign or a disease. The facts for the 52 symptoms and signs are grouped into ten fact lists (ie., ten deffacts), five for symptoms on five plant_parts each and five for signs on five plant_parts each. The fact template for symptom or sign has the form of: (<plant> <pathogen>

<symptom/sign> <value>). Figure 1A shows the fact lists for signs on seedlings.

The fact template for disease, however, needs two additional fields, a Certainty Factor (CF) field, and a tag field. The certainty factor ranges from 0 to 1, to indicate the degree of confidence for the firing of certain disease(s) (RHS of the rule) from the observed symptom or sign (LHS of the rule). To tag each fact uniquely, a unique tag is generated for each disease fact (OAV triple) using the gensym function [2]. Thus, the final fact template for disease has the form of: (<plant> <pathogen> <disease> <value> <CF> <tag>). There are 52 IF-THEN rules (ie., defrules) that associate each one of the 52 symptoms or signs to its related disease(s) (Figure 1B). The same OAV triples that are derived by separate rules are combined to produce a single OAV triple with a combined certainty factor (Figure 2A).

CLIPS fires rules based on a pattern-matching mechanism. The fact representation method combined with the pattern-matching mechanism creates a natural rule-grouping. The priority of the firing of each rule group can be further controlled by the use of salience. In the corn expert system there are 15 rule groups, each corresponding to an object (ie., a plant_part and pathogen_type combination). The rule-grouping mechanism and the chained inference engine result to a speedy execution. Furthermore, the rule-grouping provides a natural user interface to query only a subset of symptoms or signs in order to reach a conclusion (Figure 2B). This makes it unnecessary to emulate the backward chaining inference engine commonly used for goal satisfaction. However, the emulation of backward chaining in CLIPS is fairly straightforward. For example, one can simply add a fact list that relates all diseases with their associated symptoms and signs for the back tracking and let the rule fire in the normal forward fashion.

ARTIFICIAL NEURAL CORN EXPERT SYSTEM

ANES is an artificial neural expert system tool developed at the University of Texas at Tyler that uses back-propagation network [6]. With ANES, it is possible to build a diagnostic expert system by mapping the symptoms/signs directly to diseases without knowing the exact contribution (with certainty factor) of individual symptom/sign to a particular disease. The former is the implicit knowledge of a domain expert. The latter is an explicit if-then rule derived from the implicit knowledge by the domain expert through a time-consuming process.

The facts are represented by the same OAV triples that are used in the CLIPS system. Each input fact of a rule, a triple, is converted to an input vector of 32 neurons (Figure 3) by a preprocessor, while each output fact of a rule is obtained from the output vector by a postprocessor. Thus, the input (LHS) and output (RHS) facts are mapped into the input and output layers of the ANES, respectively; and the inference engine for the rules are imbedded in the hidden layer (Figure 4). The fact representation by OAV triples gives a natural grouping of the rules. There are two rule groups in the corn ANES: rule group 1 to connect symptoms and signs to diseases, and rule group 2 to determine a disease from all possible candidates (Figure 5). Rule group 1 consists of 4 subgroups, each of which corresponds to symptoms, fungal signs, bacterial signs, and viral signs, respectively. The subgroup in turn consists of five rules, each associates symptoms or signs on a particular plan_part to certain diseases.

Because of the rule-grouping mechanism of the ANES, the system can be implemented onto a parallel architecture to break down one large neural networks to many small parallel networks [6]. This would speed up execution and make the expansion of the knowledge base much easier. The direct mapping of the CLIPS corn expert system to ANES using the same rule-grouping mechanism allows the development of an automated knowledge acquisition scheme. The ANES inference engine is capable of extracting the implicit knowledge embedded in the neural network [6].

CONCLUSION

Both CLIPS and ANES expert system tools produced corn diagnostic systems that diagnose accurately and are easy to use. The representation of facts using OAV triples in both systems allows the grouping of rules, which speeds up the execution, provides a natural way to break down a complex system to subsystems, and allows a chained inference and natural query system. Building an expert system using ANES is easier, however, because of the automated knowledge acquisition.

SYMBOLS AND ABBREVIATIONS

ANES (Artificial Neural Expert System), OAV (Object-Attribute-Value), RHS (Right-Hand Side),
LHS (Left-Hand Side).

```
A.      (deffacts seedling-sign
                  (seedling fungus sign mycelia-and-spores)
                  (seedling bacterium sign bacterial-droplets))


B.      (defrule mycelia-and-spores
                  ?has <- (it has seedling fungus sign mycelia-and-spores)
        =>
                  (assert (seedling fungus disease seedling-blight .7 =(gensym)))
                  (assert (seedling fungus disease root-rot .7 =(gensym)))
                  (retract ?has))
```

Figure 1. Fact (A) and rule (B) representation in CLIPS

```
A.      (defrule combine-CF
                  ?fact1 <- (?plant ?pathogen disease ?name ?CF1 ?)
                  ?fact2 <- (?plant ?pathogen disease ?name ?CF2 ?)
                  (test (neq ?fact1 ?fact2))
        =>
                  (retract ?fact1 ?fact2)
                  (bind ?CF3 (+ ?CF1 ?CF2))
                  (assert (?plant ?pathogen disease ?name ?CF3 =(gensym))))


B.      (defrule goal
                  (?plant ?pathogen disease ?name ?CF ?)
                  ?sym-sign <- (?plant ?pathogen ? ?value)
        =>
                  (if (> ?CF .8)
                  then
                  (fprintout t "The disease may be a(n)" ?name crlf)
                  (fprintout t "Certainty Factor is  " ?CF crlf)
                  else
                  (fprintout t "Has it " ?value "on " ?plant crlf)))
```

Figure 2. Rules for (A) combining certainty factors, and (B) goal satisfaction in CLIPS.

32 Input/output neurons for each OAV triple

8 Objects		3 Attributes	21 Values
5 Plant_Parts +	3 Pathogen_Types	3 Descriptors	21 Diseases
Seedling	Fungus	Symptom	or
Whole_plant	Bacterium	Sign	Symptoms/Signs
Leaf	Virus	Disease	on Plant_Part
Stalk_or_root			
Tassel_or_ear			

Figure 3. Mapping of facts (OAV triples) to input and output vectors in ANES.

```
Fact in-1            ->     Rules        ->    Fact out-1
    8 Objects                                      8 Objects
    3 Descriptors                                  3 Descriptors
    21 Values                                      21 Values
Fact in-2                                      Fact out-2
    .                                              .
    .                                              .
Fact in-M                                      Fact out-N

Input Layer              Hidden Layer          Output Layer
(LHS Facts)                (Rules)             (RHS Facts)
```

Figure 4. Fact and rule representation in ANES.

```
          Rule Group 1                               Rule Group 2

          Rule Subgroup 1

                    r1                        Diseases        r21
31 Symptoms         r2    21                  Identified      r22
on 5 Parts  ->      r3 -> Diseases            From       ->   r23  ->  Disease
                    r4                        Group 1         r24
                    r5                                        r25

          Rule Subgroup 2

                    r6
Fungal Signs        r7   Fungal
on 5 Parts  ->      r8 ->    Diseases
                    r9
                    r10
                    .
          Rule Subgroup 3
                    .
```

Figure 5. Rule groupings in ANES

REFERENCES

1. Gallant, S.I. 1988. Connectionist expert systems. Communications of the ACM 31(2): 152-169.
2. Giarratano, J. and G. Riley. 1989. Expert Systems: Principles and Programming. PWS-KENT Publishing, Boston.
3. Kulkarni, Arun D., George Whitson, Jim Bolin and Cathy Wu. 1989. Some applications of the parallel distributed processing models. Proceedings of Workshop on Applied Computing '89: 185-192.
4. Rumelhart, D.E. and J.L. McClelland. (eds). 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations. A Bradford Book, MIT Press.
5. Saito, k and R. Nakano. 1988. Medical diagnostic expert system based on PDP model. Proceedings of the IEEE Second Annual International Conference on Neural Networks I: 255-262.
6. Whitson, George, Cathy Wu and Pam Taylor. 1990. Using a neural system to determine the knowledge base of an expert system. Proceedings of the 1990 ACM Sigsmall/PC Symposium on Small Computers: 268-270.
7. Yoon, Y.O., R.W. Brobst, P.R. Bergstresser and L.L. Peterson. 1989. A desktop neural network for dermatology diagnosis. Journal of Neural Network Computing, Summer: 43-52.

A Neural Network Simulation Package in CLIPS

Himanshu Bhatnagar, Patrick D. Krolak, Brenda J. McGee, John Coleman.

Center for Productivity Enhancement, University of Lowell, Lowell, Ma. 01853

ABSTRACT

The intrinsic similarity between the firing of a rule and the firing of a neuron has been captured, in this research, to provide a neural network development system within an existing production system (CLIPS). A very important by-product of this research has been the emergence of an integrated technique of using rule based systems in conjunction with the neural networks to solve complex problems. The system provides a tool kit for an integrated use of the two techniques and is also extendible to accommodate other AI techniques like the semantic networks, connectionist networks, and even the petri nets. This integrated technique can be very useful in solving complex AI problems.

1. INTRODUCTION

Direct hardware implementation of Neural Networks is not always easy and hence there is a need for simulating them through computer software. Early examples of software simulation models can be found in [1] and [2]. These and the other simulation models primarily simulate the neural states, neural architectures and connection strengths, and implement the tools to manipulate them. Several learning techniques (rules) have been proposed in the Neural Network literature, one of them being the generalized delta rule (or Back Propagation)[3]. Our first level goal is to provide a more efficient package, in CLIPS, for simulating neural networks employing back propagation, together with expert systems.

CLIPS is an expert system shell developed by NASA [4], which provides a LISP like interface and allows both forward and backward chaining. The production rules, under forward chaining, have facts on the lhs and action commands on the rhs. When facts, in the facts database, match the lhs of any rule that rule fires, possibly causing assertion of more facts and hence firing of other rules. In a binary neural network, a neuron fires when its activation has exceeded its threshold value. There is an inherent similarity in the way rules fire in an expert system and the way neurons fire in a Neural Network, suggesting the modeling of one in terms of the other, and hence CLIPS can prove to be a very effective simulation tool for Neural Network modeling. We, at the Center for Productivity Enhancement, University of Lowell, have developed a shell called Neural CLIPS, or N-CLIPS which allows Neural Network Simulations to be built, tested and implemented along with regular expert systems. N-CLIPS provides a common environment for development, implementation and operation of two competing and radically different artificial intelligence techniques : the C Language Integrated Production System (CLIPS) for writing expert systems and a Neural Network system. These systems can either operate independently to solve different classes of artificial intelligence problems or can cooperate to help solve much bigger AI problems [9]. In [6] Rabelo has shown the usefulness of combining the neural networks and the expert systems. Knowledge representation, acquisition and manipulation, decision making and decision support are the major characteristics of these techniques and hence when they are used together they can share knowledge and can share the decision making process itself.

To further emphasize the importance of such a common platform we are using it to model a traffic control system for mobile robots operating the Material Handling System of a Flexible Manufacturing System based factory [7]. The (simulated) mobile robots have on-board neural networks which work together with expert system modules to guide them through the factory floor without collisions and with minimum delays. Since CLIPS provides an excellent interface with C, these expert system rules can interact with other processes and also interact with different types of peripheral hardware [5].

The next section provides a brief description of the terms relevant to neural networks, followed by a survey of the features common to currently available simulation packages. The need for integrating AI techniques is discussed next followed by a description of N-CLIPS. The last section gives a detailed explanation of the system developed.

2. ARTIFICIAL NEURAL NETWORKS

2.1 Definitions

For our purposes a **neural network** is a densely connected, possibly layered, network of simple processing units (**neurons**). The connections, known as **synapses,** are weighted links between two such units where the **weight** of a link is modifiable, and determines what fraction of the signal, between the two units, is actually passed. A negative weight usually signifies an **inhibitory link**(synapse) which causes an inhibitory effect on the firing of a post-synaptic neuron. A positive weight usually signifies a **excitatory link** which excites the neuron to which it is connected.

Neurons, in the network may be classified into three types depending on the roles they play. They are either **input neurons** (input layer), **output neurons** (output layer) or **hidden neuron** (hidden layer) depending on whether they accept input from outside world, provide an output to the outside world or receive input from units within the system and generate output for the units within the system. Processing within a neuron may be divided into three stages : a) determination of net input to the neuron ; b) determination of neural state (an **activation function** associated with a neuron determines the state); and c) determination of the neural output (an **output function** determines the final output value).

2.2 Learning

The two major learning paradigms available currently are: generalized delta rule (GDR) or back propagation [3] and its variations for both feed forward and **recurrent networks**[16], and **hebbian learning,** with its sophisticated variants (by which we mean to include methods employed in Bi-directional Associative memories and other associative memory models) [10][17][18][19] [20].

2.3 Generalized Delta Rule

In the initial phase of our work we have focused on the GDR as applied to feed forward networks. In this approach a set of patterns is repeatedly presented at the input layer of a multilayered network. The output pattern generated is compared with a target pattern. The difference is propagated back and is reflected as a change in the weights of the links, all the while minimizing a global energy function (**mean squared error** function). The difference or the **delta** is used to modify the weights of links between neurons. This process is repeated till the actual pattern is within a close range of the target pattern, for a particular input pattern. This is done for each input pattern.

3. EXISTING SIMULATION PACKAGES

A brief survey of most of the commercial neural network simulation and development packages reveals the following characteristics :

* A strong user-interface : Pop-up menus within a windowing environment, a file system and interface with major database systems for I/O.

* Types of Learning Paradigms supported : All major learning paradigms along with their variations.

* Capability for Customizing and designing user-specified Neural Nets : Ranges from just setting up of network parameters to script based design of neural networks.

* Debugging & Interaction tools : On-line graphical editing of a neural network; pausing, restarting and saving snap shots of neural nets during different states of their operation: displaying weight change, delta change, noise and a host of other features.

The different information processing paradigms are particularly well suited for the problem domain in which they evolved. However, when addressing classes of problems that span more than one domain an integrated approach seems attractive. This approach involves several different AI techniques. The inter-relationships of these techniques is still not well understood and there is a need to study their interaction with each other. None of the systems available today have the capability of providing a common platform to investigate these 'inter-relationships'. In N-CLIPS we provide a common playing ground for at least two of these, with the capability of extensions to accommodate others.

4. WHY CLIPS ?

By extending CLIPS to accommodate neural networks, semantic networks, connectionist networks and other knowledge representation techniques, we, will have a tool to understand their complex inter-relationships and the mapping of one technique into another. In real life systems we need the precision of expert systems, the localized representation of semantic networks and the flexibility of neural networks all encompassed into one. This is so because each of these techniques have strengths which compliment the weaknesses of the other. The brittleness of expert systems can be supplemented with the plasticity of neural networks on one hand and the lack of precision of neural networks can be substituted by precise rules and facts. Adding new knowledge to an expert system is quick (as a new rule) but its interaction with the existing rules can be of a conflicting nature. On the other hand adding a new pattern to a neural network takes a long time but can be made to interfere minimally with the old patterns. On a factory floor, new situations can be quickly learned by plugging in temporary rules. However, over a period of time, these rules get to be unmanageable and redundant and have to be trimmed. They can be collectively mapped into a neural network which could iron out the conflicting rules, and once trained it can be mapped back to a more parsimonious set of rules. To illustrate this further, assume a set of rules which do not trigger each other. The combinatorial arrangement of the union of facts on the lhs of these rules and the actions on the rhs can be translated to the input and the output patterns of a back propagation neural network (BPNN). Out of the available output patterns the ones actually needed can be selected without difficulty. Then by applying the inverse mapping technique proposed by Williams [11] where the input values (at the input layer) instead of the weights are modified via back propagation of error, the neural network can be converted back into an expert systems. Of course, a major problem to be considered in this process is that of knowledge representation since patterns must be translated into facts. In addition there may be many-to-one mappings that are dependent upon initial states of the system.

Sometimes, at a higher level of design the localized representation of a problem can be done through semantic networks and the rest as expert systems and neural networks . For example the higher level path planning of mobile robots on a factory floor can be done using semantic networks, while the low level path planning and traffic control can be done by expert systems which in turn depend on neural networks for decision support. As can be seen all three models will need to communicate with each other. CLIPS allows that via rules and facts, moreso because all of these techniques shall have rules and facts as their building blocks.

Another example would be the cooperative use of multiple neural networks for mortgage underwriting and industrial parts Inspections [13][14][15]. In [13] the system is a collection of nine coupled sub-networks have three sub-networks acting as 'experts' and their cooperative effect helps in validating the confidence level of the decisions made by the whole system.

The major functions which were added to the existing CLIPS code have been briefly explained in Appendix A. The engine for neural networks manipulates its own data structure but eventually uses clips' agenda and fact lists to let the clips execute the neural network. The functions listed in the appendix are driver, nassert, add_nfact, ncompare, ndrive, nretract. PCLIPS [8], a distributed version of CLIPS has also been developed at the university.

5. N-CLIPS

This shell provides an object oriented approach to problem solving in the neural network and fuzzy logic domain and at the same time maintains the integrity of the CLIPS production system. The expert systems and sub-systems can be written as rules and facts while a neural network is represented as a collection of objects and a set of actions to be performed on them. It provides well known neural network learning paradigms as objects which the user can use to map their problems onto or use them as subsystems of more complex user-designed neural networks. Users can also build their own variations of the existing paradigms and can also create their own learning rules and models within the given environment. A library of functions for creating and editing neural network objects like neurons, synapses, activation functions and layers is made available to the user. The *ntrain* and *nrun* functions are a collection of rules linked with facts which can be invoked to train a neural network or execute it. The rules and facts making up the expert systems are written in the same way as in regular CLIPS. At the lowest level of expert system-neural network communication the two systems interact via rules and facts. However, at a higher level, complex but abstract interaction is possible. For example the neural network actions, composite and primitive, can be written as a set of rules linked with facts while an expert system can spawn off a neural network to extract useful information from available fuzzy or smudged knowledge. This system can also be used as a first level tutor for

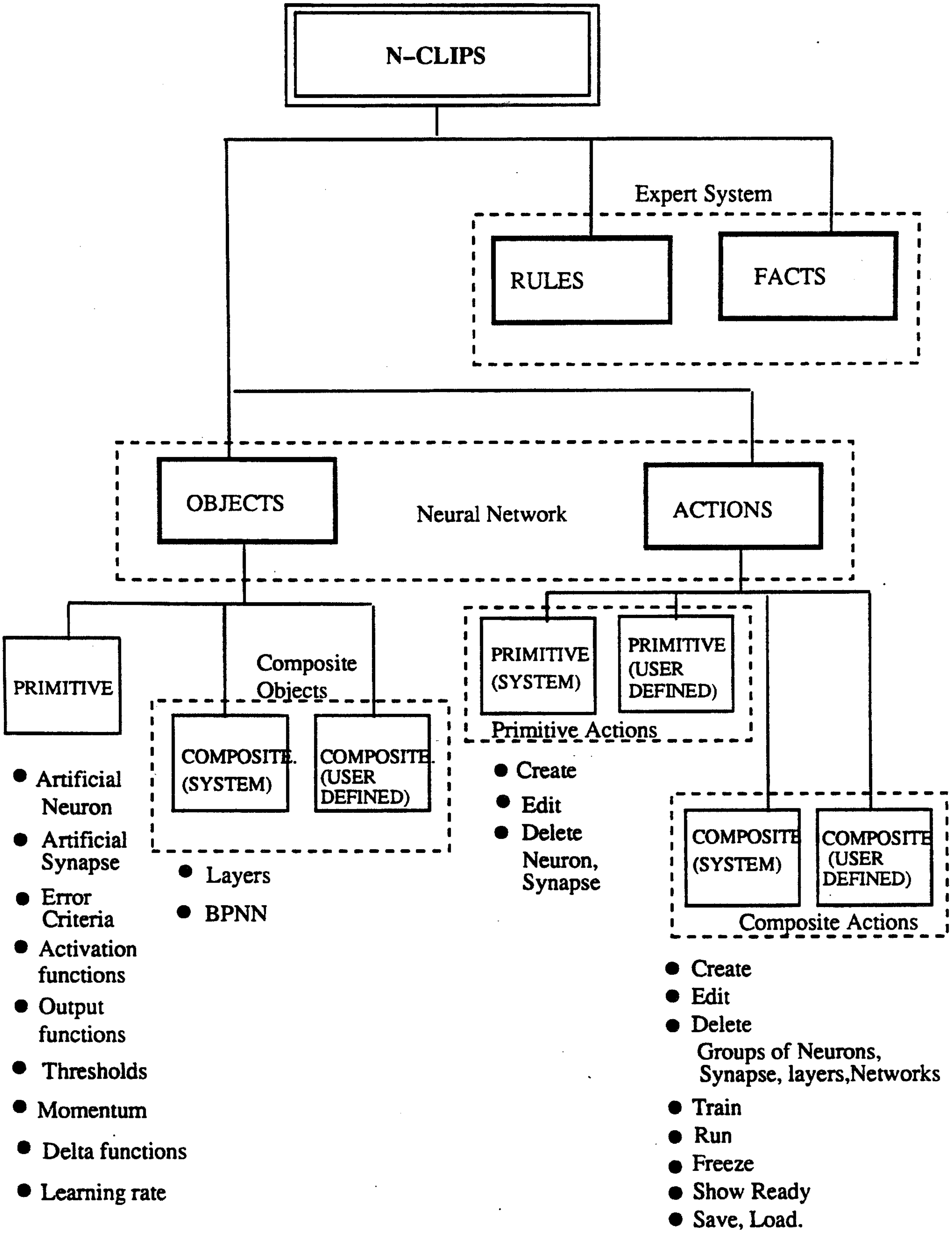


fig 1 : N-CLIPS : A hierarchical description

understanding basic existing models. CLIPSs' capability to interface with other languages viz. C, Ada is exploited for a graphical (X-Windows and/or Motif) user-interface and a file-system interface for saving snap shots and networks themselves. In this system the following graphics user-interface is available :

- * Neural Network interconnection diagram.
- * CLIPS rules interconnection diagram for seeing which rules fire which other rules and on what basis.
- * Mouse interface with the Neural Network diagram.
- * 'Click-on-connection-for-weight-change' graphical facility.
- * Change of color if a node fires.
- * X-Windows link editor.
- * X-Windows weight editor.

The file system interface allows saving and loading of neural networks via save_nn() and load_nn() functions, at any instance.

6. SYSTEM DESCRIPTION

6.1 OBJECTS (Primitive)

6.1.1 Artificial Neuron

An artificial neuron is basically of three types i.e. Input, Output and Hidden. Its major characteristics (for back propagation) are an identifying number, layer number, an activation and output function, threshold value and its type. These parameters could be either passed to a C function call or through a template invoked from the CLIPS interpreter. After the parameters of a neuron are accepted from the user they are encoded as a special rule in a string which is then compiled and loaded into the network. These parameters could be edited and a complete neuron deleted at any given instance. Internally in CLIPS the specifications of a neuron are also stored within a data structure (see fig. 2). Any modification of a neuron's specifications are automatically reflected in the data structures and the associated rule. A deleted neuron will also result in deletion of all the connected links.

The composition of the special rule (for back propagation only) is as follows :

```
(defrule artneu#
    ? neu  <-  (neuron # layer # ready to fire)
=>
      (nretract ? neu)
      (propagate layer #)
      (calculate_delta layer #)
      (change_weights from layer # to layer #)
)
```

On the rhs the function propagate(), propagates the output signal to the next layer neurons after duly multiplying it by the strength of the connection of the links. The next function calculate_delta(), calculates the deltas based on the error signal propagated by the succeeding layer and stores them in the data structures. Finally, the change_weights() function changes weights based on the calculated deltas. These functions manipulate the network data structure (fig. 2)

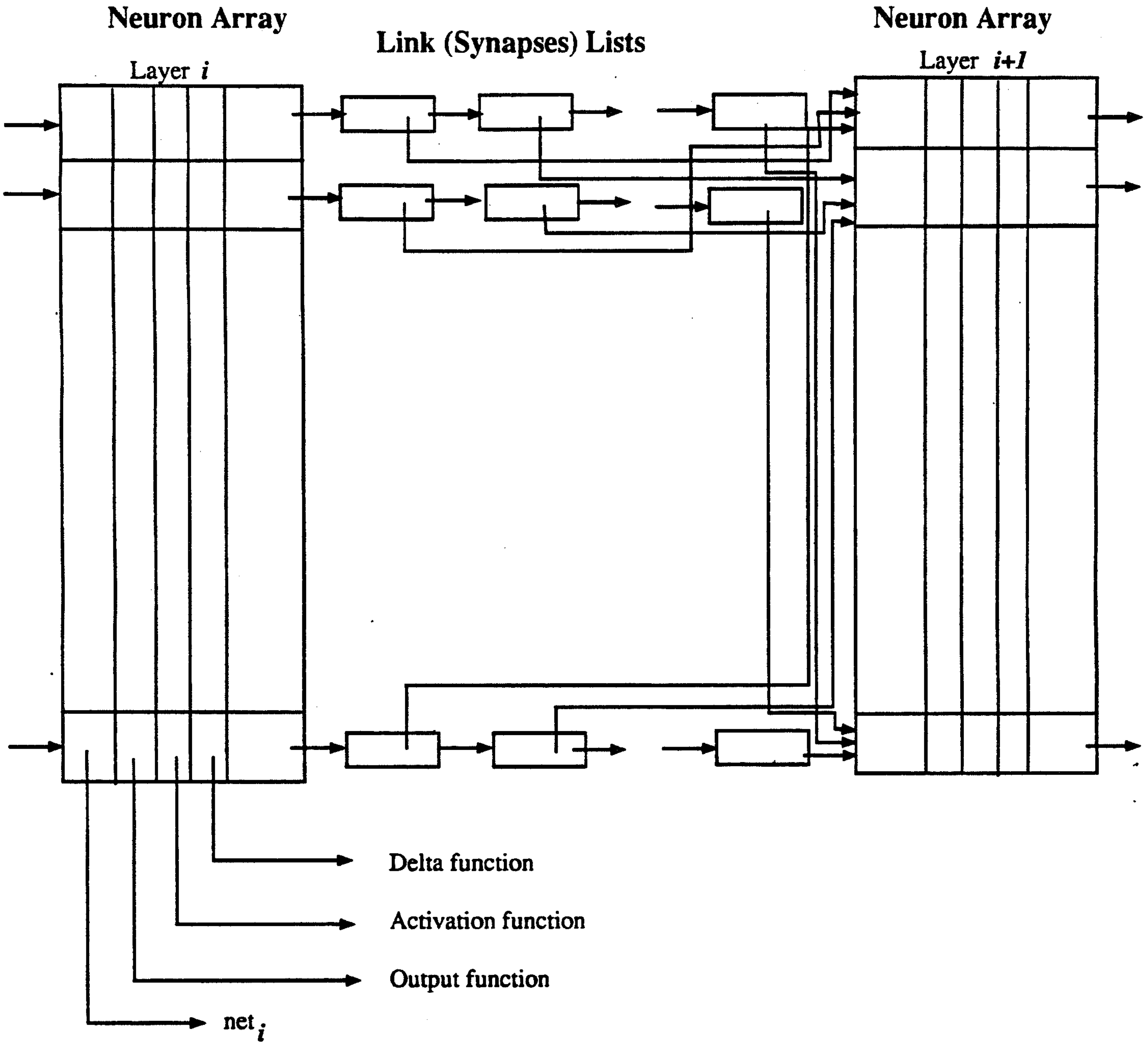


fig. 2 : A sample Data structure for storing a Neural Network in N-CLIPS

for performing the above mentioned functions. This neuron is specifically suited for representing the hidden layers of a feed forward neural network. The rules for input and output layer neurons are slightly different. These special rules can be modified via functions provided in the system to represent any other kind of neural network model. A more generalized model of a neuron is in design.

6.1.2 Artificial Synapse

These are the links between neurons, and are mainly characterized by the following parameters: 'from' and 'to' neuron # and layer #, the type (in or out link), weight. They are stored in a special data structure (see fig. 2) and can also be stored as facts; as in the case of the outgoing links from the output layer neurons. They can be created/edited and deleted as individual links or as a group (from one layer to another). Individual links can be created as C functions or from within CLIPS interpreter (a template possibly from within a windowing system) and group links can be created through a X windows graphics link map editor (explained later). This way fractional (percentage of total neurons) connectivity between layers can be represented very easily.

6.1.3 Activation functions

A library of different existing activation functions is provided to which a user can add a function or modify or delete a function. These functions can be selectively applied to individual neurons or to a group of neurons.

6.1.4 Input/Output functions

Different input/output functions, for neurons in the input/output layers, which are currently popular are provided in a library. The user can add, modify or delete a function from the library. The user can select a function from this library to apply to a single neuron or to a group of them. The input function is usually a linear function, nevertheless a different input function can also be provided. Also for single layer feature maps [10] the input functions could be much more complex. In N-CLIPS this complexity can as well be mapped directly in a neuron rule.

6.1.5 Threshold types

A high pass threshold is the most general type used, where if a neuron's activation is above a certain threshold it fires. A low pass threshold type is characterized by its ability to allow a neuron to fire only if its activation is below a certain threshold. The band pass (and the multiple band pass) threshold types [12] are applied when a neuron fires if its activation is within a single range of values or several ranges. These are available as choices when the user is describing a neuron and can be applied to a solitary neuron or a collection of them.

6.1.6 Constants of the Equations

The constants applied in the various equations can be changed during the network training sessions via the user interface provided by the system. Momentum factor, and Learning rates are two such constants which are applicable to the back propagation neural networks. Different momentum factors and learning rates can be applied to different parts of the network.

6.1.7 Delta functions

Delta functions, as prescribed in [3], are available in this system. Users can also add customized delta functions to the library.

6.1.8 Error Criteria

While the mean squared error is the most generally used error function, and is the one currently supported, future extensions will provide for other error criteria (e.g. entropy).

6.2 OBJECTS (Composite)

6.2.1 Layers

This system provides both layered and non-layered neural networks. Neural layering allows for grouping of neurons wherein information is passed between a group of (layer) and its two 'nearest neighbours (layers)'. Information flow between neurons of the same layer (horizontal connectivity) is also permitted. The layers can be created, edited or deleted by the user through the system provided functions. The parameters are accepted via a template provided to the user, after which the parameters are encoded and saved in the network data structure (fig. 2).

6.2.2 BPNN

A multi-layer feed forward neural network which follows the generalized delta learning rule is provided with modifiable parameters. The user can specify in the BPNN template the number neurons/layer, the number of hidden layers, the bias (threshold) values, the input/output and activation function, layer specific learning rates and momentum factors and other parameters from a list default and optional parameters provided by the system. The user can also update the links between neurons by the link map editor.

6.3 ACTIONS (Primitive)

6.3.1 Create, Edit & Delete Neurons, Synapses

The user shall be given a library of functions for creating and modifying the above mentioned objects. The create_neuron function can be called from within a C program or from the CLIPS interpreter just like defrule. In CLIPS> the user can enter the parameters of a neuron from the template provided. The template will carry default parameters and also provide help on different options available for each parameter. The parameters have to be passed to the create_neuron function if called within a C program. The function will encode the parameters into a special rule and shall also update the network data structure (fig. 2). The function for creating a synapse is called create_synapse and it also is C and CLIPS callable. The synapse information though is only stored in the network data structure. Other functions like edit_neuron and edit_synapse, are basically invoked in the CLIPS interpreter. They let the user modify the values of the neuron/synapse parameters. The delete_neuron functions simply take the neuron and layer numbers and delete the neurons and the links from/to them. The delete_synapse requires the 'from' neuron and layer numbers and the 'to' neuron and layer numbers. The network data structures and clips data structures are updated accordingly.

6.4 ACTIONS (Composite)

6.4.1 Create, Edit & Delete Neurons, Synapses

When a group of neurons or synapses have similar characteristics they can be created, edited and deleted by a single function call. Functions to create, delete and edit a group of neurons and

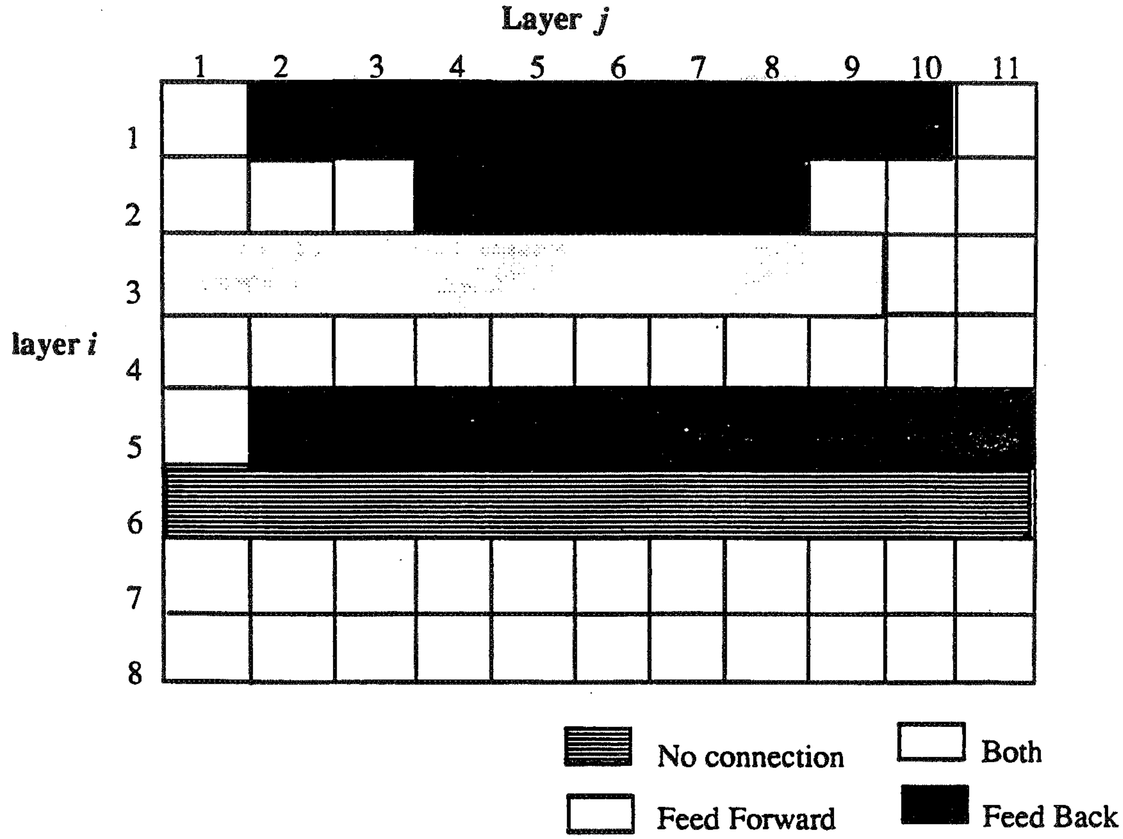
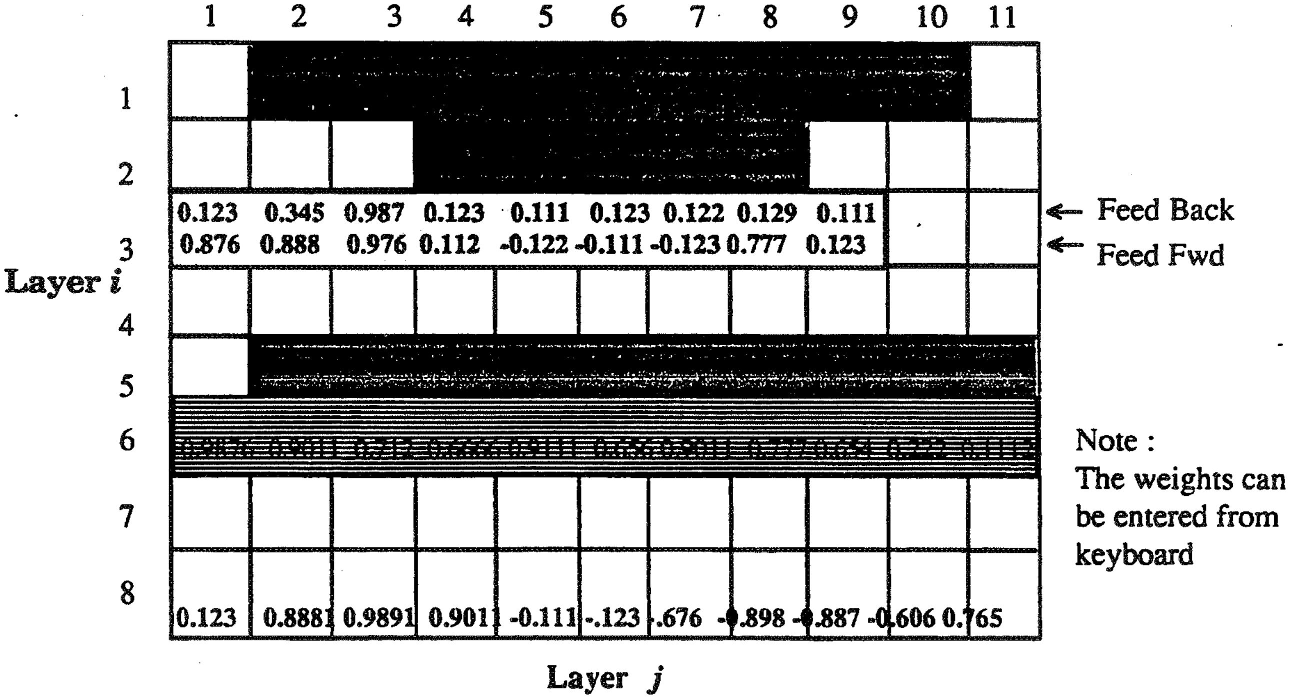


fig 3a. X Windows Link Map Editor : Modifying links, an example.



0.123 0.345 0.987 0.123 0.111 0.123 0.122 0.129 0.111

0.876 0.888 0.976 0.112 -0.122 -0.111 -0.123 0.777 0.123

0.9876 0.8011 0.712 0.6666 0.8111 0.656 0.8011 0.777 0.654 0.222 0.1112

0.123 0.8881 0.9891 0.9011 -0.111 -.123 .676 -0.898 -0.887 -0.606 0.765

Note :
The weights can
be entered from
keyboard

fig 3b. X Windows Weight Editor : Modifying weights, an example.

synapses are provided in the function library. As in the case of primitives, these functions (for the actions) can also be accessed both, from within a C program and from the CLIPS interpreter. The template invoked from the interpreter, however would request additional information from the user apropos the number of neurons, synapses or the layers under consideration, their topological relationship etc. The group is treated as a composite object in the system which stores it as a collection of possibly inter-connected primitive neurons and synapses. These groups can be connected to other groups, though it is a very difficult task to determine the actual neuron to neuron connection as it could be a one-to-one, one-to-many or a many-to-many from one group to another. Also, the connection from one group to another can be a higher level, logical (or abstract) connection. Besides these there can be a neighborhood effect [10] which can be programmed into the group as a rule. The creation and editing of groups of synapses is carried out with an X-Windows link map editor explained next (fig. 3a). The weights of the links can be changed through a similar graphical editor.

6.4. 2 X-Windows link map editor

It is a two dimensional link map where the rows represent the 'from' neurons on a layer and the columns represent the 'to' neurons in another (defaulted to next). It has a mouse interface to switch between four types of connections, namely the feed-forward (black color), feed-back(white color), none at all.(B&W pattern 1), both (B & W pattern 2). After the user has created or modified the links between two layers and has saved them, the map will return a matrix with the values (-1,0,1,2) for feed-back, 'none', feed-forward or 'both' connections between neurons. The user could then either use that matrix to create his/her own link specs in a C program or can let the library function create and modify the data structures. The map has default link connection specifications to create the links automatically.

6.4.3 X-Windows weight editor

It is the same as the link map editor in appearance and functionality with the exception that the user can enter the weights or modify them manually for each type of synapse at the time of creation or at any point during training, even during the execution (fig. 3b).

6.4.4 Create, Edit & Delete Layers

These can be created via direct function calls to create layers, or can be built incremently by first creating the other sub-components of the layers. The layers can be of basically three types input, output and hidden, though feature maps usually have only one layer. The system provides functions to create a standard layer or a group of them. These can be edited as individual layers or a group of (hidden) layers. Once all the neurons on a layer are deleted, the layer automatically collapses. Deleting a layer would result in all connecting synapses being purged too. If a hidden layer is deleted resulting in partition of the network the user shall be prompted with available options which would include destruction of the network and default connections.

6.4.5 Create, Edit & Delete Networks

A user can create, modify and even delete complete neural networks. In this system the user will have the capability of creating his/her own networks by either modifying the system defined neural networks (BPNN, currently, is the only available Neural network) or by customizing one of his/her own.

6.4.6 Ntrain

This function is a set of expert system rules (in CLIPS) which is system defined for feed forward type networks. But the user can write his own training function, if desired. The system defined training function first reads the input pattern and then systematically triggers each layer. To write ones own training function the user will have to write an expert sub system which will then override the previously defined training function. It could be possible to have different training functions if the network consists of different learning algorithms as sub networks. Since there can be more than one network active at any given time, the training functions should be classified by the network number to which they pertain.

6.4.7 Nrun

The neural networks or sub networks can be run from a CLIPS interpreter, a C program, or can be spawned off from CLIPS rules. Since there can be more than one network active at any given time, hence this function also needs to be passed a network identifying number.

6.4.8 Freeze

This function pauses the execution of the network after which the save function can be called to save the snap shot of the system for later analysis.

6.4.9 Show_ready

If the user wants to know, at any given instance, which set of neurons is ready to fire, he can invoke the show_ready function. This function provides a display, either in the form of a list of neurons or as a change of neuron color in a graphical representation of the neural network interconnections. The function can be invoked via a mouse.

6.4.10 Save, Load

A neural network can be saved at any given time in the disk files via the save_nn() and load_nn() functions. The save function saves all the rules in appropriate files and also the data structure associated with that network. The load function reads the same files and builds the neural network representation within the system.

7. CONCLUSIONS

N-CLIPS has turned out to be a very useful tool for solving real life technical problems for which a single knowledge representation or AI technique does not suffice. The building-in of a neural network simulator within CLIPS (the expert system shell) made it easy for the two to communicate with each other, share a common fact (data) base and utilize the other's strengths to overcome its weaknesses (e.g. expert systems brittleness versus the neural networks associative capabilities). The problem of mapping one system into another is a very difficult research topic to be addressed in future extensions of N-CLIPS. As far as the neural network paradigms are concerned, we plan to add all known learning paradigms as stand alone objects. The user-interface, can be enhanced to a complete windowing environment (e.g pop-up menus, mouse selectable options list, graphic templates, etc). The most important enhancement to the system would be the incorporating of semantic networks, searching algorithms, more general connectionist networks, frame based systems, and even petri nets.

8. REFERENCES

[1] Hoskins J. and Jones W. "Back Propagation", BYTE , Oct 87.

[2] D'Autrechy C.L., Reggia, J.A. "MIRRORS/II, Connectionist Simulation", First Annual INNS Meeting, Boston, 1988.

[3] Rumelhart et al, "Parallel Distributed Processing", vol I., 1987, MIT Press,Cambridge,Mass.

[4] CLIPS User's Guide, Artficial Intelligence Section, Johnson Space Center,June 1989.

[5] McGee B., Miller M., Krolak P., and Barr S., "Interfacing PCLIPS into the Factory of the Future", "The First CLIPS Users Group Conference", NASA J.S.C., Houston, Texas, Aug. 1990.

[6] Rabelo L.C. and Alpteking S., "Synergy of Neural Networks and Expert Systems", Proceedings of the Third TIMS/ORSA Conference on FMS, MIT, Cambridge, MA., Aug. 1989.

[7] Bhatnagar H., Krolak P., and McGee B. "A Traffic Controller for Material Handling Systems", submitted to SOAR Conference, Albuquerque, New Mexico. June 26-28, 1990.

[8] Miller R., Korlak P. "PCLIPS : A Distributed Expert System". "The First CLIPS Users Group Conference",NASA J.S.C., Houston, Texas, Aug. 1990.

[9] Coleman J. "Evolutionary Telerobotics: An Approach to the Designing of Telerobotics System", #CPE-NERV-90-5, Center for Productivity Enhancement, University of Lowell, Lowell, Ma. 01854.

[10] Kohenen T. "Self Organizing and Associative Memory. " Springer-Verlag, New York. 1989.

[11] WIlliams R. J. "Inverting Connectionist network mapping by back prop error." Proc. 8th Ann. Conf. Cog. Sci. Soc. 1986.

[12] Gelband P. "Neural Selective Processing and Learning, " Proc. of the First Ann. INNS Meeting, Boston, 1988.

[13] Collins E., Ghosh S. and Scofield C. "An application of a Multiple Neural Network Learning System to Emulation of Mortgage Underwriting Judgements, " Nestor Inc., 1 Richmond Sq, Providence RI 02906.

[14] Reilly D., Scofield C., Elbaum C., and Cooper L.N. "Learning System Architectures composed of Multiple Learning Modules".

[15] Reily D. et al., "An application of a Multiple Neural Network Learning System to Industrial Part Inspection," ISA, 1988, Houston, Texas.

[16] Pineda F.J., "Generalization of Back-Propagation to Recurrent Neural Networks, " Physical Review Letters, Nov. 1987, pp 2229-2232.

[17] Hopfield J., and Tank D.W. "Computing with Neural Circuits," Science 233, 625-633 (1986).

8. REFERENCES *Contd.*

[18] Kirpatrick S., Gelatt C.D., and Vecchi M.P., "Optimization by Simulated Annealing," Science 220, 671-680 (1983).

[19] Grossberg S., Carpenter G.A. "A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine," Chapter 5., Neural Networks and Natural Intelligence, MIT Press, Cambridghe, Massachusetts,1988.

[20] Kosko B. "Bi-Directional Associative Memories, ". IEEE Trans. on systems, Man & Cybernetics, vol 18, pp 49-60, 1988.

APPENDIX

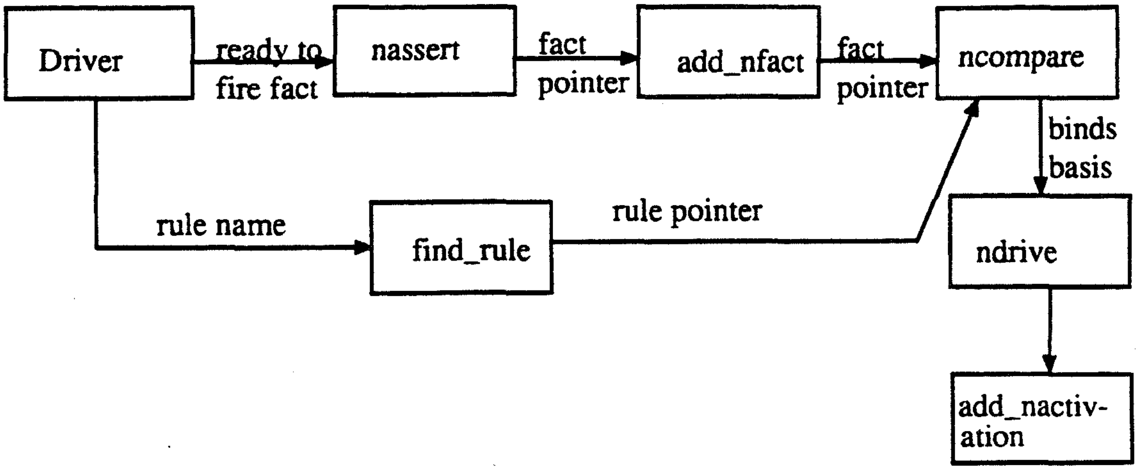


fig. 4 : **A data flow diagram of the changes made to CLIPS for N-CLIPS**

Driver

This function goes through an array of neurons (a layer) and for each neuron that is ready to fire it calls find_rule to set up a global variable pointer which points to the current neuron rule. This is followed by a call to nassert to assert the following fact : (neuron # layer # ready to fire).

Nassert

It calls add_nfact() with the above fact after making sure it has not been asserted already.

Add_nfact

It adds the above fact to the fact list and calls ncompare to filter through the special neuron rule.

Ncompare

It make s the var list (binds), the joins and gets the rule pointer from the global variable and then calls ndrive to drive the fact through the network patterns for that rule .

Ndrive

its task is to put the input parameters in proper data structures and calls add_nactivation to add the rule to the agenda.

An important feature of the above functions has been that only one rule and one fact is in picture. this is done since we know both the fact and the rule which its assertion will trigger. However in case of output neurons other facts are asserted which could trigger an expert system.

Nretract

It retracts the ready to fire fact from the fact list after the neuron has fired.

omit

B10 Session:
Enhancements to CLIPS – Reasoning/Representation

5̶2̶0̶-61 S20-61

N96- 12933

p. 11

Implementation of a Frame-based Representation In CLIPS

358588

Hisham Assal and Leonard Myers[1]

Abstract:

Knowledge representation is one of the major concerns in expert systems. The representation of domain-specific knowledge should agree with the nature of the domain entities and their use in the real world. For example, architectural applications deal with objects and entities such as spaces, walls and windows. A natural way of representing these architectural entities is provided by frames.

This research explores the potential of using the expert system shell CLIPS, developed by NASA, to implement a frame-based representation that can accommodate architectural knowledge. These frames are similar but quit different from the 'template' construct in version 4.3 of CLIPS. Templates support only the grouping of related information and the assignment of default values to template fields. In addition to these features frames provide other capabilities including: definition of classes, inheritance between classes and subclasses, relation of objects of different classes with "has-a", association of methods (demons) of different types (standard and user-defined) to fields (slots), and creation of new fields at run-time.

This frame-based representation is implemented completely in CLIPS. No change to the source code is necessary.

Keywords:
Architecture, Design, Engineering, Expert Systems, Frames, Knowledge-Based System, Knowledge Representation.

Introduction:

Architectural design involves large amounts of information in often diverse fields of knowledge. In order to create a computer-aided design environment for architecture, there should be a uniform representation for architectural entities that is capable of describing all attributes and characteristics of these entities in different contexts of the design activity. Although architectural objects seem to be well defined as the components of a building such as space (a more generic term for room), wall or window, the attributes and characteristics of these objects vary in response to the context of the design activity. For example, in the conceptual design phase, a space may be described in terms of its orientation, adjacency to other spaces or/and access to circulation elements. Whereas in a different level of design, such as daylight analysis, the space may be described in terms of its geometry, window material or/and the amount of daylight it has. The knowledge representation scheme for such an environment should be flexible enough to handle the needs of different activities of design.

The ICADS Model:

The Intelligent Computer-Aided Design System (ICADS) is a project that is being developed in the CAD Research Unit of California Polytechnic State University, San Luis Obispo.

[1] Hisham Assal is a graduate student in the architecture department and Leonard Myers is a professor in the computer science department at the California Polytechnic State University, San Luis Obispo.

C-3

The ICADS model [1] provides on-line access to knowledge pertaining to the kind of design project under consideration; and expert assistance during the iterative analysis, synthesis and evaluation cycle of the design activity. It consists of several components that deal with architectural knowledge on different levels.

The first component is an existing drawing system that produces point/line drawings to represent the architectural solution. In order to allow for analysis or evaluation of the evolving design, there is a geometry interpreter [2] that transforms the point/line representation into architectural objects, such as spaces, walls or windows. The interpreter also formulates the relations that connect the objects (such as, the walls in a space) to provide a meaningful description of the evolving design solution. This information then flows to a control system (the blackboard) [3]. The blackboard receives different information from all the components of the system. It has knowledge about the information needed by every component and it uses this knowledge to efficiently propagate its information. The intelligent design tools (IDTs) are narrowly focused expert systems that perform the analysis and evaluation of the design and send their results back to the blackboard. If there is a conflict in the results of two or more IDTs, the blackboard tries to resolve it in the context of the project as a whole using its own set of rules (conflict resolver). There is also a relational database component that stores prototype information about building types and sites.

One of the inherent problems in this model is the diversity of the formats of information needed in different components. For example, the geometry interpreter produces architectural objects in C structure format, the database queries return tuples in SQL format and the IDTs use CLIPS facts. In fact this diversity is common in systems where a variety of databases are needed [4]. There is a need for a common representation to make it possible for all components to communicate with each other.

The common representation of information designed for the ICADS system is the frame-based scheme described in this paper.

CLIPS Knowledge Representation:

CLIPS is a forward chaining rule-based expert system shell, developed by NASA [5]. It has three major components:

- fact-list which is the working memory of facts.
- knowledge base which is the set of rules and initial facts.
- inference engine that controls the overall execution.

Information in a CLIPS expert system is represented in the form of facts. The structure of facts is quite simple. A fact is merely a list of one or more fields which may be one of three types: a word, a string or a number. A word is any field that does not start with a number or a special character; a string is any character or set of characters between quotes; and a number is always a floating point number. Fields cannot be lists themselves. That means that nested lists are not allowed in this environment. There is no restriction whatsoever on the field values that can be in a fact or the order of fields in a fact. In addition to the simple fact structure, there is a 'template' structure that was introduced in the CLIPS version 4.30. The 'template' provides two features: field identification and default values. The structure of a template has two components: a label and a list of name-value pairs. The use of field names in templates permits the fields to be identified regardless of the order in which they are written. It also makes it possible to provide default values for the fields declared in a template.

Templates enhance the representational power of facts in CLIPS. Further enhancement can be provided by a more general frame-based representation scheme.

The Frames Paradigm:

Frames provide a structured mechanism of representing different types of knowledge [6]. They have some powerful features that help to capture human knowledge in such a way as to facilitate both conceptual level and programming level uses of the knowledge. A frame can be viewed as a collection of information about an object. It may represent a physical object, such as window, or a conceptual object, such as climate. A frame may represent a class of objects by describing its general characteristics and relations to other objects. It may also represent an instance by specifying its class and specific characteristics. Classes may be arranged into taxonomies; i.e. a frame may represent a subclass which is a specialization of a class. The class information is available to any instance of the same class or of any of its subclasses through inheritance.

The structure of frames consists of slots that represent different types of information [7]. The content of a slot may be a value of any type (number, string, ... etc.); a restriction upon another slot's value (range, type, ... etc.); a demon, which is a method of performing a special task; a relation to another frame; or any other kind of information. Inheritance can be applied to any type of slot, or it can be suppressed for a particular instance. Different types of relations may be defined among frames, such as is-a, has-a, a-kind-of, ... etc.

Combining frame-based representation with pattern matching techniques adds power to frames in terms of reasoning facilities. Reasoning with frames involves several levels: class level, instance level and slot level. For example, operations may be performed on a particular slot in all instances of a class; a certain type of relation may be identified in all classes; and restrictions may be imposed on a type of value (e.g. boolean: true or false).

Creating Frames:

The implementation of frames in the CLIPS environment comprises three parts: representation, generation and manipulation.

- Representation is the form and collection of facts that compose a frame.
- Generation is the phase or module that creates new frames and/or slots and relations.
- Manipulation is the module that performs operations on frames, slots or relations, such as add, delete or modify the contents of a frame.

It should be noted that the manipulation rules are different from the application rules that use the information stored in frames without directly changing any of it. The basic purpose of the manipulation module is to provide a mechanism for dealing with frames so that the user can set up the conditions or restrictions or specify actions to be taken upon additions, deletions or changes in frame contents.

Representation of Frames:

A frame can hold either a class or an instance. If a frame holds a class, then the information in this frame will describe the basic characteristics of this class such as default values, demons as methods of obtaining values or performing particular tasks, names for the value slots in this class (without actual values), and relations between this class and other classes. It may also include any other information that the user wishes to have such as: restrictions on slot values, facets for describing how to deal with a particular piece of information, ... etc. On the other hand, if a frame holds an instance, then the information in this frame will be the actual values for the value slots and the actual instance identifiers for the relations. Through inheritance, all the class information will be available to any frame of this class or any of its subclasses.

A frame is represented by a set of facts that have one or more common fields to connect them together. Each fact has a keyword in the first field to indicate the type of information it represents. The keywords are: CLASS, DEFAULT, DEMON, FRAME, RELATION, and VALUE. The second field has the class name which is used to connect all instances of this class, relate the class to its superclass, or establish a relation with another class. In the instance frames, there is a field for the frame identifier which is used to connect all the facts representing a particular frame instance. In addition to these basic fields, every fact contains different number of other fields to describe the piece of information it holds.

Definition of Classes:

The first step in creating frames is the definition of classes that will be used in the application. A class definition has the following components:

- A class header that declares the class name and its superclass (if any). If the class does not have a superclass (i.e. it is the uppermost level class), the class name is repeated in place of the superclass. The class header is a fact of the form:

(CLASS <class> <superclass>)
where CLASS is a keyword,
<class> is the class identifier and
<superclass> is its superclass identifier.

Since this class header is the only place that has information about the class/superclass relationship, the names of all classes and superclasses must be unique.

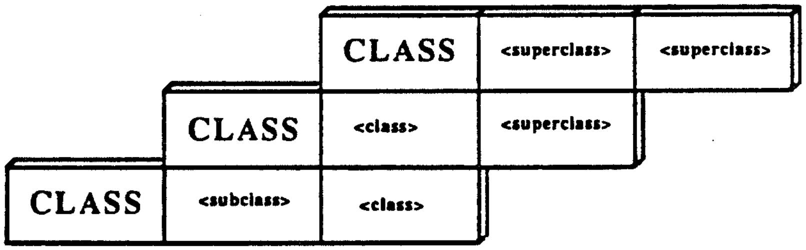


Fig. 1. Class Hierarchy.

- Default slots for all the default values in this class. A default value can be accessed by any instance of its class through inheritance. The default slot is a fact of the form:

(DEFAULT <class> <attribute> <value>)
where DEFAULT is a keyword,
<class> is the class identifier,
<attribute> is the slot name and
<value> is the default value.

- Demon slots that declare all the demons of this class as methods of obtaining values. A demon is represented by a fact of the form:

(DEMON <class> <attribute> <type>)
where DEMON is a keyword,
<class> is the class identifier,
<attribute> is the slot name that should have this value and
<type> is the type of the demon that controls its firing.

Along with this fact, there should be a set of one or more rules that actually describe the method of obtaining the value. Users can define the type of demon and set the conditions that control its firing. For example, if it is of type 'if-needed', it will fire only once when there is no current value for this attribute. However, it will not fire again until this value has been deleted. If it is of type 'if-changed', it will fire every time the value of this attribute has been changed. Since demons belong to classes, a fact must be asserted, when firing the demon, to indicate the instance of the class that will receive the result of the demon.This fact has the form:

(DEMON <class> <attribute> <instance> <type>)
where DEMON is a keyword,
<class> is the class identifier,
<attribute> is the slot name that should have this value,
<instance> is the frame identifier of the instance, and
<type> is the type of the demon that controls its firing.

- Value slots that declare the basic attributes of the class. These slots do not have values since the actual values will be in the instance frames. A value slot in the class definition is a fact of the form:

(VALUE <class> <attribute>)
where VALUE is a keyword,
<class> is the class identifier and
<attribute> is the slot name.

- Relation slots that describe the relation between this class and other classes. A relation in this implementation is a 'has-a' relation. As in value slots, relation slots do not have the actual instances of the classes. A relation slot is a fact of the form:

(RELATION <class> <other class>)
where RELATION is a keyword,
<class> is the class identifier and
<other class> is the identifier of the related class

The interpretation of this type of fact should be: every instance of <class> has an instance of <other class>. That means that the whole frame of <other class> is a part of the frame of <class>. However, a relation does not imply any inheritance. It is, rather, a way of defining the relationship between classes that are not derived from one another.

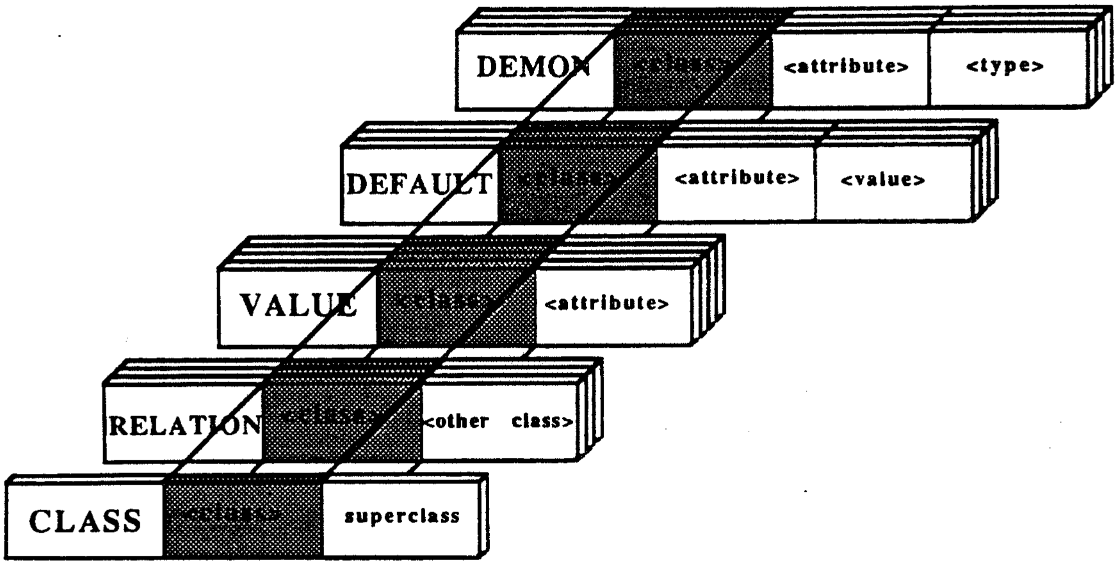


Fig. 2. Class Definition.

Definition of Instances:
An instance of a class is defined as follows:

- a frame is defined by a FRAME header which is a fact of the form
  (FRAME <class> <instance>)
  where FRAME is a keyword that should be in the first field,
  <class> is the name of the class of this frame and
  <instance> is the frame identifier.

The FRAME header may not be necessary in accessing the slot value in a frame, but it is useful in performing operations on the whole frame, such as displaying frame information, deleting a frame or relating a frame to another frame.

- a slot value is defined by a VALUE slot of the form:
  (VALUE <class> <attribute> <instance> <value>)
  where VALUE is a keyword,
  <class> and <instance> are the same as in the frame header,
  <attribute> is the slot name or attribute and
  <value> is the actual value of this slot.

The <value> field may be a single-field or a multi-field value depending on the nature of this slot. If the attribute in this slot has the nature of a list, such as the coordinates of a point (x,y), then a multi-field value should be used in the slot fact.

- a relation is defined by a RELATION slot of the form:
  (RELATION <class1> <class2> <instance1> <instance2>)
  where RELATION is a keyword,
  <class1> and <class2> are two class identifiers,
  <instance1> is an instance of class1 and
  <instance2> is an instance of class2.

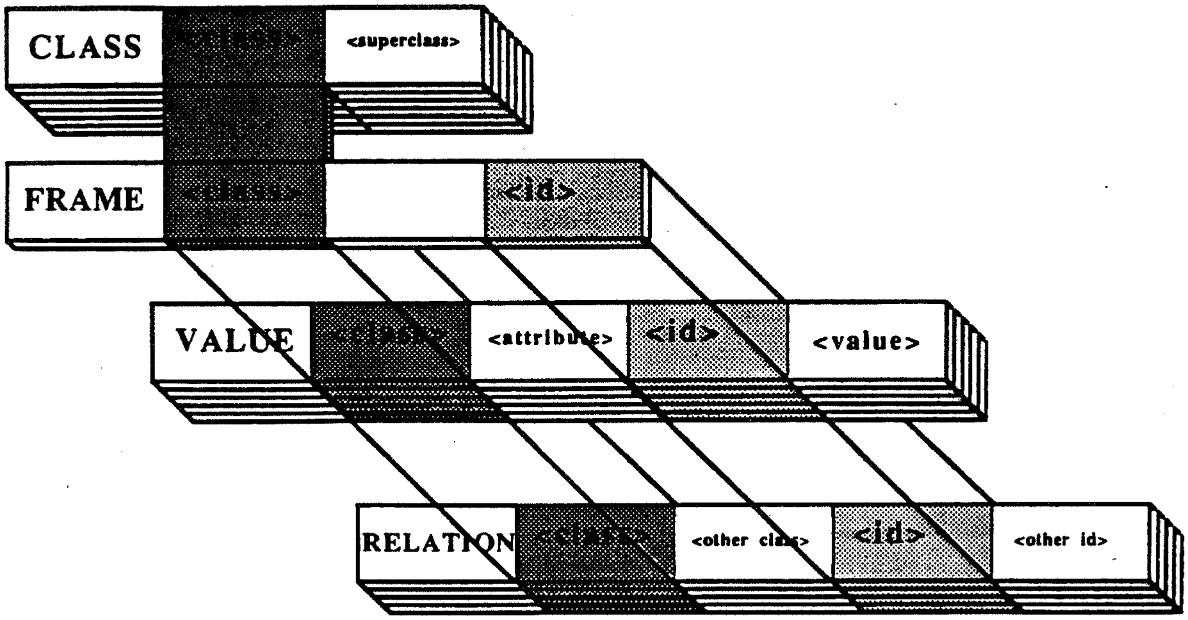


Fig. 3. Class-Instance Relation.

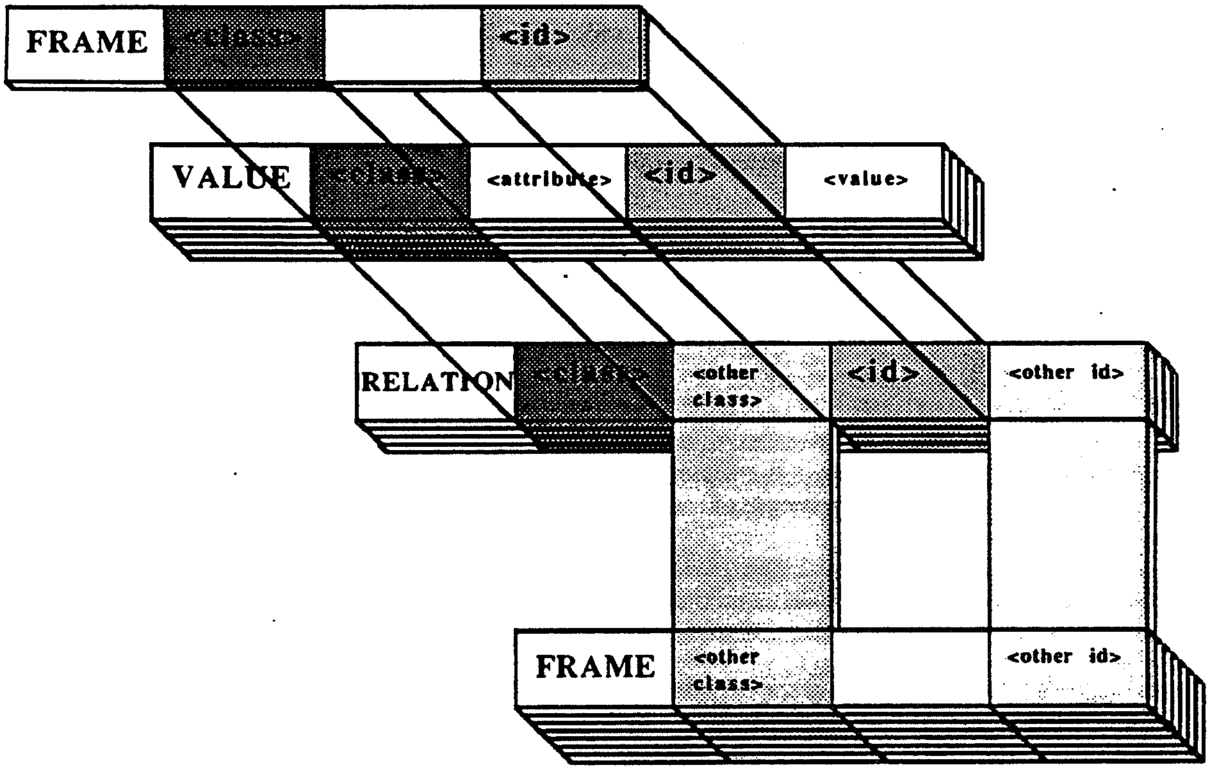


Fig. 4. Instance-Instance Relation (has-a).

Generation of Frames:

The definitions of the class frames are kept in a separate file to allow them to be reused in other programs. This file typically contains all the facts that describe each class and all the rules for the demons. The generation of instances for an application can be either static or dynamic. Static generation involves the creation of fact files that contain all the instances that are known prior to execution. Dynamic generation is usually achieved by having a module that is responsible for creating frames, slots or relations according to the state of the system and the conditions set by the user.

In the ICADS model, there are two modules that create frames dynamically: the Geometry Interpreter (GI) and the Attribute Loader (AL). The GI is responsible for creating frames that contain the geometry of the evolving solution drawn by the user in the CAD system . The AL is responsible for creating frames that contain the non-geometric attributes of the building being designed from a prototype database and all the relations that relate these frames to the geometric frames of the GI. The GI is a C module that was added to a modified version of CLIPS, while AL is a CLIPS module that has access to the SQL relational database.

Manipulation of Frames:

Frames are controlled by a module that takes care of performing the actions, enforcing the restrictions and checking the facets while manipulating the frames. The main three actions to be performed on frames are: ADD, DELETE and MODIFY. Each of these actions can be applied to FRAME, VALUE or RELATION slots (with the exception of MODIFY RELATION). If, for example, there is a restriction on a slot value to be of a certain type or within a certain range, then this module will check this restriction and enforce it.

Inheritance in Frames:

There is a set of rules that perform the inheritance operation. These rules are kept in a separate file that should be loaded with any application that uses inheritance. The inheritance rules have a priority (salience) of 10000 to allow the inheritance to take place as soon as it is invoked. The rules of the application itself should not have a higher priority.

Class-subclass inheritance:

Inheritance must be explicitly requested. This means that there should be a rule to issue a request for inheritance when the absence of a value is detected. The request is a fact that activates the inheritance rules. This fact has the form:

```
(INHERIT <class> <attribute> <instance>)
where  INHERIT is a keyword,
       <class> is the class name of the requesting frame,
       <attribute> is the slot name to be inherited, and
       <instance> is the requesting frame id.
```

When a request for inheritance is issued, the class frame of the requester is searched first for the requested slot. If it is found, its value is inherited; i.e. a VALUE slot is created for the requester with the value field. If the slot is not found and the class has a superclass, a request for inheritance is issued for the same slot in the superclass. This process continues until a slot with the required name is found or no other classes are to be searched. The slot to be inherited need not be in a VALUE slot. It may also be a DEFAULT slot or a DEMON slot. When a DEMON slot is inherited, the demon fires and creates a value. This value is then inherited in a VALUE slot.

Other Types of Inheritance:

Instances may implicitly inherit slots from other instances that are not in the same class hierarchy. In this case, DEMONs are used instead of the inheritance rules. Since DEMONs describe ways of obtaining values for specific slots, they can simply get the value of any other slot in the same instance frame or in any other frame. For example, if a wall instance has a slot for 'height', a space instance may get the value of this slot for its 'ceiling-height' slot using a DEMON in its class definition. This DEMON must have knowledge about the relationships between wall frames and space frames.

Reasoning With Frames:

The arrangement of frames as sets of separate facts connected by common fields makes it possible for different levels of reasoning to take place using the powerful pattern matching of CLIPS. Levels of reasoning involve:

* Class reasoning.
* Instance reasoning.
* Slot reasoning.
* Relation reasoning.

- Class reasoning: Using the class field in a frame, operations may be performed on all instances of this class. For example, to display the names of all spaces, a rule as the following may be used:

```
(defrule display-space-names
        (VALUE space name ?id ?value)
=>
        (fprintout t "Space " ?id " has the name " ?value crlf)
)
```

-Instance reasoning: Using both the class and the identifier fields, operations may be performed on all slots of a particular instance. For example, to display all the information of a particular wall instance 'wall-1', a rule as the following may be used:

```
(defrule display-wall-slots
        (VALUE wall ?attribute wall-1 $?value)
=>
        (fprintout t "The attribute " ?attribute " has the value(s) " $?value crlf)
)
```

- Slot reasoning: Operations may be performed on slots that have specific characteristics such as the name, the value or the number of values regardless of what frame they belong to. For example, to display the height of all the objects that have a 'height' slot, a rule as the following may be used:

```
(defrule display-heights
        (VALUE ? height ? ?value)
=>
        (fprintout t "The height of " ?object " " ?id " is " ?value " ft." crlf)
)
```

- Relation reasoning: The information in a frame that is related to another frame can be accessed by using the <other class> and <other id> fields in the relation slot. For example, to display the length of all the walls in a space instance "space-1", a rule as the following may be used:

```
(defrule display-wall-length
        (FRAME space space-1)
        (RELATION space wall space-1 ?wall-id)
        (VALUE wall length ?wall-id ?value)
=>
        (fprintout t "Wall " ?wall-id " has length " ?value crlf)
)
```

Conclusion:

Rules are useful in representing knowledge about situations in the domain world and actions to be taken in each situation. In addition, there is also a need to represent the entities of the domain world, relationships among these entities, and operations that could be performed on them. These entities are referred to as objects. When dealing with a problem that uses objects, it is appropriate to use frames. This frame-based representation takes advantage of the pattern matching technique of CLIPS to provide a flexible yet powerful frame environment.

Flexibility is achieved by arranging the frame as a set of facts. This provides the ability to add a new slot at run time, deal with one slot in a frame without having to retrieve the whole frame, or remove a slot or modify its value without affecting the rest of the frame.

The power of this representation is attributed to the pattern matching, which allows different kinds of associations, such as class-subclass, class-instance, or class-class relations. Class-subclass relations are necessary in order to provide an effective taxonomy of the architectural entities in the ICADS system. Class-instance relations are used to effect the inheritance functions that make it possible to efficiently store the large numbers of architectural details necessary in the ICADS project. The class-class relation 'has-a' is used to synthesize, or define an object by specifying its components or features. The use of these associations in the prototype ICADS system has proved to be paramount to providing a robust, efficient representation of the architectural objects that naturally reflects the way the objects are perceived by human designers/architects.

Pattern matching also provides different levels of reasoning, such as, class reasoning, slot reasoning, or relation reasoning. Demons represent methods of performing operations that are specific to a class of frames. The frame manipulation module offers a means of control for the user (expert system developer) to impose some restrictions or to perform some tasks upon adding, deleting, or modifying slots.

REFERENCES:

1. Pohl, J., Chapman, A., Cotton J. and Myers L. *ICADS: Working Model Version 1, technical report, CADRU-03-89*, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, CA, 1989.

2. Taylor, J. and Pohl, J. *A Geometry Interpreter for Extracting Architectural Objects from the Point/Line Schema of a CAD database.* Proc. Intersymp-90, Baden-Baden, West-Germany, August 6-12, 1990.

3. Taylor, J. *A Framework for Multiple Cooperating Agents in an Intelligent Computer-Aided Design Environment*. (Master Thesis). School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA. 1990.

4. Howard, H.C. and Rehak, D.R. *KADBASE: Interfacing Expert Systems with Databases*. IEEE Expert, Fall 1989.

5. NASA. *CLIPS Architecture Manual (Version 4.3)*. Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, May, 1989.

6. Minsky, M. L. *A Framework for Representing Knowledge*. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975

7. Fikes, R. and Kehler, T. *The Role of Frame-Based Representation in Reasoning*. Communications of the ACM, vol. 28, No. 9, September 1985.

BB_CLIPS: Blackboard Extensions to CLIPS

Robert A. Orchard, Aurora C. Diaz

Lab for Intelligent Systems, Division of Electrical Engineering
National Research Council of Canada
Ottawa, CANADA K1A 0R6

NRCC Publication No. 31505

aurora@ai.dee.nrc.ca, bob@ai.dee.nrc.ca

Abstract

This paper describes a set of extensions made to CLIPS version 4.3 [1] that provide capabilities similar to the blackboard control architecture described by Hayes-Roth [2]. There are three types of additions made to the CLIPS shell. The first extends the syntax to allow the specification of blackboard locations for CLIPS facts. The second implements changes in CLIPS rules and the agenda manager that provide some of the powerful features of the blackboard control architecture. These additions provide dynamic prioritization of rules on the agenda allowing control strategies to be implemented that respond to the changing goals of the system. The final category of changes support the needs of continuous systems, including the ability for CLIPS to continue execution with an empty agenda.

Keywords: CLIPS, blackboard, dynamic control

1. Introduction

This paper describes changes that add a blackboard control architecture to CLIPS version 4.3 and enable the operation of continuous systems. This extended version of CLIPS is called BB_CLIPS.

One class of modifications implements changes in the syntax of CLIPS, allowing the facts base to be partitioned into appropriate user defined blackboards and levels within a blackboard. A second class implements changes in CLIPS rules and the agenda manager to incorporate some of the powerful features of the blackboard control architecture. These include modifications that allow for (1) a more detailed description of the features of a rule in its declare section, (2) the use of special rules to manage problem-solving control and strategy decisions, and (3) the use of a combining function to bring together the current control and strategy decisions with the features of the rules to calculate the current priority of each rule on the agenda. A third class of modification implements changes in the functionality of CLIPS to facilitate the operation of continuous systems. These enhancements include (1) the extension of the *run* command to receive other parameters that allow BB_CLIPS to continue executing even with an empty agenda, (2) the addition of *runstart* and *runstop* functions (very much like the exec functions of CLIPS) which are invoked whenever the run command is executed or terminated, and (3) the addition of a function that, when executed, changes the recency control strategy from most-recent to least-recent.

The use of the above modifications are optional and existing CLIPS programs will execute correctly with no changes. In addition, it should be noted that these modifications add very little runtime overhead (in some cases it is faster than the unmodified CLIPS).

Section 2 describes changes made to CLIPS to implement the blackboard control architecture and discusses the first two types of modification. Section 3 describes the changes that enable the operation of continuous systems. And finally some discussion of the use and future of BB_CLIPS is presented in section 4.

2. Blackboard Architecture

A blackboard-based system consists of three basic components:

1. The knowledge sources which are separate and independent modules of knowledge needed to solve the problem.

2. The global blackboard structure that contains the problem-solving state data. The knowledge sources post changes to the blackboard that incrementally build a solution to the problem. Communication and interaction among the knowledge sources are through this blackboard.

3. The scheduler that supervises knowledge source execution and blackboard access.

In BB_CLIPS each CLIPS rule serves as a knowledge source, its facts base as the blackboard, and its agenda manager as the scheduler.[1]

2.1. Specifying Blackboard Locations

A blackboard-based system is usually organized into one or more blackboards that are partitioned into various levels according to the needs of the application (see Figure 1). The syntax of the facts and patterns in CLIPS has been modified to allow the system designer to clearly specify the two components of the blackboard data; the blackboard entry or relation which is the information content of the data and the blackboard specification which indicates the location within the blackboard structure where this information is stored.

The following (1) illustrates the syntax of a fact that is associated with a particular blackboard and placed at a specified level within that blackboard.

(status PUMP1 ON) $in (component_bb pump 100) (1)

The relation *status* contains the information that *PUMP1* is *ON* and this information is found in the *component_bb* blackboard with value *pump* in the component type level and value *100* in the time level.

[1] In this document the term *rule* and *agenda manager* are used when talking about BB_CLIPS and *knowledge source* and *scheduler* when talking about the blackboard architecture in general.

In general a blackboard specification has the following syntax:

```
$in (bb_name level1 ... leveln)
```

where *$in* is the delimiter separating the relation information and the blackboard specification. The information between the parentheses identifies the name of the blackboard and any sublevels within it.

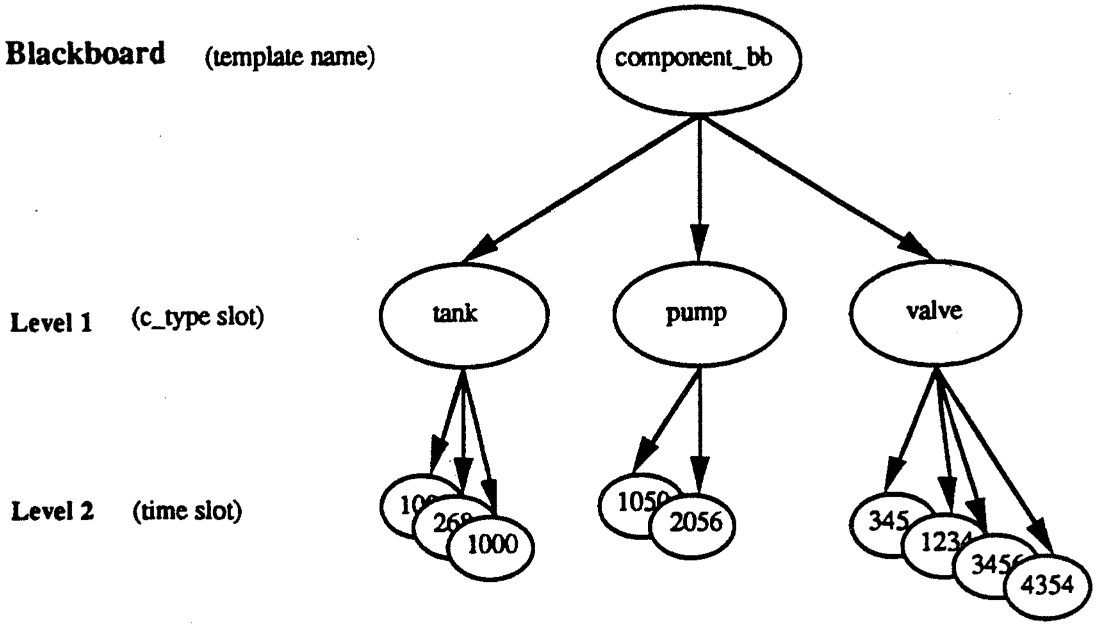


Figure 1 – Blackboard Structure

With CLIPS version 4.3, templates may be used to describe a relation more fully. Similarly, in BB_CLIPS 4.3, a template can be used to describe the relation and another to describe the blackboard specification. Consider the following template definitions:

```
(deftemplate status (field c_instance (type WORD)) (field has_value (type WORD)))
(deftemplate component_bb (field c_type (type WORD) (field time (type NUMBER)))
```

Fact (1) above may be rewritten, given the above template definitions, as:

```
(status  (c_instance PUMP1)  (has_value ON))
        $in (component_bb (c_type pump) (time 100))                    (2)
```

No distinction is made between templates used to describe relations and those used to describe blackboard specifications. Any operation that is valid for a relation template is valid for a blackboard specification template. Thus, to change the blackboard specification and one of the relation slots for fact (2) above, the following modify command could be used:

```
(modify ?fact_id (has_value OFF) $in (time 200))
```

This modify command retracts the old fact (status PUMP1 ON) $in (component_bb pump 100) and asserts the new fact (status PUMP1 OFF) $in (component_bb pump 200). The fact must have been previously bound to ?fact_id.

For a single fact, template and non-template relations and blackboard specifications may be mixed[2]. The modify command may be used only for templates, therefore, given a fact that has a non-template relation and a template blackboard specification, only the slot values in the blackboard specification may be *modified.*

2.2. Blackboard Control Architecture Features

The blackboard architecture has been implemented in many different ways. One such implementation, developed at the Knowledge Systems Lab at Stanford University, allows the system to reason about and explicitly represent control decisions on knowledge source firing. It is called the blackboard control architecture [2]. This allows for the unification of goal-directed and data-directed control which forms the relationship between actions and results that is needed in order to make intelligent control decisions [3].

The blackboard control architecture separates knowledge sources into two types. The first is used to solve the domain problem and knowledge sources of this type are called domain knowledge sources. The second deals with solving the control problem; that is, to determine which of the potential actions (rule firings) to perform at each point of the problem-solving cycle. These are called control knowledge sources and they embody the strategy and control knowledge or meta-level knowledge of the system. There are also two types of blackboards. One type are called domain blackboards and contain decisions made when solving the domain problem. The other hold decisions made when solving the control problem and are referred to as control blackboards. Also there is a single scheduler that supervises knowledge source execution and blackboard access for both types of knowledge source and blackboard. The scheduler decides which knowledge source to execute and considers (1) the features of the knowledge sources which have been triggered and are currently on the agenda, (2) the decisions that have been posted on the control blackboard(s), and (3) some combining or integration function to determine current priorities for the knowledge sources on the agenda.

In BB_CLIPS there is no difference in the syntax that distinguishes domain and control rules. Also, the organization of both the control and domain blackboards are left to the system designer. The next subsections describe additions made to CLIPS that allow flexible and dynamic prioritization of rules.

2.2.1. Declare Section

Standard CLIPS allows a static *salience* to be specified in the declare section of a rule definition. This is used to order the rules found on the agenda. In BB_CLIPS, the declare section is enhanced to allow a more detailed specification of the *features* of a rule. Feature

[2] Each template may have only one multifield slot. For a fact with a template relation and a template blackboard specification, the template relation may have one multifield slot and the template blackboard specification may also have one multifield slot.

values may be integers, elements of a predefined set (e.g. low, alarm), or a blackboard specification (e.g. $in (interface_bb operator_cmd)).

Consider the following declare section of a rule:

```
(declare                                                        (3)
        (salience 100)
        (problem alarm)
        (efficiency low)
        (importance 5)
        (focus $in (interface_bb operator_cmd))
)
```

This declares that the rule belongs to the set of rules dealing with the alarm problem and that it has a salience of 100, a low efficiency, an importance of 5 and will produce a blackboard entry in the operator_cmd level of the interface_bb blackboard. This interpretation is determined by the system designer, as are the features that are needed for the problem at hand.

The declare section of each rule is validated when the rule is loaded. The rule compiler will check the syntax of a feature and ensure that the values for each feature are allowable. Therefore, each feature must be identified by the system designer in a file containing declaration definitions for each feature that is to be allowed in the rules. This file is compiled and linked with BB_CLIPS providing the predetermined set of features[3]. The system designer specifies the feature names and the valid values that these features may take. For a feature of type *integer* this means defining a valid range; for a feature of type *set* this means enumerating the valid set members; and for a feature of type *blackboard specification*, no validation is done because the blackboard organization is determined dynamically. Below is part of such a feature declaration.

```
struct declare_template valid_declarations [] =
  {
        {"salience", SALIENCE_FEATURE, &salience_range ,NULL},
        {"reliability",INTEGER_FEATURE, &reliability_range, NULL},
        {"efficiency", SET_FEATURE, NULL, &efficiency_set},
        {"focus", BB_SPEC_FEATURE, NULL, NULL},
        {"problem", SET_FEATURE, NULL, &problem_set},
        {"prob_type", SET_FEATURE, NULL, &prob_type_set},
        {"sub_type", SET_FEATURE, NULL, &sub_type_set},
  };

struct set_descriptor efficiency_set =
        { 3, efficiency_set_mem};

charptr efficiency_set_mem[] =
        {"low", "medium", "high"};
```

[3] This is similar to the method for adding user defined functions to CLIPS. The authors acknowledge that it would have been more flexible to allow the features to be dynamically created and loaded when BB_CLIPS starts up and this could be considered at some future date. Similarly the combining function used to determine dynamic priorities would also have to be attached to BB_CLIPS at runtime (this is more difficult).

2.2.2. Control and Intercept Rules

As stated earlier, there are separate knowledge sources that post control or metalevel decisions on the control blackboard. These decisions are taken into account when the scheduler is deciding which knowledge source to invoke, thereby providing dynamic prioritization of knowledge sources. For example, a decision on the control blackboard might specify that knowledge sources with efficiency of low or medium be given a certain weight. The scheduler when calculating priorities, will use this weighting factor attached to the efficiency feature for any knowledge sources that are currently triggered and for future knowledge sources as they become triggered. Later, should this control decision be retracted, the priorities of any triggered knowledge sources with the efficiency feature are recalculated immediately and future knowledge source priorities will also be adjusted.

In BB_CLIPS decisions posted on the control blackboard are asserted in much the same way as decisions posted on the other non-control blackboards. In addition, however, some *intercept rules* need to be included which when fired invoke procedures to store these decisions in a separate data structure which is available to the agenda manager. The assertion of the control decision:

```
(efficiency 100 == low medium) $in (control_bb policy)                    (4)
```

might, for example, cause the following intercept rule to be instantiated and added to the agenda.

```
(defrule intercept_cf_set                                                 (5)
        (declare (salience MAX_SALIENCE))
  ?f <- (?feature_name ?wt ?func $?val) $in (control_bb policy)
=>
        (set_cf_set ?f ?feature_name ?wt ?func $?val)
)
```

The intercept rule (5) above calls the external function set_cf_set[4] that ensures that the function (?func) is valid for the set type feature (?feature_name) and that the values given for the feature ($?val) are valid for the set feature. If all checks are passed, the weighting factor for the feature (?wt) is stored in a data structure used by the agenda manager when calculating the priorities of the rules on the agenda.

Intercept rules usually have a maximum salience so that they are executed immediately. Once the intercept rule illustrated in (5) is executed, all rules in the current and succeeding agendas that declare either a low or medium efficiency are given priorities that take into account the control decision made in (4) – until this control decision is retracted. The next two rules are examples of intercept rules for the integer and blackboard specification features.

```
(defrule intercept_cf_int
        (declare (problem intercept))
  ?f <- (?feature_name ?wt ?func $?val) $in (control_bb policy)
=>
        (set_cf_int ?f ?feature_name ?wt ?func $?val)
)
```

[4] There are predefined external functions to handle integer, set, and blackboard specification features. These are set_cf_int, set_cf_set, and set_cf_BBspec respectively.

```
(defrule intercept_cf_BBspec
        (declare (problem intercept))
  ?f <- (?wt $?BBspec) $in (control_bb focus ?type)
=>
        (set_cf_BBspec ?f ?type ?wt $?BBspec)
)
```

There are predefined functions associated with integer and set features. For integer type features these are <, <=, >, >=, ==, !=, IN_RANGE, and NOT_IN_RANGE. For set type features these are == and !=. The operation of these may be changed or new functions may be added by modifying appropriate files. Only functions previously defined as valid for the different feature types may be used in facts asserted by the control rules to reason about the features. For instance, if a control rule concludes that rules with low or medium efficiency should have a weighting factor of 100, given the *current state of the problem*, then it could assert a fact of the form illustrated in (3). This fact makes use of the efficiency set feature and the == (equality) function which has been predefined for set type features.

2.2.3. Combining Function

The agenda manager in BB_CLIPS uses the feature declarations of a rule and control decisions plus some predefined combining function to determine a priority for a rule. The features of a rule are set when the rule is loaded and can be changed only by modifying the rule definition and then reloading it. Control decisions are posted on the control blackboards and are trapped by user-defined intercept rules (as explained in the previous section). Upon execution of one of the functions set_cf_int, set_cf_set, or set_cf_BBspec, the priorities of the rules currently on the agenda are recalculated to incorporate the new control decision.

A predefined function is used to combine the control decisions and the features of the rules on the agenda to determine the priority of a rule. Consider the following control decisions:

```
(problem 500 == alarm) $in (control_bb policy)
(200 interface_bb operator_cmd) $in (control_bb focus strategic)
(efficiency 100 == low medium) $in (control_bb policy)
(importance 10 IN_RANGE 0 5) $in (control_bb policy)
(importance 20 IN_RANGE 6 10) $in (control_bb policy)
```

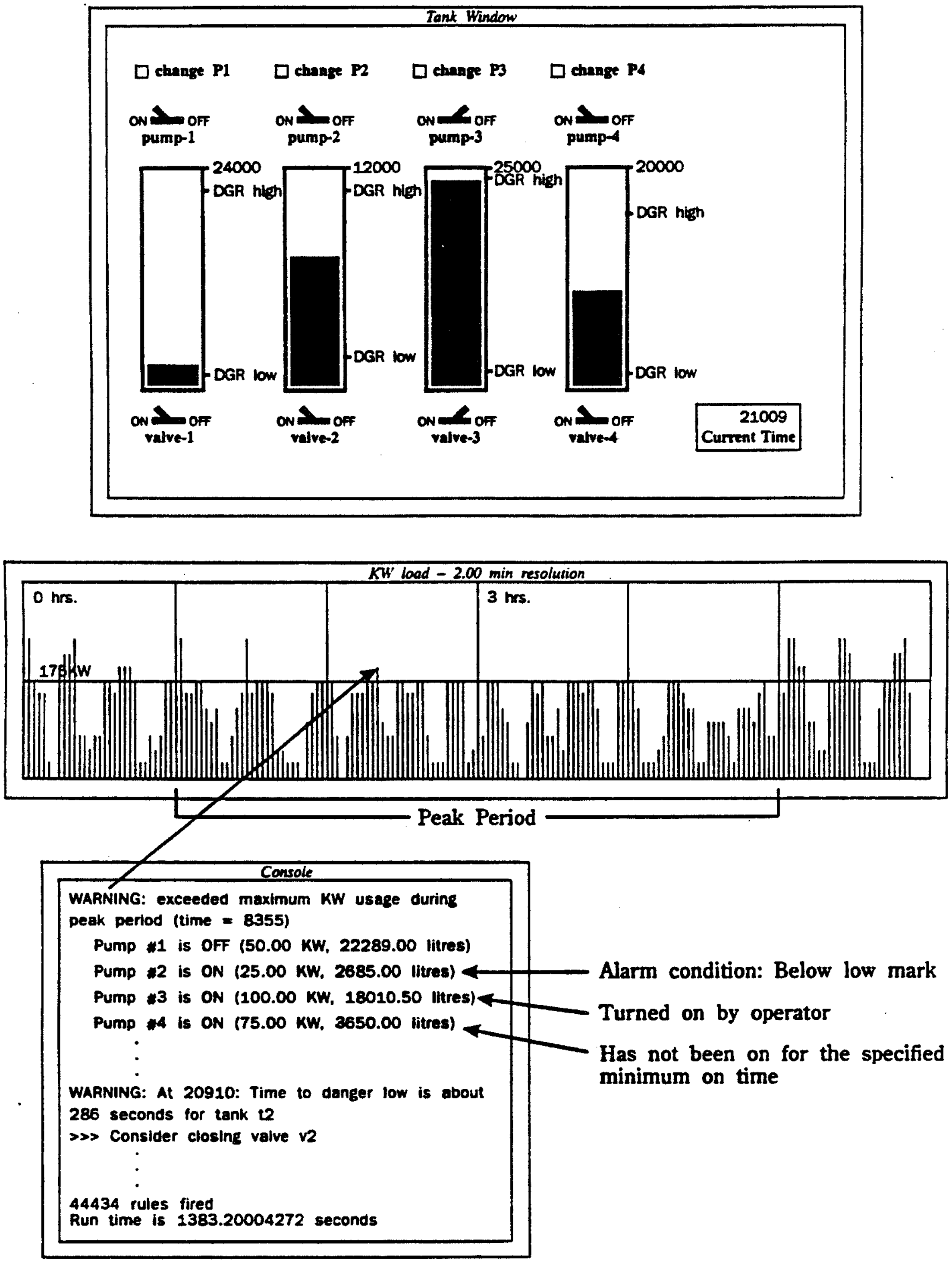
If the combining function adds the weights assigned to the set and blackboard specification features and adds the product of the value of the integer features and the weight assigned to these, then a rule in the agenda with the declaration shown in (3) will have a priority of:

$$500 + 100 + 100 + (5 * 10) + 200 = 950.$$

The above combining function is defined in a file that is provided and may be modified by the system designer as necessary to fit the problem at hand.

3. Continuous Operation and Other Additions

This section describes further extensions made to CLIPS to address the needs of continuously operating systems and to provide other features that were found to be useful.



```
WARNING: exceeded maximum KW usage during
peak period (time = 8355)
   Pump #1 is OFF (50.00 KW, 22289.00 litres)
   Pump #2 is ON (25.00 KW, 2685.00 litres)
   Pump #3 is ON (100.00 KW, 18010.50 litres)
   Pump #4 is ON (75.00 KW, 3650.00 litres)
       .
       .
       .
WARNING: At 20910: Time to danger low is about
286 seconds for tank t2
>>> Consider closing valve v2
       .
       .
       .
44434 rules fired
Run time is 1383.20004272 seconds
```

Alarm condition: Below low mark

Turned on by operator

Has not been on for the specified minimum on time

Figure 2 — Test Program

3.1 The Run Command

Normally, CLIPS terminates when the agenda is empty. For real-time systems (or any continuously operating system) there is need for a mechanism that allows the inference engine to idle, waiting for events to occur without executing a dummy idle rule. In BB_CLIPS, the run command was extended to receive any of the following parameters:

A positive integer n.
BB_CLIPS will run until n rules have executed or until the agenda is empty, whichever comes first. e.g. (run 10)

-1.
BB_CLIPS runs until the agenda is empty. e.g. (run -1)

-2.
BB_CLIPS runs forever (in an idle state if no rules are on the agenda). e.g. (run -2)

A negative integer -n (less than -2).
BB_CLIPS runs until n rules have executed (in an idle state if no rules are on the agenda). e.g. (run -10)

The *halt* function or a keyboard intercept (e.g. control-C) may halt the execution of CLIPS at any time.

3.2 Runstart and Runstop Functions

A list of external functions that are executed at the end of each cycle of the inference engine (i.e. after each rule firing) can be created. This is done with the *add_exec_function* of CLIPS. In certain cases, however, it is useful to be able to execute special routines on entry or exit from the run command. The *runstart* and *runstop* functions of BB_CLIPS provide such a capability. Consider the situation where a simulation is being done and a clock driven by the time of day is used to keep track of the simulated time. When the system is stopped (when n rules have been fired after the (run -n) command or a control-C interrupt occurs, for example), the simulated clock should not advance. When the system continues, the clock should resume from where it left off when the system was stopped. In this case the addition of a runstart and a runstop function will allow the appropriate adjustments to be performed.

A function is added to the list of functions to be invoked when the run command is executed by calling the add_runstart_function and it can be removed from this list by calling the remove_runstart_function. Similarly, a function is added to the list of functions called when the run command is terminated by calling the add_runstop_function and removed with the remove_runstop_function[5]. The following are examples of calls to these four functions.

```
add_runstop_function("haltTimer",haltTimer);
add_runstart_function("continueTimer",continueTimer);
remove_runstop_function("haltTimer",haltTimer);
remove_runstart_function("continueTimer",continueTimer);
```

[5] These external functions must have been previously defined as user functions [1].

3.3 Recency Control Strategy

If there are a number of rules on the agenda with the same salience, CLIPS chooses the last rule that was added to the agenda for execution (thus implementing a *most-recent-first* control strategy). It has been found that for some systems it is more important to execute the first of the rules added to the agenda (i.e. execute the least recent, as opposed to the most recent). In BB_CLIPS this is done by invoking the *set_most_recent_first* function on the right hand side of a rule with an argument of true or false (the system default is true)[6]. The following is an example of a rule that will set the agenda manager to give preference to rules (within the same priority grouping) added least recently to the agenda.

```
(defrule change-recency
  =>
    (set_most_recent_first false)
)
```

4. Discussion

The additions described in this paper have proved useful in practice. A test program was constructed which simulates a series of tanks being filled by turning pumps on and emptied by opening valves. The system monitors the tank levels trying to keep the tanks below some high level mark and above some low level mark, raising alarms when these conditions are violated. It also has to plan the use of the pumps such that the total power consumption at any given time during peak periods in the day remains below some predetermined value (this is to avoid surcharges by the power company). Additional functionality was developed to complete the program. This included: (1) a simulator written in C to control the reading of level sensors in the tanks and to control actuators which turn pumps on and off and open and close valves as required; (2) a graphical interface using the NeWS [4] system on Sun microcomputers (see Figure 2); and (3) a suitable blackboard structure to partition the problem (partially shown in Figure 1). A detailed discussion of this problem can be found in [5].

Other ways to provide the features described in this paper are being considered. For example, allow the dynamic specification of rule features and the combining function rather than requiring the creation of a separate version of BB_CLIPS for each problem specific set of features; use a special assert function (*control_assert*) to handle assertions into the control blackboard rather than the assert function and the intercept rules described herein; and allow the dynamic specification of an agenda selection function which currently always selects the highest rated rule on the agenda.

Future work may involve determining how to most effectively use CLIPS in a multiprocessor environment and in collaboration with other expert and non-expert systems in a multi-paradigm environment.

[6] Calling the set_most_recent_first function has the same effect as executing an intercept rule in that it causes the reordering of the agenda to occur. This, however, causes some problems for the current BB_CLIPS implementation. It does not keep information that determines when a rule is added to the agenda. When the current agenda is reordered, some rules that were previously at different priorities may now have the same priority and it is not possible to determine which rule was added first to the current agenda. Subsequent agenda additions, though, are prioritized properly.

References

[1] Artificial Intelligence Section. *CLIPS Reference Manual, Version 4.3*. Lyndon B. Johnson Space Center, August 1989.

[2] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 26:251-321, 1985.

[3] V.R. Lesser, D.D. Corkill, R.C. Whitehair, and J.A. Hernandez. Focus of Control Through Goal Relationships. In *IJCAI*, pages 497-503, 1989.

[4] Sun Microsystems. *NeWS Manual*. 1989.

[5] A.C. Diaz, R.A. Orchard. A Prototype Blackboard Shell Using CLIPS. Submitted to the *Fifth International Conference on AI in Engineering*, 1990.

omiT

A11 Session:
Parallel and Distributed Processing I

CLIPS meets the Connection Machine or How to create a Parallel Production System

Steve Geyer
MRJ, Inc.
10455 White Granite Drive
Oakton, Virginia 22124

Abstract

Production systems usually present unacceptable runtimes when faced with applications requiring tens of thousands to millions of facts. Many efforts have focused on the use of parallelism as a way to increase overall system performance. While these efforts have increased pattern matching and rule evaluation rates, they have only indirectly dealt with the problems faced by fact burdened applications. We have implemented PPS, a version of CLIPS running on the Connection Machine, to directly address the problems faced by these applications. This paper will describe our system, discuss its implementation, and present results.

1 Introduction

As production systems have been used to implement a wider and wider range of applications, the limits of current technology have been stretched. One particularly sensitive limit has been the problem size and how this size impacts the total runtime of a system. Most systems degrade rapidly once their size limits are reached. Indeed, the acceptable runtime is often an important, if not the most important, factor in setting an upper limit on problem size. Many applications have had to wait for technology to mature enough to support the application's minimum acceptable problem size.

Several factors influence the size of a problem. Two common factors are the number of facts manipulated by the application and the number of rule evaluations required to come to a solution. Studies demonstrate that many production system spend 90% of their total time matching facts to rule patterns. In an attempt to create more efficient systems on serial computers, algorithms have been developed to optimize this task. Rete is the most commonly used algorithm [1]. This algorithm can efficiently manage large numbers of simultaneous pattern queries, and as queries are completed, Rete updates the list of rules ready for execution. Rete caches internal data structures to remember partially matched queries and the number of cached entries increases rapidly as facts enter working memory. The memory required by these data structures and the computation necessary to manage them sets a practical limit on how many facts can be placed in working memory.

Many production systems built on parallel hardware have also focused on efficiently matching facts to patterns. Parallel pattern matching does support large rule sets and increases the rate at which facts can be processed. However, if the application is fact driven, the resources consumed in parallel pattern matching can overwhelm the increased resources brought by the parallel architecture. This is especially true if the parallel pattern matching algorithm caches partially matched queries. Special procedures are necessary when designing production systems that will process applications with large numbers of facts.

We are interested in problems requiring tens of thousands to millions of facts. Some examples are simulation and modeling, package/vehicle scheduling, intelligent databases, and low-to-mid level processes for image understanding. In each of these applications areas, many real world problems demand more facts than can be processed by current production systems. To get these systems away from the laboratory and running real world problems will require new techniques. We have developed PPS to explore one possible technique.

This paper is organized as follows: Section 2 presents necessary background material and describes the algorithmic approach taken by PPS. The changes made to CLIPS to create PPS are discussed in Section 3. This section can be skipped by those uninterested in implementation details. Experimental results are presented in Section 4 followed by a discussion of potential enhancements in Section 5. The paper finishes with a summary and conclusions in Section 6.

2 How PPS works

This section describes PPS. It begins with a description of the Connection Machine and explores the features that makes the CM well suited to this problem domain. Next it discusses the choice of CLIPS as a software base and describes the syntax changes necessary to allow CLIPS programs to run on PPS. Finally, the section will discuss the internal changes necessary to CLIPS to allow parallel execution on the Connection Machine.

2.1 The Connection Machine

The Connection Machine, or CM, is a parallel computer architecture that supports between 4 and 64 thousand separate processors. Figure 1 is a pictorial diagram of a CM. Each individual processor has a local memory, an ALU (Arithmetic Logic Unit), and a general inter-processor communication system. All processors share the same instruction stream supplied to them from a front end computer. Individual processors can perform separate operations by executing or ignoring, selectively, the sequences of instructions supplied by the front end. More complete technical information can be found in reference [2]. The CM has several properties that separate it from the other parallel architectures commercially available.

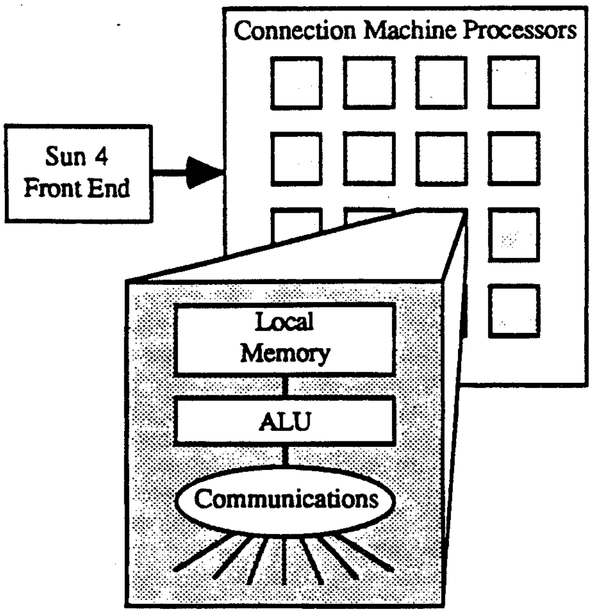


Figure 1. The Connection Machine

By supporting thousands of processors, the CM encourages the programmer to focus on how the data is manipulated and how it interacts with other data. This is in contrast to more conventional multiprocessors where the focus tends to be on the parallel algorithm's flow of control. The CM system software supplies even more flexibility by creating "virtual" processors. The programmer can choose the number of processors necessary to solve a problem and the CM will automatically divide the physical processors into virtual ones. The CM does constrain the number of virtual processors to be a power of two. With the vast number of processors available, it is natural to place each data structure manipulated by a program into a separate virtual processor. Each data structure can then be viewed as having its own processor to perform any computation required.

The CM has a general purpose, hypercube based, communication system that allows each processor to efficiently communicate with any other. Virtual processors generalize this system to allow communication between themselves. Some specialized operators have been created on top of the communication system to perform certain functions very rapidly. Important to PPS are the operations that allow the CM to rapidly replicate data from thousands of virtual processors to thousands of others and a mechanism that allows all active processors to enumerate themselves.

The most idiosyncratic property of the CM is how instructions are supplied to the processors. The CM is a Single Instruction Multiple Data or SIMD machine. While each processor in the CM has its own memory for data storage, it must share its instructions with all others. Each processor has a context flag to control its individual execution of the instructions supplied to all processors. For example, if an *if then else* is reached, all processors calculate the *if* expression together. Those processors failing the *if* test will have their context flag cleared and the remaining processors will execute the *then* clause. The context flag is reversed and those failing the *if* test execute the *else* clause. The context flag is then restored to its original value and execution proceeds. Some efficiency is lost as one or more sets of processors are disabled. The advantage of this approach is that the individual processors and local memory can be made simpler and smaller and hence the CM is able to have thousands of physical processors. For PPS, the SIMD nature of the CM is not limiting and having thousands of physical processors is very important.

The front end processor is responsible for supplying instructions to the CM processors and performing serial computations not well suited to the CM. The front end also supplies the development environment, editors, and the file system. The work done on PPS was performed on a Sun-4 front end.

2.2 Software Considerations

Many reasons support the decision to base PPS on top of CLIPS. Compared to any system we may have built from scratch, CLIPS is a mature system. It was already supporting a user community and was actively being used to write production systems. By starting with CLIPS, we would only need to write and debug those sections of code necessary for parallel evaluation. From a users point of view, PPS only requires small additions to source syntax which allow serial versions of CLIPS, with their debugging tools, to be used to debug productions destined for parallel evaluation. Finally, C source code was supplied with CLIPS without the need for complex negotiations with a vendor.

To avoid losing the advantages gained by basing PPS on CLIPS, it was important to keep the programmers view of PPS very close to CLIPS. Extensions and restrictions from standard CLIPS syntax should be limited in nature and necessary to support parallel evaluation. The major innovation of PPS is to break facts into serial and parallel groups. The first word of a fact is used to determine the fact's class (in a manner similar to *deftemplate*). The programmer can choose to place certain classes of facts into parallel working memory and they will automatically be processed by the CM. Serial facts are processed by the normal CLIPS mechanisms. The programmer chooses where PPS places facts based on the number of facts in a class and the type of operations performed on these facts.

The first field in a parallel fact is constrained to start with a word which specifies the fact's class. Only classes designated by the programmer will be placed on the CM. Parallel facts must also be of a fixed length and each field of the fact must have its data type specified. Multifield variables are excluded in rule patterns. These restrictions are to lessen the CM memory requirements and to avoid dynamic allocation. Future versions of PPS could lift these restrictions.

The form *deffactfields* creates a fact class and allows a detailed description of the fact's contents. Its syntax is:

```
(deffactfields classname
  parallel | serial
  (fieldname type)...)
```

The first argument, *classname*, defines this word as a fact class. The next argument is either parallel or serial and it specifies how to process this class. The rest of deffactfields is a list of field names followed by their data type. The standard CLIPS data types have been extended to include integer and boolean. Facts whose first word has never been described with a deffactfields are assumed to be serial facts.

Once a class of facts has been described as a parallel class, the system will automatically place all facts belonging to that class into the parallel working memory. All work required of the parallel working memory is performed on the CM.

2.3 Parallel execution in PPS

As stated earlier, PPS splits the working memory into a serial and a parallel part. When a rule enters PPS, Rete (the standard CLIPS algorithm) is used to compile and process the serial patterns. Parallel patterns are converted to queries of parallel working memory and these queries are attached to the rule body. During execution, Rete manages the serial patterns and when they have matched, the rule is placed in a queue, ready for execution. Upon a rule's execution, a parallel query is performed to collect matching parallel facts, and the rule's body is evaluated in parallel over these facts. In PPS, a single parallel rule evaluation processes all facts that currently match its pattern. The large number of processors available on the CM makes the cost of processing a rule almost independent of the number of facts that it matches. Efficiently processing facts in parallel is very important as the number of facts increases to millions.

There is no certain knowledge that any fact or combinations of facts will actually match the rule pattern. Since it is not known when there is work for a parallel rule to perform, they have to be periodically executed. Currently, PPS uses a simple round robin approach to schedule parallel rules. After each execution, a parallel rule will place itself at the end of the agenda for future execution. This gives other rules an opportunity to execute before reevaluating the current one. Execution terminates when all serial rules have been removed from the activation agenda and no parallel rules are able to find facts or combination of facts not already evaluated. It is assumed that parallel rules, on average, will find many facts to evaluate and this will mask the inefficiencies caused by extra rule evaluations. Section 5 discusses other, more efficient, control strategies.

Since the scheduling scheme used by PPS allows rules to be executed many times, some mechanism is necessary to eliminate the reevaluation of a rule over facts already processed. A global time, based on the number of rule evaluations, is maintained by PPS. Each fact in parallel working memory is timestamped by the rule creating or modifying it. When rule evaluation begins, the timestamp of its previous evaluation is compared to each fact's timestamp. This comparison identifies facts that have entered working memory since the rule's previous evaluation. Only facts, or combination of facts, more recent than the rule's previous evaluation are processed in the current evaluation. This mechanism eliminates the reevaluation of facts by rules.

For PPS to execute efficiently, the CM must be able to query the working memory in parallel, get all matching combinations in parallel, and evaluate the resulting matches in parallel. The CM places each parallel fact into it own virtual processor and is quickly able to query these facts, filter out the uninteresting ones, and create matches. The matches end up in separate processors and all matches are simultaneously available for execution of the rule body. Since the matching and evaluating of rules happen together for all facts, the SIMD nature of the CM has no negative impact on how PPS performs. Instead, it has simplified the writing and debugging of PPS.

What advantage can PPS gain by replacing the Rete algorithm with a potentially expensive database query? When processing a million facts, Rete would have to create millions of intermediate data structures to hold pending queries. These structures would consume megabytes of storage and the management of this storage would place a large computational burden on the system. In its place, PPS requires only a small (32 bit) fixed memory cost per fact. The cost of querying can be justified as long at the average number of matches and subsequent rule evaluation is faster than performing a similar match and evaluation in another manner (such as Rete). With millions of facts being queried, it is possible that, on average, hundreds or thousands of facts will match each rule execution. Under these conditions, PPS can perform better than alternative methods.

This section has outlined the approach taken by PPS. PPS is more memory efficient than Rete and, under the proper circumstances, PPS will also be more time efficient. The next section will outline that changes necessary to create PPS from CLIPS.

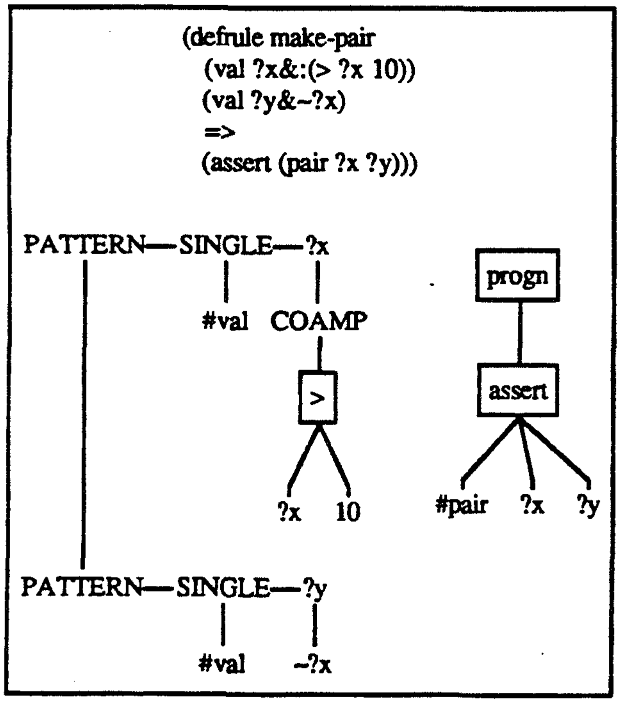
3 Implementation

In order to create PPS, it was necessary to modify the normal CLIPS processing in several places. This section will begin with a short description of how CLIPS compiles and evaluates rules. This is followed by a description of how the rule compiler was modified. Finally, the changes to rule evaluations are outlined.

3.1 Normal CLIPS processing

CLIPS uses *defrule* to create a rule. When CLIPS receives a defrule, it creates two descriptions of the rule being processed. Figure 2 is an example rule with the two descriptions created by CLIPS. The lower left diagram in Figure 2 shows the internal structure of the rule's pattern, the lower right is the internal description of the rule body.

After CLIPS creates the pattern description of the rule, it checks the pattern for internal consistency. The Rete tree builder is then called to create appropriate modifications to the discrimination and join network. Finally, the rule body is processed and it is attached to the Rete join network. A more complete description of this process can be found in reference [3].



```
(defrule make-pair
  (val ?x&:(> ?x 10))
  (val ?y&~?x)
  =>
  (assert (pair ?x ?y)))
```

Figure 2. Sample rule and its internal representation

CLIPS has a built in expression evaluator used to evaluate expression trees. Each box in Figure 2 contains a function that CLIPS will evaluate with the expression evaluator. The lines under each box, connected to other objects, are the arguments required by this function. Each function determines the values of its arguments and then performs its operation. Expressions and the expression evaluator are use both to evaluate the rule body and evaluate conditions inside the Rete algorithm.

PPS interrupts the normal compilation process in two places, the processing of patterns and the processing of the rule body. It also extends the expression evaluator.

3.2 PPS Pattern Compilation

PPS steps in after CLIPS has created the pattern description. It separates the pattern clauses that match serial facts from those that match parallel facts and reorders the clauses to have the serial ones first. If no serial clause is found, one will be created to match the fact *initial-fact*. After the standard internal consistency checks are made on the reordered pattern, the serial clauses are passed to the Rete tree builder for normal processing. The parallel clauses are passed to a pattern compiler which converts rule patterns into equivalent database queries. These queries will be attached to the rule body in a later stage in the processing. From this point the CLIPS processing precedes in a normal manner.

Since PPS uses queries to match the rule patterns, each legal CLIPS pattern must be converted into a database query. These database queries are made up of restricts and joins. Restricts are used to select a subset of the original database based on some conditional test. Joins create new databases containing the possible permutations of the input databases. For example, if two databases had elements (A B) and (1 2 3), the joined database would have the elements (A1 A2 A3 B1 B2 B3).

The pattern compiler examines the rule pattern and creates a database query. Restrict is generated when a pattern limits the value of some field. A join is generated to combine each new pattern clause to the ones already processed. The number of joins will be one less than the number of clauses in the pattern. Where a restriction is placed depends on the required information. If all the information is found in the clause being processed, then the restrict is placed before a join. If the clause needs information from other clauses in the pattern, then the restriction is placed after the join.

In the example found in Figure 2, the rule has a pattern that searches for two facts named *val*. When they are found, the variable ?x is constrained to have a value

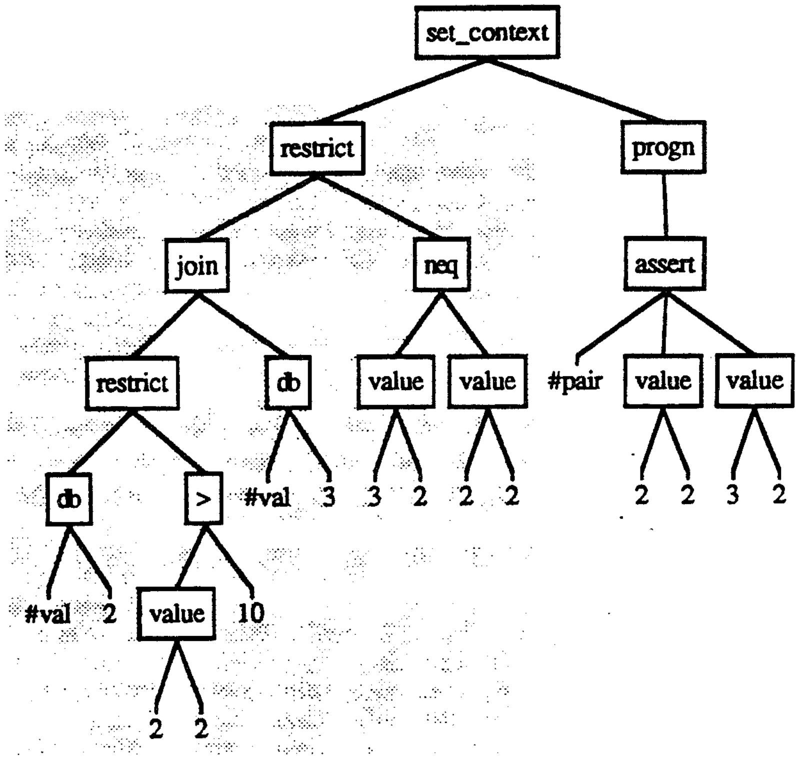


Figure 3. Database query merged with rule body

greater than 10 and variable ?y must not have the same value as ?x. When a pair of facts matches all these constraints, a new fact named *pair* is asserted into working memory using the fact variables found in the pattern. The results of PPS pattern compilation can be found in the gray area of Figure 3.

The pattern compiler begins by examining the first PATTERN in Figure 2. The first field of this pattern, a SINGLE with a name of *val*, describes the fact's class. The pattern compiler uses this fact class to create a new database (the left hand most db in Figure 3). Processing continues on this clause at the ?x. This variable is found to have an expression constraining its value (found under the COAMP). Since this constraint only uses information found in the current clause, it immediately creates a database restriction to evaluate the constraint (the left hand most restrict in Figure 3). This clause is now finished and the processing is begun on the next PATTERN. Like the previous clause, this pattern expects a fact class of *val*, so another database is created. Since ?y has a constraint dependent on another clause, the creation of the restrict is delayed and the two databases are combined with a join. Processing finishes with a restrict being created to constrain ?y from having the value in ?x (the remaining join and restrict in Figure 3).

This query, seen in Figure 3, is equivalent to the original pattern. The execution of this form would be as follows. The db in this expression creates new databases from the original facts. The arguments to db are the name of the fact class and the index number of this pattern. Parallel patterns always start at two or greater since there is always at least one serial pattern in every rule (remember that one is added if none exist). The result of the far lower left db is passed into restrict. Its two arguments are a database and an expression. This restrict limits the database to facts having a value for field 2 of pattern 2 (or ?x) greater than 10. The result of this restrict is passed to the join. Join's arguments are always two databases. The second database entering this join is created frọm the original facts. Once these databases are joined the resulting database is passed to the top level restrict. This restriction forces the value for field 2 of pattern 3 (or ?y) to be different from field 2 of pattern 2 (or ?x).

Once the database query is created, it is ready to be attached to the rule body. The function *set_context*, seen in Figure 3, takes a database on its left and prepares it for the rule body evaluation. Then the rule body, on the right, is executed. Only one more step is necessary to finish preparing the rule for execution. It will be examined next.

3.3 PPS Expression Compilation

The parallel expression compiler examines expressions to find parallel computations. The standard functions found in the original expression are replaced with parallel equivalents. This module makes use of the data supplied by *deffactfields* to determine what to should be made parallel. Overall, this module uses standard compiler techniques to compile expressions. It keeps track of each source datatype and can convert between different datatypes as appropriate. The only twist is that the CM allows all operands to have variable length and when parallel instructions are emitted, they must include lengths. The final results from the example problem can be see .in Figure 4.

In this final expression tree, various functions have been converted into CM versions. For example, the > found in the first restrict has been converted into a *cm_i_gt* (or CM Integer Greater Than). This function will be performed on the CM and will be applied to all facts in the database at once. The function *cm_conv_si_pi* converts serial integers, 10 in this example, into parallel integers. The 32 appended to various functions is the bit length of the operand. Finally, all references to fact variables is converted into *cm_get_var*. This routine will use current databases to acquire a value for computation. The CLIPS expression evaluator has to be extended to allow PPS to evaluate parallel expression trees.

3.4 PPS Expression Evaluator

The expression evaluator is extended by adding new data types and by creating new parallel functions. Two data types have been added to the standard CLIPS set. One type is used for parallel databases and the other for parallel variables. The parallel database is used by restrict, join and set_context to identify which database is being manipulated. The parallel variable type points to an address in CM memory. All the parallel arithmetic and boolean functions return this type.

PPS creates a separate set of virtual processors for each class of parallel facts. Each field of a fact is stored in separate parallel variables inside the virtual processors. The system also creates two auxiliary variables. The first is an in-use flag which determines active facts. The second is a timestamp which holds the time this fact was created or last modified (see section 2.3). When facts enter parallel working memory, a free virtual processor is selected and its data fields are initialized. The in-use flag is set and the timestamp is initialized to that of the current rule. When facts are retracted, the in-use variable is cleared.

A PPS database also creates a set of virtual processors. Each virtual processor contains a set of indexes, an in-use flag, and a recent-fact flag. Instead of managing the fact data directly in a database, indexes are used to point to the processors containing the actual facts. When joins merge two databases into a new one, each active processor of the new database has one index for each source database. The in-use flag is true in all processors that contain active database information. The recent-flag is true for database entries that contain facts whose timestamp designates this fact as a recent fact and therefore requiring rule evaluation.

A separate function is used to evaluate each parallel instruction. Each arithmetic and boolean function acquires its operands and invokes the appropriate CM instruction to perform its function. For example, the *cm_i_gt* function in PPS acquires its operands and calls the *CM_s_gt_1L* instruction on the Connection Machine.

The database instructions manipulate the database data structures. The *db* function creates a new database. It examines the database's source facts and initializes a set of virtual processors with the appropriate information. The *cm_get_var* function uses the information in the CM database description and returns the value from this field in the fact. The restrict and join commands directly manipulate the database data structures. Restrict modifies the in-use flag and join creates a new database whose indexes point to the source facts of the original databases.

This section has outlined the major modification made to CLIPS in the process of creating PPS. These modifica-

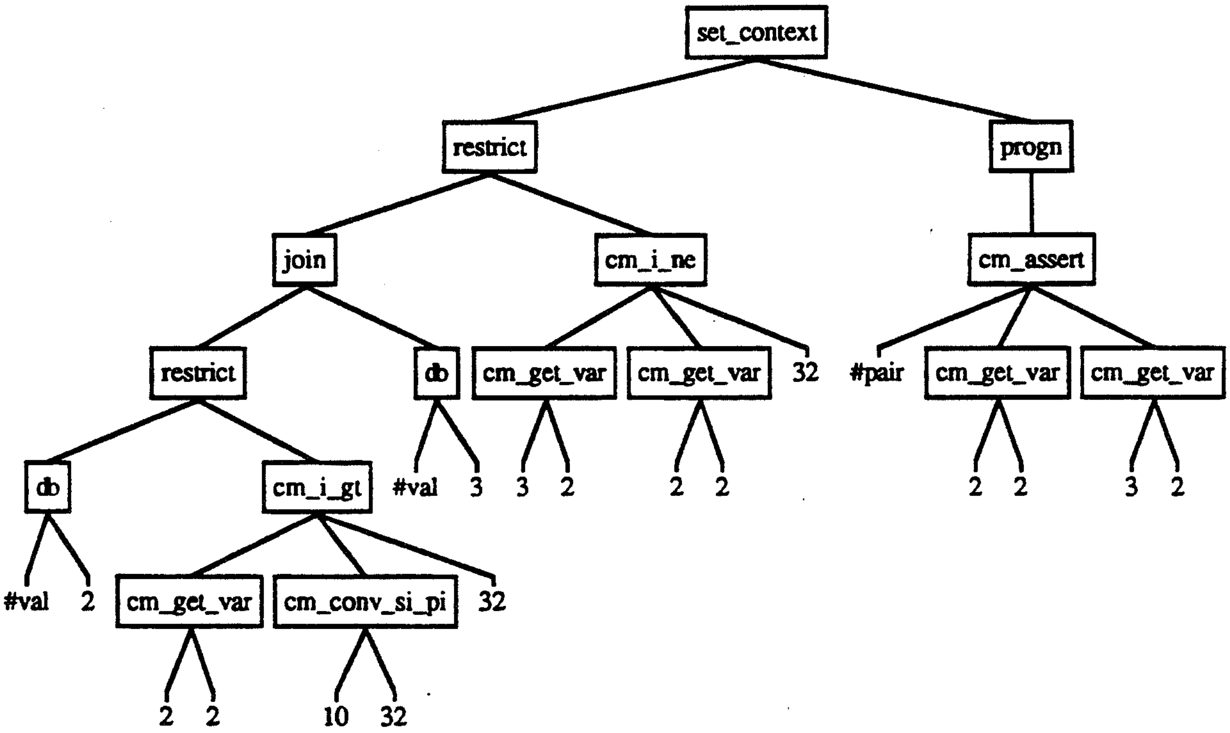


Figure 4. The final parallel expression

tions have focused on converting parallel patterns into database queries and giving the expression evaluator the ability to evaluate expressions on the CM.

4 Performance results

This section will describe the procedure used to test the performance of PPS. Two different tests will be used to compare the PPS results to those of CLIPS.

Both CLIPS and PPS were run on the same Sun-4. In addition to the Sun-4, a CM-2A with 8K processors was used for the PPS benchmarks. Runtimes were measured using the Sun-4 system clock (with *ftime*) and they represent the wall clock runtime. We chose to perform the benchmarks without disabling the normal background processing performed by the Sun-4. This processing occasionally caused small hiccups in the data.

A special command has been added to CLIPS and PPS to perform a benchmark. This command performs a series of runs differing only in the number of facts processed. Runs begin by marking the start time and then entering the correct number of facts into working memory. They are of the form (x *index*) where *index* is from 1 to the number of facts being tested. The production system is started and allowed to run to completion. Finally, a stop time is recorded and the total runtime presented to the user.

The first benchmark examines the ability of a production system to perform simple pattern matching with field values being restricted. The rule used for the benchmark was:

```
(defrule test-rule-1
  (x ?i&:(evenp ?i))
  =>
  (assert (y ?i)))
```

This rule examines working memory for any fact of the form (x ?i) where ?i is constrained to be an even value. When such a fact is found, a new fact (y ?i) is asserted into the working memory. Given the initial facts (x 1) (x 2) (x 3) (x 4), this rule will assert (y 2) and (y 4).

Figure 5 displays the results of PPS as the number of input facts runs between 2048 and one million. The steps seen in this graph are a result of the number of virtual processors required to process the input facts. Since the number of virtual processors is constrained to be a power of two, steps form in the data each time the number of facts forces the CM to go to the next higher size of virtual processors. Once some virtualization level is reached, the runtimes are independent of the number of facts. Since each fact being matched is processed by a separate virtual processor, and there are no interactions between facts (in this benchmark), then the runtime for one fact is the same as that for many. This is very encouraging. To process one million facts, the 8K processor CM-2A took 4.55 seconds. If, however, a CM with 16K processors were available, it would be processing one million facts at the next lower step, or in 2.26 seconds. If a 64K machine were available, the runtime for one million facts would be .57 seconds.

The benchmark run for CLIPS was between 128 and 8192 input facts. As the number of facts increase, CLIPS total runtime increases mostly linearly. Figure 6 shows the results of CLIPS against those of PPS (originally seen in Figure 5). Compared to the PPS, CLIPS increases its runtime very rapidly.

The second benchmark examines how well a production system can perform matches that require more than one fact. The benchmark used the following rule:

```
(defrule test-rule-2
  (x ?i) (x ?j)
  =>
  (assert (y ?i ?j)))
```

This rule will create a (y ?i ?j) with all combinations of indexes found in the (x ?) facts. Given the initial facts (x 1) (x 2), this rule will assert (y 1 1) (y 1 2) (y 2 1) (y 2 2). The number of facts asserted into the working memory is the square of the input facts.

The result from this benchmark can be seen in Figure 7. Both the results from PPS and CLIPS are displayed together. CLIPS was run between 8 and 160 facts and PPS between 16 and 560 facts. At their maximum, CLIPS will assert 25.6K facts (from 160 input facts) and PPS will assert 313.6K facts (from 560 input facts). The runtimes of CLIPS nearly form a parabola and clearly show that for CLIPS the work increases quadratically to the number of input facts. The PPS results show a slow growth in runtime as the number of facts increase.

A real application using PPS is currently under development. When this application is finished it will be possible to get a better understanding of how PPS scales with a mixture of rules. Since the two benchmarks tested represent the major operations performed in production systems, we are optimistic the results will be good.

5 Future work

Most of the restrictions discussed in Section 2.2 were solely for the purpose of making the development of PPS simpler. These restrictions were created primarily to avoid dynamic memory allocation on the CM. With some modification on how parallel fields are managed, variable length facts and strings could be supported. The need to specify field datatypes could also be eliminated.

Section 2.3 discussed the round robin scheduling of parallel rules and mentions that better approaches are possible.

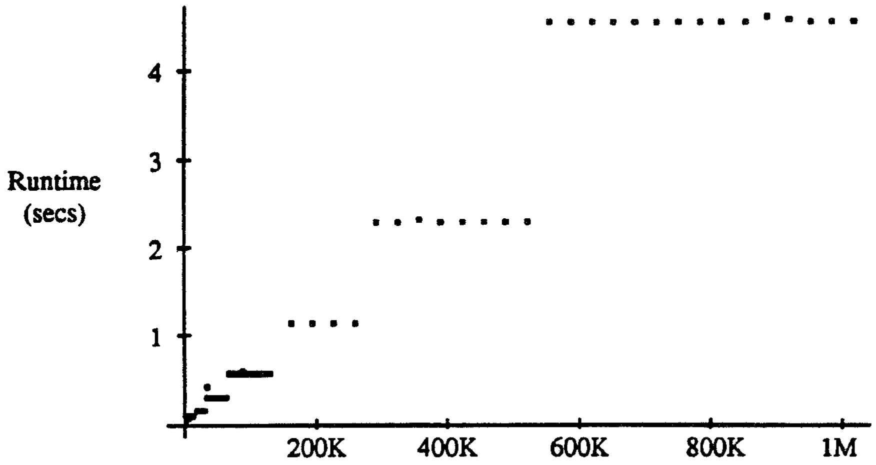


Figure 5. PPS runtimes for the first benchmark as the number of facts processed goes to one million.

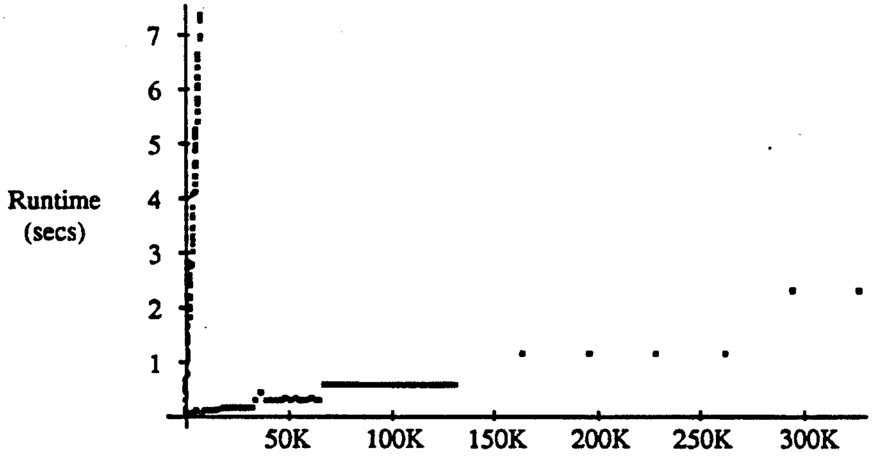


Figure 6. CLIPS versus PPS for the first benchmark. Graph has been rescaled from Figure 5.

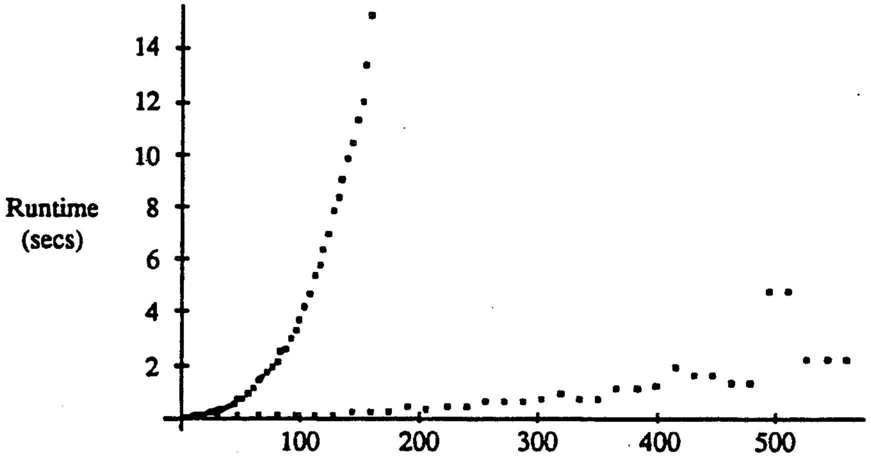


Figure 7. CLIPS versus PPS for the second benchmark.

One approach would be to modify the Rete algorithm used in CLIPS. While it would not be wise to have the Rete algorithm identify and record all partial matches in parallel memory, specialized parallel tests could be added to the Rete tree to determine if at least one parallel fact matches each of the clauses found in a parallel rule pattern. By knowing at least one fact matches each clause of the rule pattern, it has a higher probability of performing useful work when executed. This should increase the overall efficiency of the system.

Another improvement would be to group multiple rule queries together. Subqueries used by more that one parallel rule would only have to be performed once and their answers could be used by all. This is very much like the discrimination network used in Rete which also performs common pattern tests and shares results. It is even be possible to merge this approach with the modification to the Rete algorithm discussed above to seriously limit the unnecessary database queries performed by PPS.

Before attempting any of these modifications, we wish to gain a better understanding of the current performance of PPS. In this way, we will be better able to understand the impact made by changes.

6 Summary and Conclusions

We have described a method, based on parallel database queries and parallel rule evaluations, that allow production systems to process large numbers of facts. Using this technique, CLIPS was modified to create PPS. This new system has a high degree of compatibility with its parent while allowing the user to build applications impossible to process on CLIPS.

Data has been presented demonstrating PPS's ability to perform well in fact rich situations. Particularly encouraging is how well PPS scales as the number of facts increase. Many runtimes are a function of the virtualization level of the CM and are independent of the number of facts being processed. In these situations, the runtime can be controlled by the number of physical processors supplied (which controls how many virtual processors will be emulated on each physical processor).

We are now applying PPS to a low-to-mid level image understanding problem. Since this task generates hundreds of thousands of facts, we believe that PPS is well matched to the problem. Based on the results of this project, we hope to apply PPS to other areas of interest. Some of these interest areas are simulation and modeling, package/vehicle scheduling, and intelligent databases.

References

[1] Forgy, C.L., Rete: A fast algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, 1982, pp 17-37.

[2] Connection Machine Model CM-2 Technical Summary, Version 5.1 May 1989, Thinking Machines Corporation, Cambridge, Massachusetts.

[3] CLIPS Architecture Manual, COSMIC.

Mr. David G. Goldstein
2900B Cliffridge Lane
Fort Worth, TX 76116
Affiliation: University of Texas, Arlington

PRAIS: Distributed, Real-time Knowledge-Based Systems Made Easy

This paper discusses an architecture for real-time, distributed (parallel) knowledge-based systems called the Parallel Real-time Artificial Intelligence System (PRAIS). PRAIS strives for transparently parallelizing production (rule-based) systems, even when under real-time constraints. PRAIS accomplishes these goals by incorporating a dynamic task scheduler, operating system extensions for fact handling, and message-passing among multiple copies of CLIPS executing on a virtual blackboard. This distributed knowledge-based system tool uses the portability of CLIPS and common message-passing protocols to operate over a heterogeneous network of processors.

I. Introduction

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

This paper introduces PRAIS, the Parallel Real-time Artificial Intelligence System, a cost-effective approach to parallel and real-time computing. PRAIS embeds the 'C' Language Integrated Production System (CLIPS) into a blackboard architecture with artificial intelligence specific operating system extensions and standard communication mechanisms to provide a flexible development environment for distributed knowledge-based systems. The goals of PRAIS is to simplify parallelization, increase portability, and maintain a consistent knowledge representation throughout the system. Accomplishing these goals should dramatically lower the costs of developing and using sophisticated artificial intelligence software.

II. Blackboard Architectures

The blackboard architecture [Nii86] has probably been the most successful architecture for addressing complex problems. This architecture features multiple, independent knowledge sources (KS's) each of which reasons about a portion of the problem. Knowledge sources share a global data structure (the blackboard) to share information, in an analogy to experts examining data and hypothesizing solutions on an actual blackboard.

The blackboard architecture has been adopted for several reasons. First, each knowledge source has its own knowledge-base (KB, a database of knowledge driving reasoning), thereby partitioning the system's knowledge, reducing rule interactions, and making the system easier to understand and program. Blackboards also facilitate hierarchical problem-solving; results from lower level knowledge sources can be used to drive the reasoning of higher level knowledge sources.

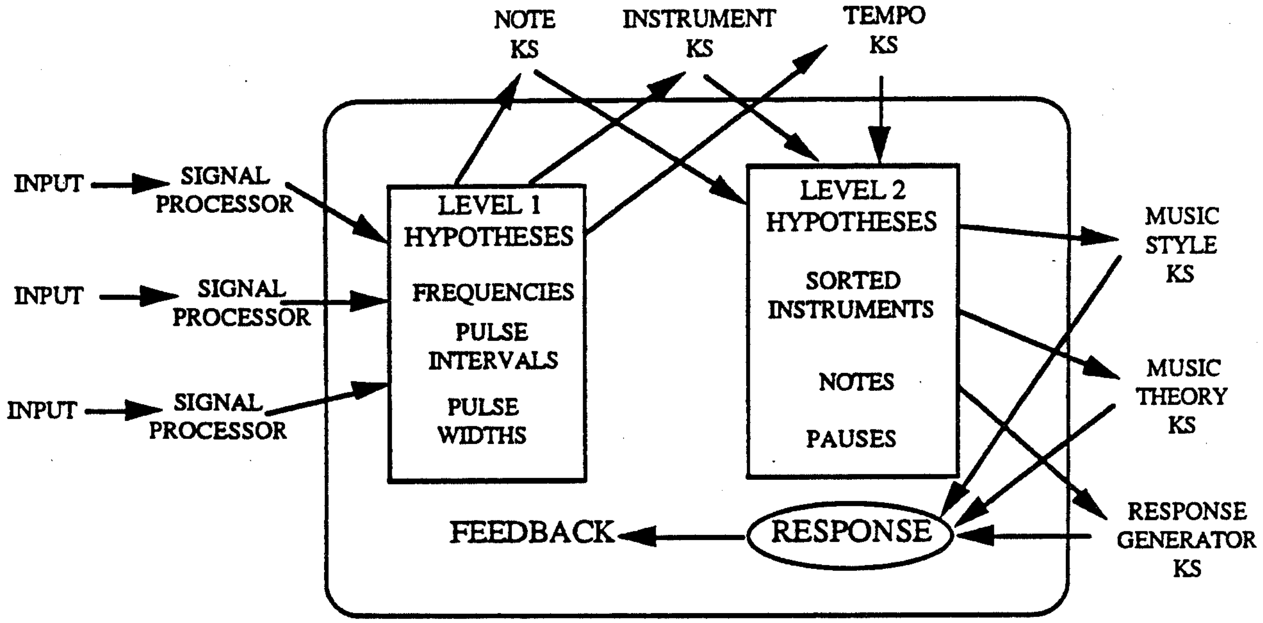
This hierarchical development of hypotheses is very useful, especially useful for problems where disparate data is encountered from multiple sources (e.g. vision, multi-sensor fusion).

The execution of knowledge sources are typically controlled by an external mechanism which activates knowledge sources based upon the blackboard's current state. The control module would normally be quite complex, since decisions it would have to make would include on which processor, for how long, and on what data a given knowledge source should execute. However, PRAIS simplifies control by determining during compilation what, when and for how long each knowledge source needs to execute, so no artificial mechanism for knowledge source activation is required. The distributed nature of the processing is accomplished by simply communicating facts asserted via either (1) a global memory, (2) messages, or (3) in the local fact database.

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

A distributed blackboard permits dedicated processors to handle specific tasks: sensor data can be filtered by signal processors, numeric computations on conventional processors, and symbolic reasoning on LISP machines. Information from one level of the hierarchy can be used at other levels of the hierarchy, and processors at different levels can explore different granularities of a solution in parallel.

Figure 1 - Hierarchical Blackboard Processing



III. Data Representations

The embedding of CLIPS in a blackboard architecture provides a tremendous degree of flexibility. However, choosing the proper data representation is possibly the most crucial aspect of any large KBS because as systems grow in size:

(1) data interactions become more subtle and difficult to predict,

(2) the entire collection of data may not be observable, and

(3) organizing even the initial state may become complicated.

Therefore PRAIS considers parallelization from a data-oriented perspective; facilitating CLIPS rule development drives the system's design. Productions in PRAIS appear almost identically as they do in CLIPS (see Figure 2). These productions are modified only to enhance real-time processing by adding the "importance" definition and a list of

Figure 2 - Production Format

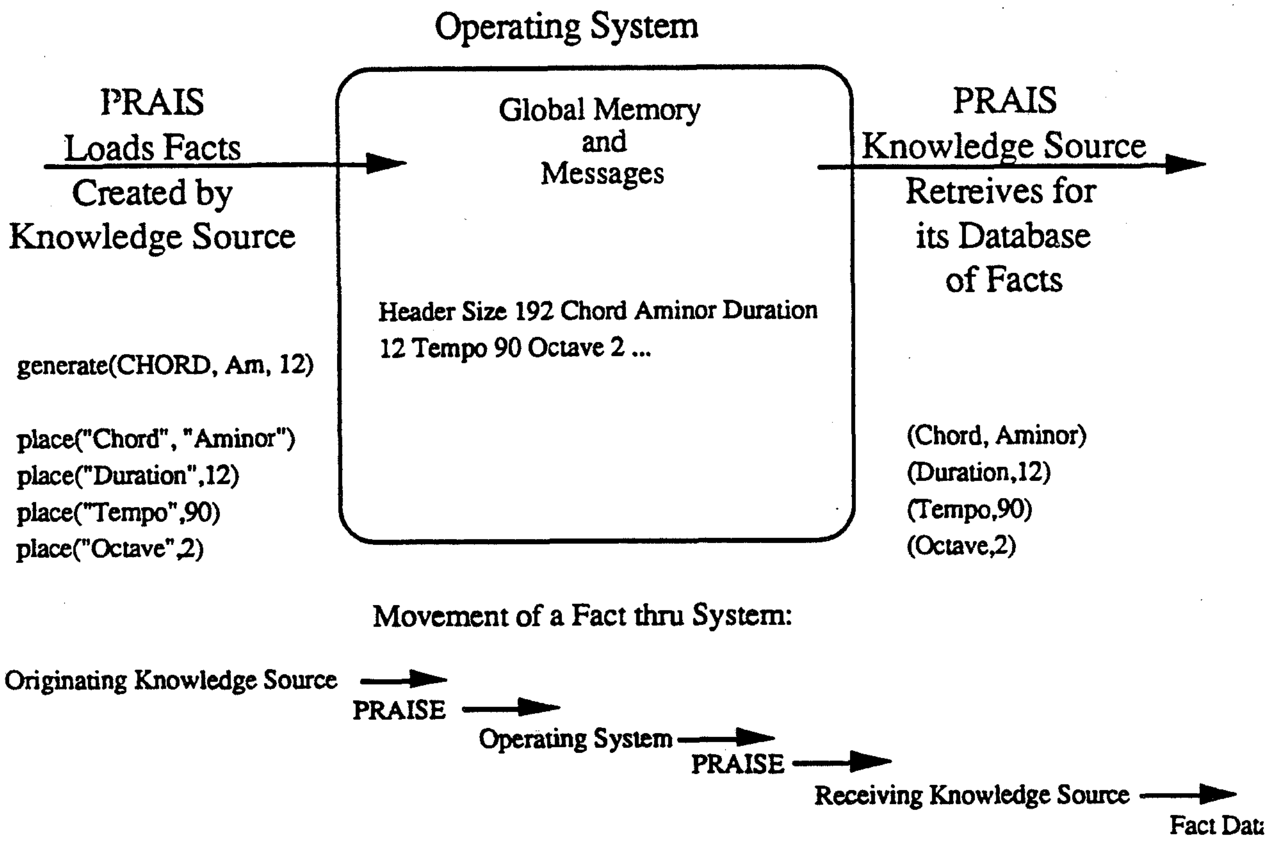
```
(defrule {rule-name}
  (salience {set of (times, priorities)})
  (importance {mandatory/optional/dropable})
    ({left-hand-side patterns})
=>
   ({right-hand-side actions})
)
```

salience values at specified times. Already developed CLIPS code can be used in PRAIS without change. Also, as information is used thruout the system, a syntax is used which resembles CLIPS facts as closely as possible (see Figure 3). This simplification of representation is especially useful in developing complicated, mixed-language, multi-platform applications; CLIPS is especially useful in such endeavors, since it supports 'C', FORTRAN, and Ada [CG].

IV. Operating System Extensions

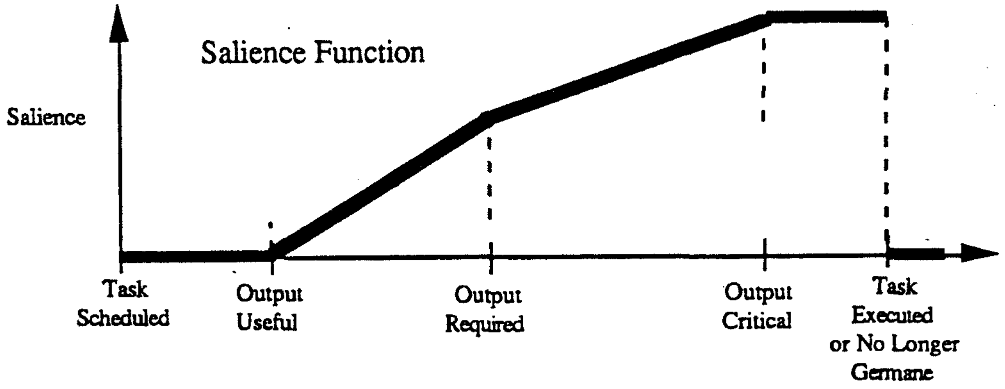
PRAIS provides real-time control, reasoning specification and interruption, and process migration of reasoning tasks as extensions to the operating system . The real-time control mechanisms incorporate salience functions, generated at compile-time, which dynamically reflect the system's current state by considering the timeliness of a given task/rule. A task's salience is initially low, and increases as the task becomes more important, until it become mandatory (see Figure 4). Untimely or lesser important tasks can be dropped by the system to provide more processing power for more important tasks. This allows the system to prioritize an approaching anti-tank missile over determining the optimal path across current terrain.

Figure 3 - Uniform Knowledge Representation



A producer/consumer model has been adopted (since facts typically travel by messages) and facts migrate with a process capable of reasoning. If processor A is overwhelmed with the amount of reasoning it has to perform it broadcasts a request for assistance. If processor B is the least loaded processor in the system, it is also the most likely to respond soonest, and so the first responding task receives both the necessary rules and facts to perform the reasoning task, performs the reasoning, and sends the results to processor A. Other extensions

Figure 4 - Time-Varying Salience Function



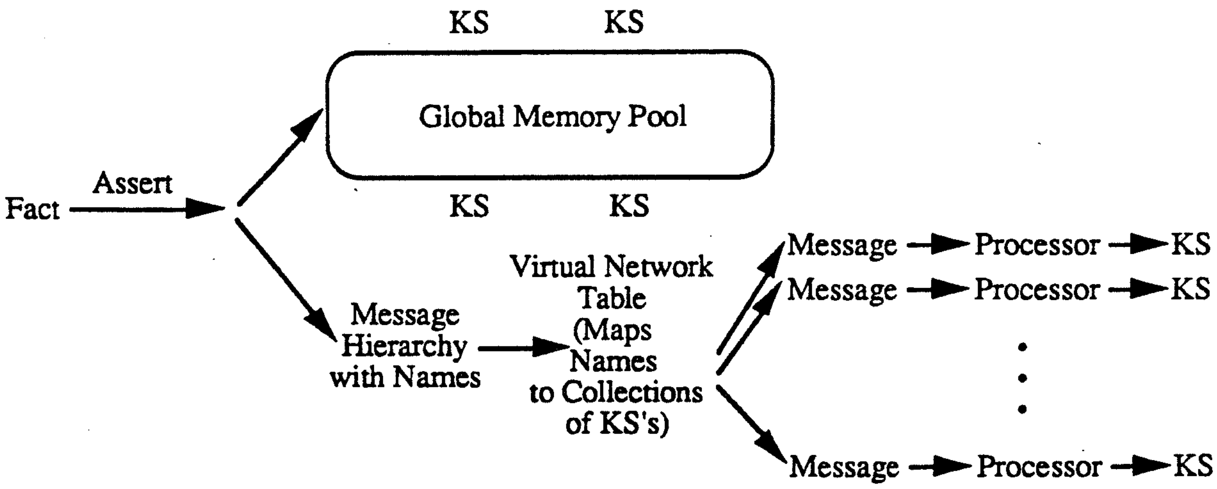
include signal-based functions to interrupt reasoning at any time and remote procedure calls to transmit information or to control the amount of time with which a knowledge source may reason.

V. Communication Mechanisms

PRAIS provides coarse-grained parallel execution based upon a virtual global memory. PRAIS is also both language and system independent, simply providing the user with a global assert command to enact parallel processing. The system is economical since: (1) relatively few changes must be made to parallelize existing software, (2) inexpensive, commonly available processing resources can be used, and (3) few hardware-specific details must be considered by users. Because PRAIS is easy for its users to work with and will operate on a variety of platforms, PRAIS offers inexpensive parallelism.

PRAIS also has a variety of features that make it appropriate for distributed knowledge-based system development. First, a deployed knowledge source communicates via message-passing to hierarchies of names knowledge sources (see Figure 5). By partially ordering the classes of message recipients communications can be minimized while: (1) replicated copies of knowledge sources can be treated uniformly, (2) processing tasks can be referenced by classes of knowledge sources (without refering to processors) and (3) virtual communications networks can be established.

Figure 5 - Passing Facts Among Knowledge Sources



Another enhancement is that tasks can be forked to use either the dynamically changing global memory or to use the global memory available at the time of the fork, without incorporating any future updates. This feature is important in applications such as game-playing and certain types of simulations where all relevant details up to the time of the fork are important, but any future details would corrupt the hypothetical universe under consideration.

Other functions incorporated in the system include load balancing, fault-detection, and fault-recovery algorithms.

VI. Status and Results

PRAIS is currently being developed under the auspices of the University of Texas at Arlington. At the time of this writing the knowledge-based system shell has been modified for real-time processing with portions of the dynamic scheduling implemented. The distributed communication mechanisms have been implemented via sockets, but is being transferred to TCP/UDP datagram RPC's for hardware transparency (via XDR) and to accommodate an unlimited number of processors. Aspects of the system that are either not implemented or are untested include those for fault recovery recovery and insufficient memory detection. It is hoped to port the system to a supercomputer as funding becomes available.

Metrics that have been used to evaluate the system include thruput, efficiency, message load, and "quality" of reasoning which has been interrupted. As the actual immediate application is real-time sensor processing, sensor inputs missed is also an extremely important. Finally, as the the system's goal is cost-effective parallel processing (via cycle-stealing from underutilized processors transparently to the programmer), programmability and maintainability are two highly desired qualities. These qualities are estimated by "system observability", or how easily a person can understand the system. A mathematical representation for system observability is currently being developed, and will hopefully be presented at next year's International Joint Conference on Artificial Intelligence.

As of the time of this writing, the application on which this research has been tested is a high-fidelity simulation featuring many parameters which themselves be connected to sensors or even higher fidelity simulations. The speedup was such that, while efficiency was not as high as hoped, the thruput provided surpassed the benchmark (running on a multi-processing mini-supercomputer) was exceeded by five Sun workstations by over 50%.

VII. Conclusions

PRAIS offers a variety of features including:

- heterogeneous hardware capability,
- real time control via dynamic saliences, and
- incorporating 'C', 'C++', FORTRAN, and Ada, with productions.

PRAIS strives to lessen the user's effort needed to build a parallel, real time KBS by incorporating many knowledge-base system functions as extensions to the operating system.

Actual knowledge-based systems often fail simply because of cost; a more cost-effective approach for developing parallel, real time knowledge-based systems is crucial for bringing AI to the "real world". Systems requiring AI are typically very complex, so sophisticated tools providing simpler solutions must be used to reduce programming costs by .

References

[CG88d] C. Culbert and J. Giarrantano, CLIPS Reference Manual Version 4.2, Artificial Intelligence Section Lyndon B. Johnson Space Center, Houston, Texas, April 1988.

[GL] A. Gupta and T. Laffey, Real-Time Knowledge-Based Systems, Tutorial: MA3, Eleventh International Joint Conference on Artificial Intelligence, Detroit, 1989.

[Kit] J. Kitfield, Military Forum, National Journal, Inc., Washington, D.C., July 1989, pp. 28-35.

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

omit

B11 Session: Enhancements to CLIPS – Object Oriented

Integrating an Object System into CLIPS: Language Design and Implementation Issues

Mark Auburn

Inference Corporation
5300 W. Century Blvd.
Los Angeles, CA 90045

Abstract

This paper describes the reasons why an object system with integrated pattern-matching and object-oriented programming facilities is desirable for CLIPS, and how it is possible to integrate such a system into CLIPS while maintaining the run-time performance and the low memory usage for which CLIPS is known. The requirements for an object system in CLIPS that includes object-oriented programming and integrated pattern-matching are discussed, and various techniques for optimizing the object system and its integration with the pattern-matcher are presented.

1. Introduction

As CLIPS, and CLIPS-like production systems, gain widespread usage and acceptance, and as the number of CLIPS applications increases, the limitations of the main CLIPS data representation, the fact, become more evident. Although facts, and the n-ary relations they represent, are a powerful and flexible method for representing arbitrary relationships between data, the lack of explicit relationships between individual facts and their lack of internal structure inhibit the representation of large, complex knowledge bases.

Object representations, such as embodied in the object-oriented programming languages of Smalltalk, CLOS and C++, and in the experimental languages KL-ONE, are a natural data extension to CLIPS's facts. Object-oriented programming languages that include the capability of pattern-matching on objects represent a combination of two separate lines of research: research on representing objects and representing the actions associated with those objects, and research on the most efficient general methods of matching on data. It is apparent that both of these lines have matured, in the form of efficient commercial object-oriented programming languages (e.g. Classic-Ada [8]) and efficient commercial production systems.

In the first section, the specific advantages of an object system will be discussed, followed by a presentation of what requirements are necessary for an object system that would maximally increase the utility of CLIPS programming and the various tools built around the basic production system component of CLIPS.

These issues will be illustrated using the example of ART-IM (Automated Reasoning Tool for Information Management) [5], a tool from Inference Corporation for development of expert systems, which shares a common syntax and many implementation strategies with CLIPS, and may be logically viewed as an extension of CLIPS.

In the second section, issues of object system implementation are examined, concentrating on the integration into CLIPS's pattern and join networks necessary to achieve the desired efficiency of pattern-matching. Although it is possible to match against an object's slots and values just as is done for facts, the nature of an object system allows for an additional degree of optimization based on knowledge of the object hierarchy and assumptions about the rate of change of various parts of the hierarchy. Just as assumptions about the frequency of working-memory change lead the implementation of a fact pattern-matcher to use the Rete algorithm, assumptions about the usage of the object system lead to additional optimization techniques. This paper discusses those assumptions and several of the techniques used by the ART-IM object system to reduce object system overhead.

Finally, some future directions for object system enhancement are sketched.

2. Language Design

2.1. Advantages of an Object System

Although fact-based data storage and retrieval, including fact-based pattern matching, provides a wide range of desirable functionality for the developer of expert systems, there remain many expert system applications whose data representation cannot be adequately represented in facts. The working-memory model, made popular by OPS5 [1] and implemented as facts in CLIPS, implicitly subdivides and flattens data down to a level comparable to a database record or a record in a conventional programming language. However, there are many problems such as classification and diagnosis for which an inheritance hierarchy is both closer to a natural understanding of the domain and more economical in expressing data. Although an inheritance hierarchy does not expand the class of possible applications beyond that of the working-memory model, in many cases it can provide a more natural, economical and maintainable representation. An object system offers the following advantages over a working-memory model:

- An explicit hierarchy.
- Explicit inheritance (along with the ability to override it).
- Explicit internal structure that can be declaratively described.
- Easier to maintain, since it corresponds better to the user's model.

There are, of course, disadvantages. Typically object systems, in exchange for these advantages, require more memory and more processing time than an equivalent fact representation. However, due to the decreased maintenance cost of a more explicit representation, the total software lifetime cost may be lower.

Once an object representation is in place, it is also possible to enhance the inheritance hierarchy with procedural attributes to achieve object-oriented programming. Although rules can be used to duplicate any procedural activity, it is often simpler, in cases where the control flow is predefined, to write procedural code. Procedural code will typically be faster than an equivalent rule version, since the overhead for control flow determination implicit in a rule implementation lacks. Object-oriented programming can be used to achieve some of the same goals of rule-based programming, in that by increasing the locality between data and the operations on that data the ease of maintenance is increased.

An object data representation also offers a finer granularity of update recalculation over the working-memory model in that a data change can be performed, and pattern-matching updated, on a change to an object's slot value, rather than only on the assertion or retraction of an entire fact. In large applications this can have a significant impact on performance.

2.2. Requirements for an Object System in CLIPS

The utility of an object system for CLIPS depends directly on the degree of integration with CLIPS, and its subsidiary features, achieved by the object system. The main requirement, of course, is that it integrates with the pattern-matcher. Object patterns must be provided that offer the same sophisticated pattern-matching available to fact patterns.

The object patterns need to be able to:

- Test for the existence of an object.
- Test the class membership of an object.
- Test for the existence of a specific attribute on an object.
- Test for the values of a specific attribute on an object.

Binding variables to various attributes and values, and comparing those variables to other attributes and values in the same object, and to other variables bound in other object and fact patterns, is also an important consideration.

The object system needs to be completely dynamic, as with facts, and to enjoy a full procedural interface for changes during execution. Object-oriented programming, while perhaps not a necessity given the availability of the powerful rules of CLIPS, is certainly desirable. Essential to the programming ease of the object system is full integration into all debugging features and into all programming utilities, such as those for verification and validation, truth maintenance and explanation generators.

ART-IM, as an example CLIPS extension, provides an integrated object system with inheritance and three types of links: subclass, class member and user-defined relations. The attributes of the objects are defined using the object system itself, and they and their values are inherited by children nodes. Object-oriented programming is also provided and consists of attaching methods to atttributes of the appropriate object. The ART-IM object system is also integrated with ART-IM's explanation-generation subsystem and with its justification-based truth-maintenance system.

3. Implementation

Although the features provided by an object system are desirable, it is clear that in a production system designed for speed and low memory usage like CLIPS an inefficient implementation of the object system features would severely restrict the usage of the object system. In particular, without the deep integration between the object hierarchy and the pattern-matcher, such as exists between the fact database and the pattern-matcher, the efficiency of rules that matched on objects would be much less than that of those rules that matched on facts, and therefore of little use in a real-world CLIPS application.

ART-IM incorporates a variety of implementation techniques to increase the efficiency of the object system, and some of these techniques are discussed below. It is possible, in some cases, for the efficiency of matching on objects to exceed the efficiency of matching on equivalent facts, using these implementation techniques.

In particular, three techniques for optimization are discussed below:

- Representing class membership with the use of bit vectors.

- Canonicalizing attribute order.

- Precomputing valid object patterns for a particular segment of the object hierarchy.

The second technique, although useful for reducing the storage requirements of a large and multilayered object base, is crucial to ensuring the success of the third and is primarily useful in that context.

This paper will not touch on the various techniques for optimizing *method selection on objects* in object-oriented programming. In general, since pattern-matching is the most important constraint in most CLIPS applications and in most production systems, the integration with pattern-matching is viewed as the most important efficiency topic.

3.1. Representing Inheritance Information

Since the test for class membership is performed often in an object system (and replaces the fact equivalent of testing for a particular value in a particular position on a fact), optimizing this test would appear to yield significant benefits.

There are at least two commonly used methods for deciding which classes an object belongs to:

- Explicitly passing class information down from each class to all of its children.

- Requiring the system to search upward from an object to its immediate parents, repeating the search until all of the parent classes have been discovered.

The processing time for such class membership determination is conserved in the first, while storage space is conserved in the second. Due to multiple inheritance and deep inheritance hierarchies, the first method can become prohibitively expensive in terms of space when implemented by representing class membership by attribute values. On the other hand, searching upward from an object to all of its classes can consume large amounts of processing time, especially if the results of the search are not cached for future use.

A technique used in ART-IM to reduce the space consumption of the first method while preserving its fast class comparison test is that of encoding inheritance chains into bit vectors. Encoding the class structure of each object into a binary vector has two desirable properties: it consumes little space (in ART-IM, one byte per ancestor link), and the test of whether or not an object belongs to a specific class is reduced to the quick test of whether or not a binary value is contained as a prefix in the vector of the object.

Of course, the encoding of inheritance values costs processing time, but the cost of the processing is on the same order as that of directly passing class information as attribute values down to the object's children, and the space consumption is approximately an order of magnitude less. The membership test itself is again only slightly more complex than the search for a particular attribute value.

3.2. Canonicalization of Attribute Combinations

A typical implementation for a fully dynamic object system (one that allows the creation and destruction of all classes, subclasses and class members, along with the creation and destruction of object attributes, during execution) of the attributes of objects is as a linked list. As attributes are added to an object, or deleted, they are inserted into or removed from the object's attribute list. In order to add or substract values from an attribute, it is necessary to search the list looking for the attribute, and then insert the value into the value list of that particular attribute.

The advantages of this representation are:

- The implementation is straightforward.
- Dynamic addition and deletion of attributes is a simple list operation.

The disadvantages are:

- Inserting or deleting a value requires a full search of the attribute linked list.
- Each attribute requires at least two words of memory, no matter how static the inheritance hierarchy is.

The linked list representation is certainly the most efficient implementation when attributes are dynamically added and deleted to objects with a high frequency. However, as the frequency of attribute changes decreases, the most efficient representation converges on an implementation which is the analog of a structure (or record) in a conventional programming language: a contiguous segment of memory with implicit positioning of attributes.

In order to allocate contiguous segments of memory (erasing the need for the link field and the attribute name per attribute), and still allow for dynamic changes, it is necessary to create a parallel data structure which represents the attribute combinations present in the object system. By creating a canonical ordering for all attributes in the system, the space consumed by this parallel structure can be reduced.

As objects are created, their attributes are sorted into canonical ordering. The attributes are then stored in an array that does not include either a link field or the name of the attribute itself. In order to determine which element of the array belongs to which attribute, a pointer is attached to the object which points at a parallel attribute-combination hierarchy. Each node in this hierarchy contains a specific combination of attributes, and the growth of the hierarchy is dependent on the canonical order of the attributes contained in each node. This hierarchy is more efficient than representing the attributes directly in the objects because many objects will share specific attribute combinations, but requires some additional time for attribute lookup. However, the time for attribute lookup can also be less than the list implementation, depending on the hardware, as an array lookup is often implemented in hardware, whereas a list lookup is not.

This canonical ordering of slots is also an essential prerequisite to the pattern precompilation technique discussed in the following sections, which further reduces the cost of matching the attributes of an object to the attributes required by a particular pattern.

3.3. Pattern Matching Technology for Record Data Types

Production systems, the software tools that have refined the technology of pattern-matching the farthest, have traditionally used either simple variables or records as their data representation. Data types called "working memory elements", which are similar to the records of data bases or traditional programming languages, have been used most frequently in systems such as OPS5. Efficient algorithms for pattern-matching on these working memory elements have been developed, including Rete [2] and TREAT [6]. Variants on these algorithms, in particular for parallel machines [3] [4], have been designed, and comparisons have been performed [7]. These algorithms, however, have typically only been tested and designed for the working-memory model.

These algorithms make several assumptions:

- That the set of patterns to match on is constant.
- That the knowledge base (the collection of working memory elements) is large.
- That the change in the knowledge base over the interval of time between each match is small.

The goal of these algorithms is to reduce the time required for deriving the matches by storing partial results for the matches, and updating the partial results as the knowledge base changes. Otherwise, the N times M comparison necessary for full derivation of the matches of a set of patterns, where N is the number of knowledge base items and M is the number of patterns, is far too computationally expensive to obtain whenever the matches are desired.

In a pattern that consists of references to several working memory elements, for example, the Rete algorithm will store two types of data for all matches: pointers to all working memory elements that match an individual reference in the pattern (a condition), and partial matches for successive subsets of the conditions in the entire pattern. As changes in the knowledge base occur, they are percolated down to a network created by the Rete algorithm which determines how to update the stored partial results based on the changes. Since the time required for obtaining the matches is dependent only on the number of changes in the knowledge base since the last pattern-matching point and the number of patterns which are affected by those changes, and not on the total number of patterns or knowledge base objects, it typically reduces the pattern-matching time by a significant factor.

As the form of data representation has migrated from records, in the form of working memory elements, to objects as the representation of choice, due to their economy of representation (from inheritance) and flexibility, the Rete and TREAT algorithms were adapted in a straightforward manner to match on objects. Objects and their attributes and values were transformed into object-attribute-value triplets, and these triplets handled exactly like simple working memory elements. As objects changed, modified triplets were sent to the pattern-matcher for updates. Although this method for object integration is straightforward and allows for the reuse of code developed for fact pattern-matching, it does not exploit the wide range of optimization possibilities inherently present in an object system. The following two sections discuss some of the features available for optimization in the object system, and one technique for exploiting some of these features.

However, since comparing bound variables across various objects allows for the same implementation as the identical comparison in the fact pattern-matcher, that comparison will not be discussed in this paper. Object systems do not present additional problems or opportunities in the inter-condition comparison, as opposed to the intra-condition case.

3.4. Object System Features Relevant for Pattern Matching

As in the case of knowledge bases constructed using working memory elements, it is possible to construct a set of assumptions about object-based knowledge bases in addition to the assumptions stated above:

- That each object may have a large set of different attributes.
- That each pattern may refer to a limited group of attributes of an object.

- That the inheritance hierarchy changes slowly, if at all.

- That many objects will be instances of classes, as opposed to representations of subclasses.

Like all assumptions, these may be violated in any particular application, but should hold in general. Based on those assumptions, it would seem desirable to implement pattern-matching on an object system such that:

- Matching on an instance of a class is highly efficient, even if the set of instances and their values change relatively rapidly.

- Each pattern need only inspect those attributes of an object that are used in the match.

- Inheritance and class information is incorporated as much as possible, given that patterns may refer to that information and that it changes slowly.

These assumptions form the basis for the next section, which describes a particular method for utilizing these apparent features. However, it is important to note that there exist many different methods for exploiting these assumptions, just as with working-memory element pattern-matchers, and that the one described below is only one of several possibilities.

3.5. An Inheritance Hierarchy for Pattern Matching Correlations

Once the pattern and join networks (or alpha and beta nodes, to use the terminology of [2]) for a set of fact patterns have been created, the process of matching a new fact to the existing patterns is described by testing the fact against the entire set of application patterns, and producing matches for those patterns which the fact successfully matched against.

Using the features of the object system described in a previous section, it is possible to reduce the size of the set of patterns considered in the matching process. By using structural characteristics of the patterns (such as which classes they address or the attributes they contain), it is possible to substantially reduce the set of patterns considered, which depending on the cost of examining each pattern for applicability can reduce the processing time required for pattern-matching considerably.

Once the parallel attribute-combination hierarchy described in an earlier section has been created for an object system, each pattern is attached to exactly one node in that attribute-combination hierarchy. Each pattern is attached to that attribute-combination node which contains exactly those attributes used in the pattern. As objects are created, then, in addition to the cost of searching for the appropriate attribute-combination node, pattern-matching information is attached to the object, derived from the nodes in the attribute-combination hierarchy that the object traverses. The pattern-matching information will apply to that class and to its subclasses. Attaching pattern-matching information to the object hierarchy, and updating it as the hierarchy and the objects contained it change, does impose overhead on changes to the object system. Based on the assumptions above, the relative infrequency of changes to the object hierarchy will compensate for the expense of those changes.

When pattern-matching occurs, preselection of those objects that are relevant to a pattern has already been accomplished, so that patterns that couldn't fulfill a particular object (e.g., they belong to a different class or do not contain the attributes required by the pattern) are not considered in the pattern-matching process. For class instances, in particular, this can bring a substantial performance improvement, as they need only use the pattern-matching information of their class in deriving the appropriate patterns. The repetitive class membership tests and the attribute presence tests required in patterns can be performed once, for the class, and amortized over the entire set of class instances.

4. Conclusions

This paper has presented several reasons for integrating an object system into CLIPS, as well as some techniques for optimizing that integration. The optimization techniques, although implemented for a production system, are applicable to other object-based processing methodologies that use pattern-matching.

There are other ideas that have not been implemented but deserve active consideration.

It would be quite desirable to introduce the capability to partition the knowledge base, and indeed individual attributes on objects, into items appropriate for pattern-matching and items upon which pattern-matching will not be performed. Since pattern-matching imposes an overhead on objects and their attributes, reducing this overhead by confining it to specified areas could greatly improve efficiency. In addition, developing protocols for passing information between a pattern-matcher and an object system that are independent on the object used, or indeed on the implementation of the pattern-matcher, would be of interest. This would allow the creation of object-oriented data bases with integrated pattern-matching, with the advantage of efficient storage of large number of objects on disk.

Taking such a protocol and enhancing it for distributed communications would present the interesting possibility of distributed expert systems communicating through a general object metaprotocol, as well as allowing for a flexible, transparent external data interface that would communicate with data from such diverse sources as databases, windowing interfaces and process monitors.

Allowing type and value restrictions on object attribute values, and being able to specify an internal structure for those values, is also a desirable addition.

References

1. Brownston, L., Farrell, R., Kant, E., Martin, N.. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley, 1985.

2. Forgy, C.L. "RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem". *Artificial Intelligence 19* (1982).

3. Gupta, A.. *Parallelism in Production Systems.* Pitman Publishing, 1987.

4. Gupta A. et. al. Results of Parallel Implementation of OPS5 on the Encore Multiprocessor. CMU-CS-87-146, Carnegie-Mellon University, Department of Computer Science, August, 1987.

5. Inference Corporation. *ART-IM/MS-DOS 1.5 Reference Manual.* Inference Corporation, 1988.

6. Miranker, D.P. TREAT: A New and Efficient Algorithm for AI Production Systems. Phd thesis, Columbia University, 1987.

7. Schor, M.I., Daly, T.P., Lee, H.S., Tibbitts, B.R. Advances in Rete Pattern Matching. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1986.

8. Software Productivity Solutions, Inc. *Classic-Ada User Manual.* Software Productivity Solutions, Inc, 1988.

CLIPS Enhanced with Objects, Backward Chaining, and Explanation Facilities.

M. ALDROBI, S. ANASTASIADIS, B. KHALIFE, K. KONTOGIANNIS, R. De MORI

McGill University, School of Computer Science, 3480 University St.
Montreal, Canada, H3A 2A7
demori@cs.mcgill.ca

Abstract

In this project we extend CLIPS, an existing Expert System shell, by creating three new options. Specifically, first we create a compatible with CLIPS environment that allows for defining **objects and object hierarchies**, second we provide means to implement **backward chaining** in a pure forward chaining environment, and finally we give some simple **explanation facilities** for the derivations the system has made. Objects and object hierarchies are extended so that facts can be automatically inferred, and placed in the fact base. Backward chaining is implemented by creating run time data structures which hold the derivation process allowing for a depth first search. The backward chaining mechanism works not only with ground facts, but also creates bindings for every query that involves variables, and returns the truth value of such a query as well as the relevent variable bindings. Finally, the **WHY** and **HOW** explanation facilities allow for a complete examination of the derivation process, the rules triggered, and the bindings created. The entire system is integrated with the original CLIPS code, and all of its routines can be invoked as normal CLIPS commands.

1. INTRODUCTION.

The C Language Production System (CLIPS) is an expert system tool written in and fully integrated with the C language. It provides high portability, and easy integration with external systems, making embedded applications easy. The primary representation methodology is forward chaining based on the Rete algorithm. A.I. methodologies not provided in CLIPS are the organization of seperate data into hierarchies which exhibit inheritance, the backward chaining inference strategy, and facilities to justify the reasoning process and the conclusions derived.

In [1] object oriented systems are discussed as one of the most promissing paradigms for the design, construction, and maintenance of large scale systems. This general model for computing has major applications in A.I. (e.g. [2, 3, 4]). Moreover, in [1] techniques such as deligation [5, 6], genericity [7, 8], conformance [7], enhancement [7], and inheritance [5, 6, 7, 8] are thought to be the basis of "object-related systems". The object-oriented system embedded onto CLIPS gives the capability to the user for defining objects using a frame-like structure, and allows the flow of information between objects by invoking methods. The above object-configuration was adopted to facilitate encapsulation, inheritance, and set-based abstraction, which are main characteristics of these systems [1, 9].

Furtheremore, production rules in CLIPS are not trigerred using a backward chaining inference mechanism. In [10] backward chaining is exhibited as an inference strategy that verifies or denies one particular conclusion or hypothesis. In [11] this mechanism is initiated by establishing a goal and then is matched with a conclusion of a production rule. This subgoal is substituted by a sequence of subgoals which are the premises of the relevent rule. The entire process terminates when all subgoals are proven to be true. Backward chaining is used in many applications such as diagnosis, decision making, and trouble shooting, and simplifies the explanation facilities [12, 13]. In our extension of CLIPS we use forward chaining to implement bakward chaining by creating data structures and traversing the structures in order to obtain a simulation of the mechanism.

Finally, the development of a "complete" shell requires to enhance the environment with informative explanations [14]. These facilities have recently been approached and many solutions have been proposed. In [15] clarity is the focus for the structure of an explanation, in [16] a proof-tree is created, while in [17] one creates a model suited for specific users. On the other hand, [18, 19] stress that the content of the explanations is of more importance than the form in terms of providing meta-rules that descibe the expert strategic knowledge. But, one of the most difficult problems in the explanation domain is to answer negative questions concerning facts that were not inferred by the shell [20, 21]. Our approach is to create a well-defined semantic structure containing the knowledge derived and the derivation process followed. This approach guarantees that the same explanation will be given for the same question [14].

In this paper we present a compatible with CLIPS environment allowing for defining objects as well as establishing hierarchies between objects, a backward chaining inference mechanism capable of performing bindings in queries involving variables, and finally two explanation facilities , WHY and HOW. The paper is organized in six sections. The first section deals with objects and object hierarchies where we present the object definition, the hierarchy schema adopted, and how attributes along with their values are inherited. In the second section the query language used to interrogate the objects is presented. In the third section we describe the backward chaining mechanism embedded in the expert system shell, as well as the data structures, and routines that implement it. In the fourth section the explanation facilities WHY and HOW are presented. Specifically, we investigate the data structures created during run time, and the mechanism involved for traversing the list in order to provide answers to the WHY and HOW facilities. In the fifth section we present a list of the new commands implemented, and what actions are taken accordingly for each command. Finally, we conclude by reviewing our work, identifying extensions we are working on, and exploring potential applications in the field of diagnosis, and troubleshooting.

It should be noted that in the first three sections some implementation details concerning algorithms used on the data structures are mentioned.

2. OBJECTS - OBJECT HIERARCHIES.

In an expert system shell such as CLIPS a necessity arises to construct a well defined hierarchical network of entities that will support user-system interaction resulting in a **structured KB**. These entities are objects that can be inherited via a network to other objects that reside lower in the hierarchy. Moreover we maintain a common syntax for facts and we use the hierarchy and the inheritance in order to create new facts and update the knowledge base in an efficient, well structured, and meaningful way.

2.1 OBJECTS.

The implementation represents an object as a record with the following fields :

- object name.
- object parent.
- object children.
- inheritance type[0] ... inheritance type[MaxAttributes].
- object type.
- attribute name[0] ... attribute name[MaxAttributes].
- attribute value[0] ... attribute value[MaxAttributes].
- comment.

The **object name** defines the name the user gives for the object which is unique in the entire hierarchy. The **object parent** is the parent of the object in the hierarchy, the **object children** points to a linked list containing all the children of the object, the **inheritance type** is either own or member (which will be explained shortly), and the **object type** is one of class, subclass, and instance. Finally, one has for each object a list of **attribute**

name value pairs which identify the characteristics of each object (they are limited to MaxAttributes), and a simple field for a **comment** is allocated for any special note about the object that must be known.

The **hierarchical network** is a set of objects distributed among three layers according to the semantic meaning of each object. The first layer contains objects of type class (the most general type of object), the second layer contains sublayers of objects of type subclass (the next least genaral type of object), and finally one has a layer of objects of type instance (the least general among all types of objects). See Fig. 1.

Inheritance is built in the network as a flow of information from objects with abstract semantic context to objects with specific semantic context. In this hierarchical network attributes, and their corresponding values are inherited from classes to subclasses, from subclasses to other subclasses, and from classes and subclasses down to instances. If the inheritance type is **member**, the flow of inheritance is not interrupted, while if the inheritance type is **own**, the values are not inherated and overwrite all other inherited values. It should be noted that each attribute name value pair for each object has a different inheritance type. The default type is member. In such a way our hierarchical network can be thought of as a set of oriented trees, where the roots are the corresponding classes.

In this schema the ideal implementation is a forest of trees, where the roots are classes, internal nodes are subclasses, and leaf nodes are instances. Also it is possible for nodes from one tree to have a parent or children in an other tree, interleaving the trees resulting in a complex forest structure. The data structure used in order to preserve all the properties, and the inheritance among the objects is to maintain n-ary tree structures for every class definition created, such that for every class maintain pointers that will allow traversals to move only down, for each subclass maintain pointers that will allow the traversal of a tree to move up or down, and for each instance maintain pointers that will allow traversals to move only up.

Inheritance alters the contents of the Knowledge base and the patterns we use to accomplish such a goal. The major observation here is that CLIPS handles and manipulates facts as strings and matching is done using string manipulation functions. With this observation in mind we restricted our facts to have a particular pattern for describing an attribute and its corresponding value as follows :

The [*attribute*] of [*object*] is [*value*].

which can be asserted directly as a CLIPS fact.

Moreover we use another pattern for all children of a class or a subclass. These patterns are :

All [*subclass*] are [*class*].

All [*instance*] are [*class*].

All [*subclass*] are [*subclass*].

All [*instance*] are [*subclass*].

All the above patterns create a complete set of facts, since the patterns encapsulate the information described by the attributes and the connections between the objects.

In such a way traversing a hierarchical network we can create facts that do not originate from the user, but can be inferred by the hierarchy. This has two advantages.

- First , the user specifies only the attributes absolutely necessary for an object assuming that all other attributes higher in the hierarchy are available.
- Second , we minimize the information stored in every object without losing any information.

Hence,the user specifies the world, and the system creates the relevant facts.

The final use of the pre-determined patterns is that knowing their syntax we can reserve positions for (single or multiple) bindings in rules or facts in forward or backward chaining. For example we know that a question :

The ?x of car is red

is a meaningful query and that the query

The color of car ?x red

is not a meaningful one.

It should be mentioned that the inheritance type controls the assertion of facts since, own attribute values participate in the generation of new facts, and overwrite all other inherited values for the same attribute. All inserted facts become immediately available to the rules, and participate equally in the derivation process.

2.2 AN EXAMPLE OF OBJECT HIERARCHIES.

Define the objects to be : Car (class), PrivateCars (subclass), Porsche (instance), BMW (instance).

Assign inheritance type **own** to : Porsche, and PrivateCars, for attribute name Color.

Assign inheritance type **member** to : Car, and BMW, for attribute name Color.

Define the connections to be : Porsche is an instance of PrivateCars, BMW is an instance of PrivateCars, and PrivateCars is a subclass of class Cars.

Assing the Color Red for Porsche, the Color Blue to PrivateCars, and the Color white to Cars.

The following facts are inserted in the **KB** of **CLIPS** :

- All Porsche are PrivateCars.
- All Porsche are Cars.
- All BMW are PrivateCars.
- All BMW are Cars.
- All PrivateCars are Cars.
- The Color of Porsche is Red.
- The Color of BMW is White.
- The Color of PrivateCars is Blue.
- The Color of Cars is White.

See Fig. 2 for details.

3. QUERY LANGUAGE.

Here we give a description of the query language applicable to the hierarchy network. This query language provides the means for obtaining information regarding the entries found in the network. Specifically we have the following possible queries :

a) (**Display?** [*object type*])

returns the description of all objects of the specified object-type

b) (**IdType?** [*object type*])

returns the object names given the type

c) (**GenType?** [*object name*])

returns the parent of the specified object

d) (**SpecType?** [*object name*])

returns all children of a specified object

e) (**GetAttribList?** [*object name*])

returns all attributes and their values an object may have.
This query takes care of own values and discrards member values for same attributes.

f) (**GetAttribValue?** [*object name*] [*attribute name*]) returns the corresponding value else returns false.

g) (**IsAttribValue?** [*object name*] [*attribute name*] [*attribute value*])

returns true or false

h) (**JustObj?** [*object name*])

returns the comments added for this object

Operations e,f,g,h take into account the inheritance that exists in the network.

The overall network constructed operates under CLIPS control, updating the KB, supplying the user with mechanisms for viewing the status of the system, and hence controlling the derivations that CLIPS produces as a result of applying ground facts to rules using forward or backward chaining.

4. BACKWARD CHAINING MECHANISM.

In this section we present a mechanism to implement **backward chaining** within the framework of the CLIPS shell. The aim is to provide means for analyzing an initial goal (query) into a set of subgoals each of which has to be solved using this method, up to the time the set of subgoals will contain only ground facts known to be true. The way we treat the set of subgoals implies that all subgoals in the set must be recursively satisfied in order for the initial goal to be true. For rules that have premise in disjunctive normal form we create a "set of subgoals" for each disjunction and every subgoal from each "set" must be proven true in order to have a successful derivation.

The user supplies a query and the system tries to match this query with an existing known true fact. If no such fact can be found then the rule(s) which has as its RHS this fact is considered and its premises are recursively considered as the new goals. The whole derivation process ends when all relevant rules have been examined and tested. Because we are interested not only in ground queries, but also in queries with variables, we use CLIPS's binding mechanism so that the appropriate bindings can be made.

In order to simulate the backward chaining mechanism we create a **Backward Chaining Network (BCN)** consisting of instantiated conclusions and facts interconnected as shown in Fig. 3.

This approach involves **four major steps**.

- In the first step we insert into the BCN all ground facts.
- The second step is to invoke Forward Chaining and add the derivations into the BCN as well.

- The third step provides a method of traversing this data structure of linked lists so that it implements the depth first search strategy. The way the linked lists are structured and traversed simulates the desired backward chaining.
- The final step is to create a dynamic data structure so that we can keep the derivation steps meaningfuly grouped in order to be used for the explanation facilities later on.

4.1 CREATING THE NETWORK (BCN).

The primary method of representing knowledge in CLIPS is rules of the form

IF [*PREMISE 1*] or

[*PREMISE 2*] or

....

[*PREMISE n*]

THEN

[*ACTION 1*] and
[*ACTION 2*] and
....
[*ACTION m*]

where each [*PREMISE i*] group could be a conjuctive expression combining different fact patterns, not necessarily ground facts, of the form

[*Fact 1*] and [*Fact 2*] .... and [*Fact k*]

and each [*ACTION i*] be of the form [*Asserted Fact i*]

Specifically, when a rule is fired we get a set of ground facts related in the following format :

[*Rulename*] - [*Asserted Fact i*] - [*Fact 1*] and ..... and [*Fact k*]

Moreover the **logical combinations** between entries in LHS of a rule as well as in the RHS of the rule are treated as follows :

1. If there are more than one OR related [*PREMISE*] in the LHS of a rule then we create a format as indicated above for each [*PREMISE*] expression.

2. If there are more than one AND related [*ACTION*] in the RHS of a rule then we create a format as indicated above for each [*ACTION*] expression.

3. For all other logical combinations involving (1) and (2) we create as many formats as can be derived when (1) and (2) are simultaneously applied (e.g. one format for each

[*PREMISE*] and [*ACTION*] combination).

These operations are equivalent to splitting the rules involving complex disjunctions and conjunctions into an equivalent set of rules of the form :

IF [*Fact 1*] and [*Fact 2*] and ... [*Fact m*] **THEN** [*Asserted Fact*]

Backward chaining is implemented using Forward Chaining in **two major steps**. In the **first step** we create a network of data structures in order to capture the relation between ground facts, premises, and conclusions in the Knowledge base, with the substitutions computed during Forward Chaining .In the **second step** we traverse the network so that we can find all possible derivation paths and bindings for a particular query. The way we traverse the network simulates a depth first search strategy. During the Forward Chaining derivation process we are creating our data structures using the following strategy :

For each rule we keep track of the premises and the conclusions that participated in each derivation step along with the Rule Name as Forward Chaining proceeds. For each conclusion reached we create a node pointing:

a) to the *next* Conclusion derived from Forward Chaining, and

b) to a list of rules that support this conclusion.

Each such rule node points to:

i) the next rule used and

ii) to a linked list representing the premises in conjunctive form.

In the case of premise groups which are combined disjunctively we maintain a different rule node pointing to a group of Premises which contains Facts in a conjunctive form.

See Fig. 3 for details.

4.2 TRAVERSING THE NETWORK (BCN).

After creating the network, the way we **traverse** it is implemented using a depth first search strategy with recursion. Specifically, the user specifies a query,which may contain variables, and the system tries to find legal bindings for the variables in order to prove or disprove the query. This is a two step process:

a) The first step is to create , if possible, legal bindings scanning all conclusions in the rule network, and, if found, generate the first goal.

b) The second step is to invoke a function in order to implement the desired backtracking. This is done using the BACKCHAIN(goal) function which is a recursive function. Specifically, if the goal is immediately derivable as a ground fact the function returns the bindings and the query has succeeded, otherwise finds the first premise that supports the current goal, sets the premise as the current goal and is re-invoked recursively. The result of the recursive execution is the creation of a derivation path which will be later used by the explanation facilities HOW and WHY. This derivation path forms a branch in the search tree so it is represented as a linked list of facts. It is possible that more than one derivation path exists so we keep them in different branches in a data structure as in Fig. 4. The process ends when there are no more conclusions to be tested in the BCN for possible legal bindings. The bindings are computed using a word by word comparison between two strings representing the ground term and the query. The notation for a variable is *?variable-name* and all words are tested with the words of same position in the instantiated ground term.

This process will return the correct answer as well as the relevant bindings because one is working in a subset of the Knowledge base that has been created using Forward Chaining. This new space is simply integrated, organized, and traversed in a way that simulates Backward Chaining.

5. JUSTIFICATION AND EXPLANATION.

The ability of expert systems to give explanations of their results and of the reasoning leading to those results is considered as one of the main advantages of these systems, as compared to usual programs. In rule based expert systems, explanations are often confined to the trace of the program execution. A trace is a record of fired rules. It may also include the data which allows these firings, cast into some readable form, preferably in a natural language. In some approaches, a distinction is made between WHY and HOW explanations.

All these types of explanations rely on the notion of trace. It seems that explanations produced depend heavily on the way the expert knowledge was encoded into rules. Often, explanations are more reminiscent of the language provided by the expert system shell rather than of the language employed by the domain expert.

WHY queries provide explanations on a conclusion that has been derived. Specifically, they allow for a quick reference on both the rule that supports the particular conclusion, and on the premises in the rule that supports this conclusion.

HOW queries provide explanations for the whole derivation The difference between the HOW and WHY facilities is that, WHY lists and gives information on the last rule triggered and HOW lists all possible derivation paths, rules, and bindings that suport the conclusion .

5.1 WHY QUESTIONS.

WHY questions are implemented using the data structure illustrated in Fig. 3. In this structure, which is implemented using linked lists we have three node categories : "CONCLUSION" nodes, "RULE" nodes, and " PREMISE" nodes. A "CONCLUSION" node contains a particular ground derivation obtained, and points to the rules that support it. Each "RULE" node contains the rule name and points to a linked list of "PREMISE" nodes. This structure allows for storing all the groups of premises that triggered the rule. Note that when two or more groups of premises are combined with OR's it may be the case that both groups may have contributed in the derivation. In such a case we keep them in separate lists under different "RULE" nodes having the same rule name.

Consider the following rule (CLIPS syntax) :

```
( defrule Rule1 (or ((p1) (p2)) ((p3) (p4) (p5)))
    ⇒ ((assert ( c1)) (assert (c2))))
```

According to this rule if all the premises p1,p2,p3,p4,p5 are satisfied we will have two "RULE" nodes and five "PREMISE" nodes linked as follows :

One "CONCLUSION" node for c1 pointing to "RULE" node (Rule1) and to the next "CONCLUSION" node c2. Rule1 node points to an identical node Rule1 since we have two groups of ORed premises, and to a linked list of premises, consisting of p1,p2. The other "RULE" node points to null and to a linked list of premises representing the ORed second group of premises, p3,p4,and p5 (See Fig. 5).

Finally the second conclusion (i.e. c2) points to an identical , as above , structure .

Actually we will have as many answers to such questions as the number of times the corresponding rules where fired. Referring to Fig. 3, each ANDed group of premises going vertically forms one answer and we have as many answers as the number of these vertical groups going horizontally. These different answers are grouped by an OR in Fig. 3.

The way we implement a WHY [*FACT*] query for a specific conclusion is as follows. First we search for a particular "CONCLUSION" node that matches the query, then we traverse the relevant linked lists for every "RULE" node and every group of premises, and print in a user friendly format all the premises encountered, as well as all relevant rules names. (refer to Fig. 3).

5.2 HOW QUERIES.

Every time a **HOW query** is asked we compute all the possible derivation paths through which this conclusion had been derived. Thus, a derivation path is equivlent to a branch in the search tree. The computation is carried out in a recursive way using both the structure which implements the WHY questions and a new data structure, "DERIVATION PATHS", which is illustrated in Fig. 4. This data structure consists of two node types. "BRANCH" nodes and "FACT" nodes. We create it every time a HOW question is asked and destroy it thereafter. The answer to a HOW question is computed as follows :

When a **HOW** [*CONCLUSION*] **query** is asked , our goal becomes the "CONCLUSION" we want to satisfy, so we refer to the BCN to find the corresponding conclusion node. If we could NOT find any, because this "CONCLUSION" is neither derived nor a ground fact, then we return false. If the corresponding node is found we consider it as the current goal and we pick up the first premise from the BCN. This premise becomes our new goal and we repeat the same operation until there are no more premises in every premise group considered. We create a data structure of linked lists as in Fig. 4. "FACT" nodes represent the backward chaining derivation process, and the "BRANCH" nodes represent derivations performed in different premise groups.

Consider the following example :

(defrule Rule2 (or (d) ((a) (b)))
⇒ (assert (y)))

where (d),(a),and (b) are assumed to be ground facts for simplicity.

According to this rule and these facts, if we ask the question:

(HOW ?y)

then the possible branches are :

branch 1 : (y) (d);
branch 2 : (y) (a) (b);

6. NEW COMMANDS.

1. **(OBJECT)** : Creates interactively a new object and places it in the hierarchy network. Also one has the ability to query the hierarchy.

2. **(QUERY)** : Starts interactively backward chaining for a user specified query. It creates bindings and returns the corresponding truth value of the query.

3. **(HOW)** : Returns all possible path derivations for a specific query and explains how the particular subgoals were established and proved.

4. **(WHY)** : Returns information on the rule that proves a particular query and explains the truth values of the corresponding premises.

7. CONCLUSION.

In this paper we presented an extension of the CLIPS Expert System shell. We have created an enhanced version by allowing Objects and Hierarchies to be defined, adding a Backward Chaining mechanism for triggering rules, and finally creating two basic explanation facilities WHY and HOW. The whole system is fully integrated in the original CLIPS environment. The new version is currently running on a SUN 4 machine. Future extensions will be available in a DOS environment as well, so that maximum flexibility and portability can be obtained. Special care is taken so that the interface for the new commands is user friendly and much attention was paid on error checking and reporting.

Currently, we are integrating methods for objects. Methods will be defined as an attribute of an object and will have the same inheritance properties as any other attribute of the hierarchy. The internal structure of a method will be identical to a normal C function, and accessing attribute values will be acomplished by implementing two functions available to all methods that will get an attribute value given an object name and attribute name, and put an attribute value given an object name and attribute name (see Fig. 6).

Furthermore, we are integrating explanation facilities to answer questions of the form "WHY a conclusion was not derived ?", and "WHY a rule was not fired ?". The basis for answering these questions is to incorporate the closed-world assumption for the current status of our knowledge base.

Finally, we are implementing a user friendly interface in the form of a natural language system in order for a user to input definitions of rules, facts, objects, methods, and a menu driven system in order for the user to access all the commands that the new version of CLIPS supports.

These extensions are currently tried under a SUN 4 and a NeXT machine environment.

References

[1] Blair G., Gallagher J., Malik J., " Genericity vs Inheritance vs Delegation vs Conformance vs .. ". *Journal of Object Oriented Programming*, Sept./Oct. 1989 Vol.2 No. 3.

[2] Stefik M., Brobrow D. G., " Object Oriented Programming : Themes and Variations". *The AI Magazine*, 1985, pp.40 - 62.

[3] Goldberg A. and Robson D. " Smalltalk 80 : The Language and its Implementation", Addison Wesley, 1983.

[4] Morris J. H., Meyer B. , Nerson J., Matsuo M., " Eiffel : Object Oriented Design for Software Engineering *In*, Proceedings of the First European Software Engineering Conference, Strasbourg, France, Sept. 1987, pp. 120 - 124.

[5] Lieberman H., "Delegation and Inheritance : Two Mechanisms for sharing Knowledge in Object Oriented Systems". *Journees Languages Orientes Objet*, 1985, pp. 79 - 89.

[6] Stein L. A., " Delegation is Inheritance ", *Special Issue of SIGPLAN Notices*, Orlando, FL. Oct. 4 - 8, 1987, 22 (12), pp. 138 - 146.

[7] Horn C. " Conformance, Genericity, Inheritance and Enhancement", *In*: Proc. ECOOP, Paris, June 1987.

[8] Meyer B. " Genericity versus Inheritance", *In* : Proceedings of OOPSLA 1986, Conference, pp. 391 - 405, Portland, OR, Sept. 1986.

[9] Taenzer D., Ganti M., Podar S. " Object Oriented Software Reuse : The Yoyo Problem", *Journal of Object Oriented Programming*, Sept./Oct. 1989 Vol.2 No. 3.

[10] Jackson P. "Introduction To Expert Systems", Addison Wesley, 1986.

[11] Winston P. "Artificial Intelligence", Addison Wesley, 1984.

[12] Waterman A. D. " A Guide to Expert Systems", Addison Wesley, 1984.

[13] Buchanan B., Shortliffe E. " Rule Based Expert Systems ", Mc Graw Hill.

[14] Millet C., Gilloux M. " A Study of the Knowledge Required for Explanation in Expert Systems ". *1989 IEEE Fifth Conference on Artificial Intelligence Applications.*

[15] Weiner J., "BLAH, a System which explains its reasoning", *Artificial Intelligence* 15 (1 - 2) pp. 19 - 48, 1980.

[16] Erickson A. "Neat Explanation of Proof Trees". *Proc. of the 9th IJCAI*, Los Angeles, Ca, 1985.

[17] Forsyth R. "Expert Systems Principles and Case Studies". Chapman and Mall Publ. Co.

[18] Hasling D. W. "Abstarct Exlanations of Strategy on a Diagnostic Consultation System". *Proc. of the National Conference on Artificial Intelligence* , Washington DC, pp. 157 - 161, 1983.

[19] Clancey W. J., "Transfer of Rule Based Expertise through a tutorial dialog". Stanford University, Dpt. of Computer Science, 1979.

[20] Krekels X., " Why-not Explanations in Expert Systems and their Use as a Debugging Tool", *Cognitiva 85*, 1985.

[21] Safar B., Rousset M - C., "Negative and Positive Explanations in Expert Systems". Tech. Rep. LRI, Univ. d'Orsay, 1985.

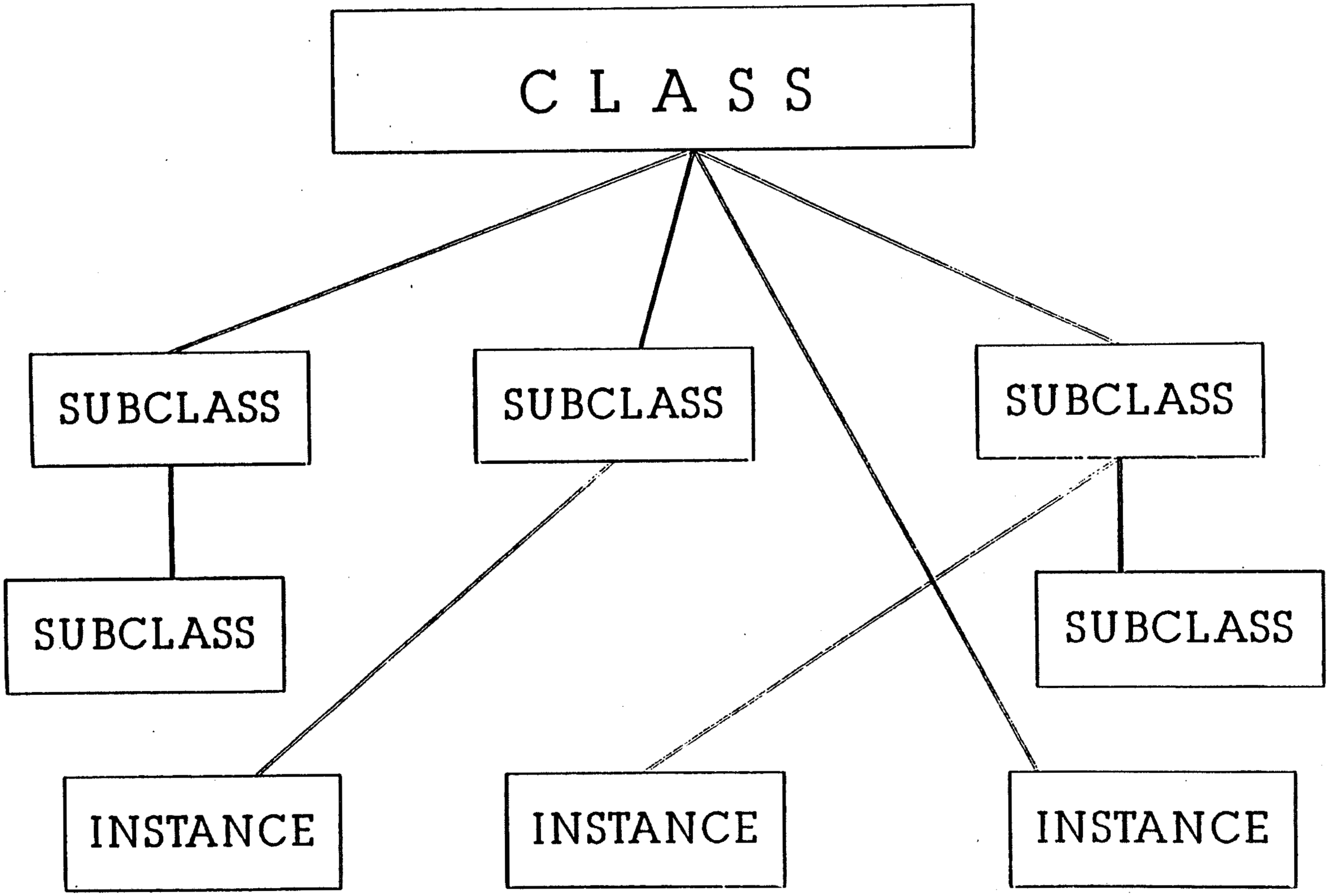


FIGURE #1 : An Object Hierarchy.

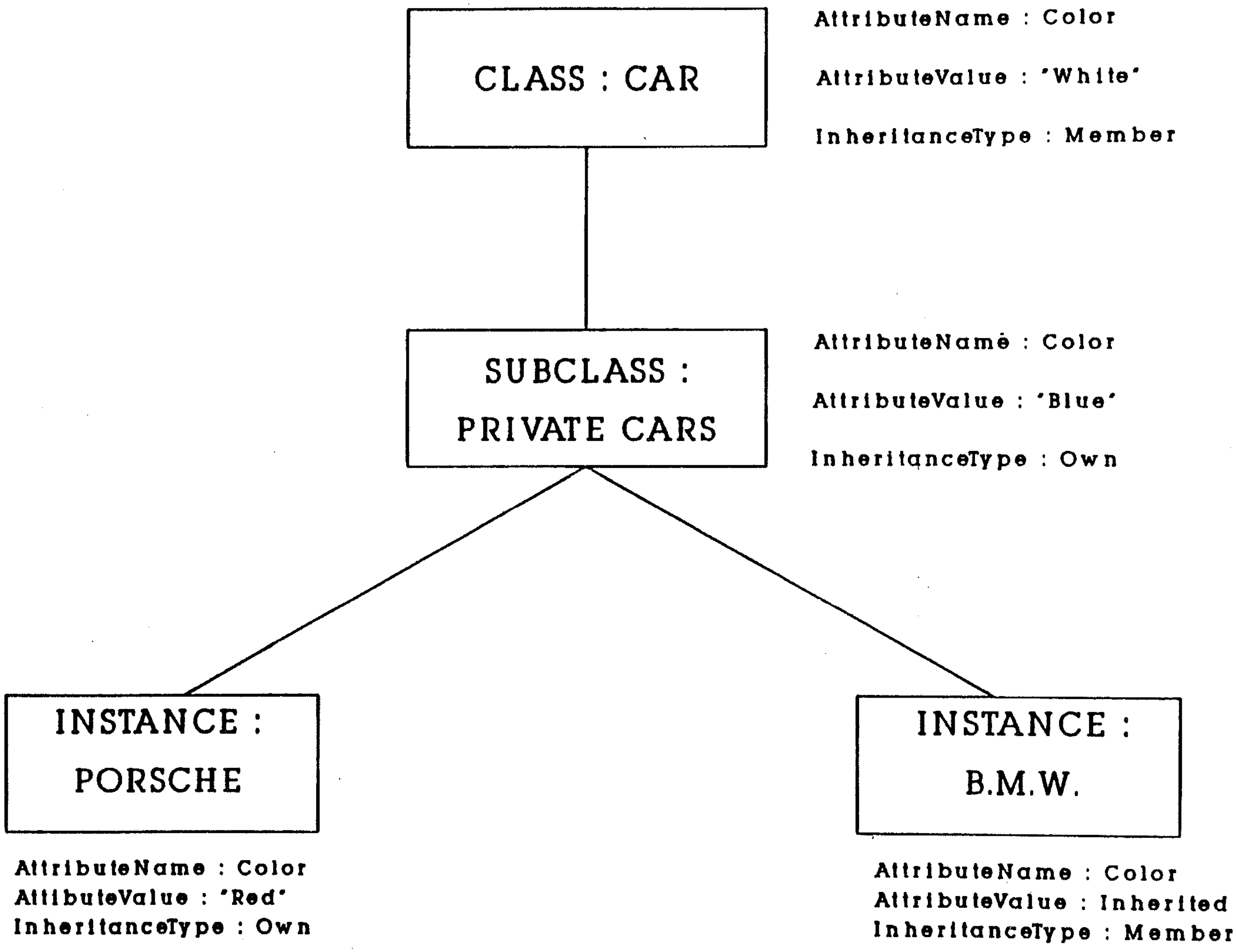


FIGURE #2 : An Example Of Object Hierarchies

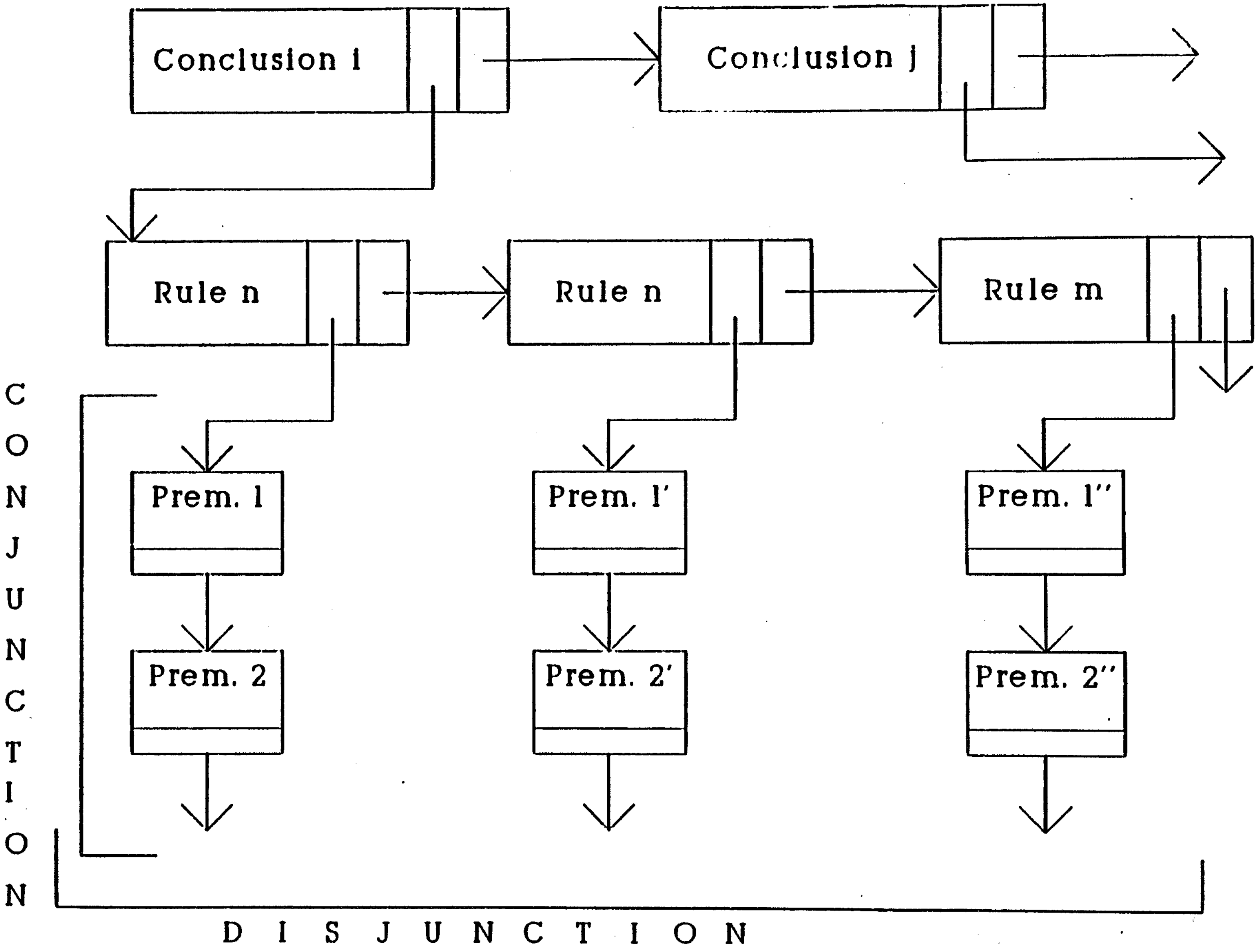


FIGURE # 3 : Backward Chaining Network.

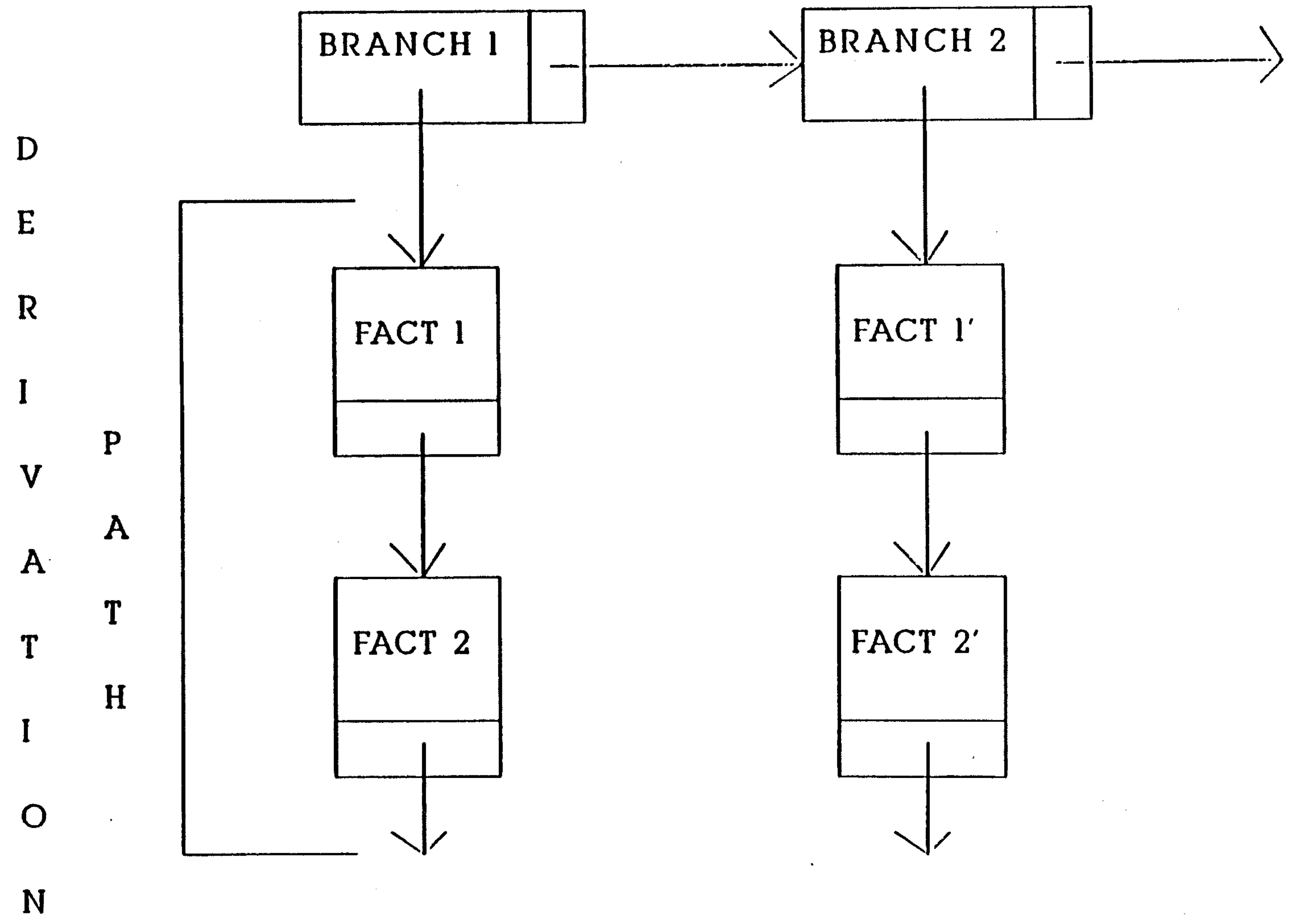


FIGURE # 4 : Derivation Paths.

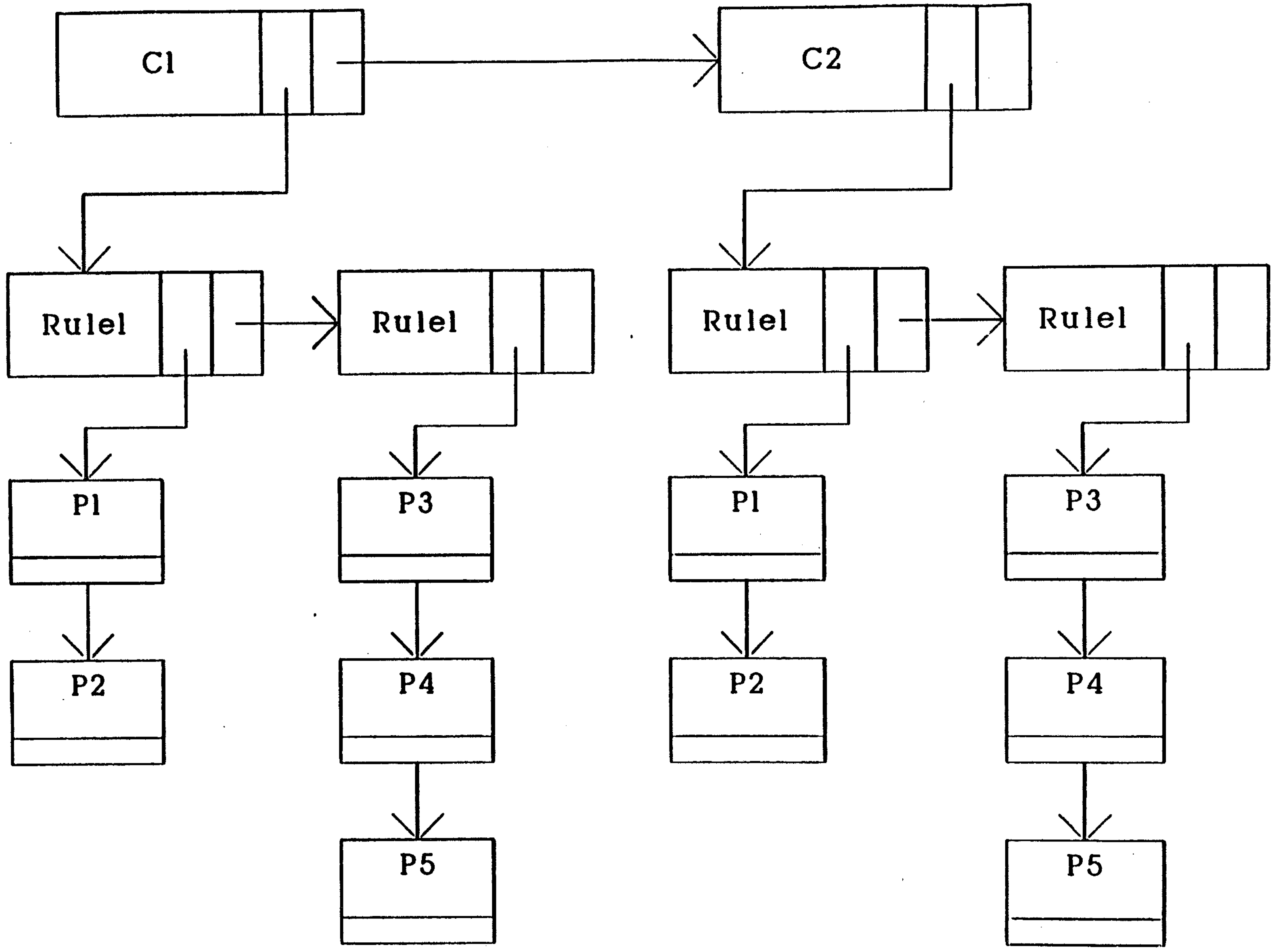
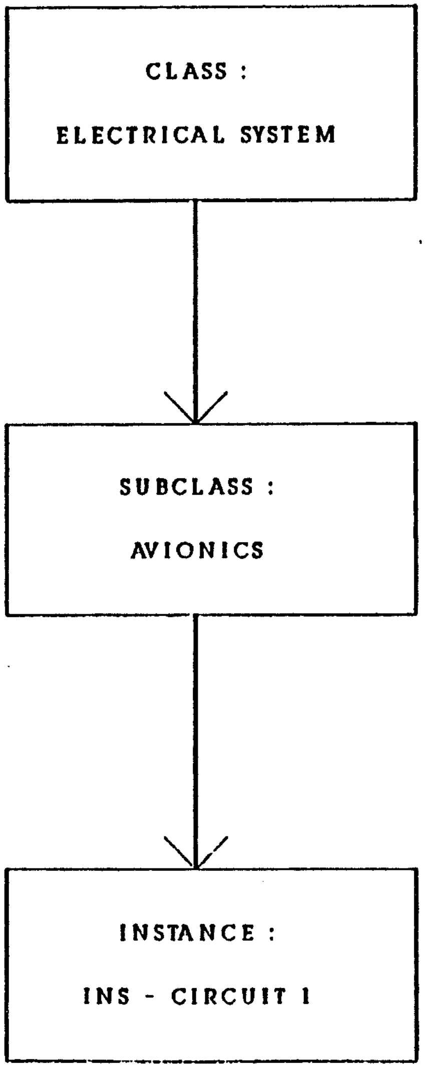


FIGURE # 5 : An Example BCN.



AttributeName : Current
AttributeType : Numerical
AttributeValue : Unknown
InheritanceType : Member

AttributeName : Resistance
AttributeType : Numerical
AttributeValue : Unknown
InheritanceType : Member

AttributeName : Power
AttributeType : Method

AttributeValue : POWER
InheritanceType : Member

```
(DefMethod POWER(ObjectName)

float Power(char ObjectName)

{

float I = getValue(ObjectName, Current)

float R = getValue(ObjectName, Resistance)

return I * I * R

}
```

(Inherited From Above)

(Inherited From Above)

FIGURE # 6 : An Example Method Implementation

Integration of Object-Oriented Knowledge Representation with the CLIPS Rule Based System

David S. Logie and Hasan Kamil
Structural Analysis Technologies, Inc. (SAT)
4677 Old Ironsides Dr. Suite 250
Santa Clara, CA 95054
(408) 496-1120

Abstract

The paper describes a portion of the work aimed at developing an integrated, knowledge based environment for the development of engineering-oriented applications. An Object Representation Language (ORL) was implemented in C++ [2] which is used to build and modify an object-oriented knowledge base. The ORL was designed in such a way so as to be easily integrated with other representation schemes that could effectively reason with the object base. Specifically, the integration of the ORL with the rule based system CLIPS [1], developed at the NASA Johnson Space Center, will be discussed.

The object-oriented knowledge representation provides a natural means of representing problem data as a collection of related objects. Objects are comprised of descriptive *properties* and inter-*relationships*. The object-oriented model promotes efficient handling of the problem data by allowing knowledge to be encapsulated in objects. Data is inherited through an object network via the relationship links. Together, the two schemes complement each other in that the object-oriented approach efficiently handles problem data while the rule based knowledge is used to simulate the reasoning process. Alone, the object based knowledge is little more than an object-oriented data storage scheme; however, the CLIPS inference engine adds the mechanism to directly and automatically reason with that knowledge. In this hybrid scheme, the expert system dynamically queries for data and can modify the object base with complete access to all the functionality of the ORL from rules.

Introduction

This project was undertaken because of the need for a practical environment for the development of large expert systems, specifically, those involving engineering domains. In general, the motivation for this work can be summarized in the following:

- □ the limited expressiveness of rule-based knowledge representation, especially in engineering domains,
- □ the inability to build large, efficient, and comprehensive expert systems consisting of thousands of rules,
- □ the need to effectively store knowledge (i.e. acquired from the user, data bases, or inferred by a rule set) for later use, and
- □ the desire to have a common environment that could link expert systems with existing data bases and procedural programs.

Even in the preliminary stages of the development of an expert system for structural/mechanical design [3], we realized that a system with a minimum of usefulness could be comprised of thousands of rules. This fact introduced some concerns with respect to hardware and software limitations and the practicality of maintaining such an extensive knowledge base. One of the most powerful uses of this enhancement is the ability to chain rule sets. A large set of rules can be decomposed into smaller sets which reason about specific subproblems. For example, a rule could state that **if** a certain piece of knowledge is *unknown* **then** load another rule set that will infer that data. The original rule set can put itself in queue to return and continue processing, transparent to the user. Also, previously autonomous expert systems can now share data through common objects and communicate with each other through the ORL queries. As illustrated in Figure 1, a very large network of rule sets can be developed giving the illusion of a large expert system when, in fact, only a small set of rules are being processed at any one time. This capability becomes especially important on a personal or desktop computer platform. Developing, modifying, updating and verifying knowledge bases for large applications is a less formidable task when small rule sets can be edited and tested independent of the entire application.

Another advantage realized from this enhancement is that rule sets shrink considerably. This is primarily because rules for handling user queries and checking user responses are now handled by the ORL. Rule sets need only contain rules for ORL queries, the actual problem solving rules and those rules that report the results[1] . An existing set of rules can easily be modified to take advantage of the ORL capabilities[2].

Disk storage of knowledge has proven to be very useful also. In our scheme, a rule set is invoked in the context of a project. Objects are first searched for in a project specific location and then in a global storage area. In a run-time environment, modifications to the object base are only specific to a particular project. This context sensitivity allows the user to examine the effect of various responses on the recommendations or findings of an expert system by simply changing contexts. For example, in a medical diagnosis system the context would be set to refer to a particular patient.

The ultimate intention of this effort is to develop a fully integrated environment in which the same ORL query initiated from a rule can not only query the user but also result in a query to an existing data base or the invoking of a procedural program [4]. The details of where the information should be retrieved would be specified as the object base is developed through the use of property metaslots (discussed later). Optimally, this integration should be seamless to the user and function efficiently in a networked environment. With this capability, ORL/CLIPS applications could have limitless potential for practical use.

For the remainder of the paper, the use of the ORL and the object-oriented knowledge representation scheme to build practical expert systems is discussed and demonstrated.

Use of the ORL

The ORL consists of a concise set of functions for building and maintaining an object base. One of the main goals in the

[1] A generic reporting mechanism is being developed that may eliminate the need for the latter type of rule, thus, leaving only the rules specifically for reasoning.

[2] Existing CLIPS rules will still run without modification.

development was to keep the use of the ORL as simple as possible so that engineers or experts in other domains, without extensive computer programming experience, could develop knowledge bases and, furthermore, that non-experts could easily utilize the resulting expert systems.

The type of commands available include those for file operations, building and displaying classes and objects, querying and asserting property values, editing the object base, and an interface to the usual CLIPS command line. The file operations allow the user to set the current project, save and load objects to and from disk, reset memory resident object properties to *unknown* or to clear memory completely. Note that when running a rule set, objects are automatically loaded as needed but must be saved explicitly to permanently store any changes made by the rules.

Command line functions for building and modifying the object base include making classes and objects, making an instance of a class, copying objects, or adding and removing properties and relationships. Menu-oriented editors are available for specific modifications such as changing the name or type of a property or defining metaslots.

To access ORL commands from a rule the developer uses the "ORL" function as the first item in the right hand side pattern. The remainder of the pattern is precisely the ORL command line function and arguments. For example, to save an object to disk from a rule, one would write:

```
( ORL save < object name> ).
```

Just as in CLIPS, several destructive functions are disallowed from within a rule.

Classes and Objects

Classes and objects are the basic structures of the knowledge representation scheme. They contain descriptive properties and relationships to other classes and objects. When a property value is required in a rule set, the class or object must be queried for that specific property's value(s). Queries to classes and objects only differ in that a query to a class results in all the instances of that class being queried. In general, an ORL query from a rule takes the form:

```
( ORL get <class/object name> <property name(s)> )
```

and results in asserted facts of the form:

```
( <object name> <property name> <value> { certainty}[3] ).
```

Qualifiers for the queries such as less-than, greater-than, or equal-to need to be implemented for fully functional querying; however, these types of tests are currently available in CLIPS which accounts for their low priority in the development.

In the same way, permanent assertions to the object base take the form:

```
( ORL assert <object name> <property name> <value> {certainty} )
```

and result in the a fact:

```
( <object name> <property name> <value> {certainty} )
```

Other queries return the instances of a class or related parts of an object. For example, to find out the instances of a class the query would be:

```
( ORL getinstances <class name> )
```

and would return facts as:

```
( <class name> instance <object name> )
```

which could be matched on the left hand side of a rule for deleting instances of a class.

Properties and Metaslots

Properties (often called 'Attributes' in similar schemes) are the mechanism by which classes and objects are described. They simply hold one or more values as they are asserted. Currently, a property may be of type integer, float, text, or boolean. Other specialized property types are being developed such as filename, equation, data

[3] For brevity, certainty factors will not be discussed, however, properties may optionally have a certainty applied from 1-100%.

base, and program. A property will automatically handle the checking of user responses and build the appropriate CLIPS facts as values are assigned.

Defining a metaslot for a property adds a considerable amount of versatility. First, a metaslot can be used to put constraints on the values that a property can hold by specifying a list of allowable values or a range of numeric values. Other useful features include assigning initial and default values for the property and defining the prompt displayed to the the user.

Possibly, the most powerful feature of a metaslot is the ability to define a search strategy with the "Order of Sources." The USER is the default source for information when a property value is queried. Alternatively, the knowledge base developer may wish the property to assume the initial value when queried for the first time or the default value if the user responds *unknown* to a query. Also, it may be desirable to query an existing data base or invoke a procedural program to generate data.[4] These facilities may lessen the need for user interaction when the level of knowledge of user may be in question or may make it easier to develop autonomous expert systems for applications such as robotics.

Relationships

Relationships allow properties to be inherited by related classes and objects. The most common types are the *instance* and *instance_of* relationships between a class and its instance. When an instance of a class is created, the relationships between them are automatically created so that the new object can inherit properties in the class hierarchy. Other types include *is_a* and *subclass* relationships between classes (e.g., **Jet** *is_an* **Airplane, Airplane** has *subclass* **Jet**) and *part_of* and *subobject* relationships between objects (e.g., **wing-x** is *part_of* **airplane-y, airplane-y** has *subobject* **wing-x**).

As mentioned earlier, the relationships come into play when the classes and objects are queried. If a class is queried for a property value, it will automatically pass the query on to its instances. Similarly, if an object is queried for a property value which it doesn't have, it may pass the query on to related objects according to the

[4] These latter capabilities are currently under development.

current inheritance protocol. The relationship capability promotes efficient handling of data by eliminating unnecessary redundancy.

Example

The example automotive diagnosis system that was distributed with CLIPS will be used for the purpose of demonstration. First, compare the rules for querying the user for the working state of the engine. With CLIPS alone, the rule was:

```
(defrule determine-engine-state
        ?rem <- (query phase)
        (not (working-state engine ?) )
=>
        (retract ?rem)
        (printout t "What is the working state of the engine:" t)
        (printout t "  (normal/unsatisfactory/does-not-start)? ")
        (bind ?response (read) )
        (assert (working-state engine ?response) )
)
```

The user must type the complete response, correctly. Using the ORL, an object, engine, is created with the property, working-state, having a metaslot that defines the allowable values and prompt as above. The new rule is:

```
(defrule determine-engine-state
        ?rem <- (query phase)
=>
        (retract ?rem)
        (ORL get engine working-state)
)
```

The new query to the user is:

```
What is the working state of the engine?
        1. normal
        2. unsatisfactory
        3. does not start
        4. unknown or other

Selection:
```

Even this small rule set was reduced by two pages of text and several rules. Note that the original rule set made no provision for incorrectly typed responses or any other error checking. In the second case, the user ***cannot*** make a typing error and provisions were made for *unknown* responses. It was found that rules modified to employ ORL objects and functions tend to read more naturally so that they can be more easily debugged or updated.

To fully utilize the ORL in building a useful automotive diagnosis system, a developer would define a class, auto, with related parts such as the engine or doors and then have the rule set instantiate these classes for a specific case. Also, the expert system could be divided into modules for specific problem areas such as the engine or transmission containing the expertise of specialized mechanics. Modularity makes an expert system more easily extendable. This approach is being used in-house at SAT in the development of rule sets for designing structural/mechanical components. With this approach, it was possible to develop rule modules containing basic knowledge ourselves and then consult experts in specialized areas to extend the capabilities of the knowledge based system or tailor it to a specific engineering problem.

Conclusions

The salient features of the ORL were discussed, including typical functions employed in the development and use of this object-oriented/rule-based knowledge representation scheme. The object-oriented paradigm is especially expressive in representing static, structured knowledge. The simple example of the automotive diagnosis system showed that the size of a CLIPS rule set can be significantly reduced using the ORL. Accompanying the reduction in size is improved efficiency and built-in error handling. Context sensitivity and permanent (data base like) disk storage promote flexibility in developing knowledge bases. The ultimate aim of this work will result in an integrated environment able to access data in distributed data bases and invoke procedural programs through a common user interface or from expert systems. With these capabilities there is no limit to the size of knowledge bases that can be built or the range of applicable domains to which such an integrated system could be applied.

References

1. CLIPS User's Manual, Version 4.3, Artificial Intelligence Section, Johnson Space Center, NASA, 1989.

2. Stroustrup, Bjarne, The C++ Programming Language, 1987.

3. Kamil, H., Vaish, A. K., and Berke, L., "An Expert System for Integrated Design of Aerospace Structures," Fourth International Conference on Application of Artificial Intelligence in Engineering (AIENG89), Cambridge, England, July, 1989.

4. Kamil, H. and Logie, D.S., "Toward an Integrated, Knowledge Based Engineering Environment," International Symposium on Artificial Intelligence, Robotics and Automation in Space (I-SAIRAS), Kobe, Japan, November, 1990.

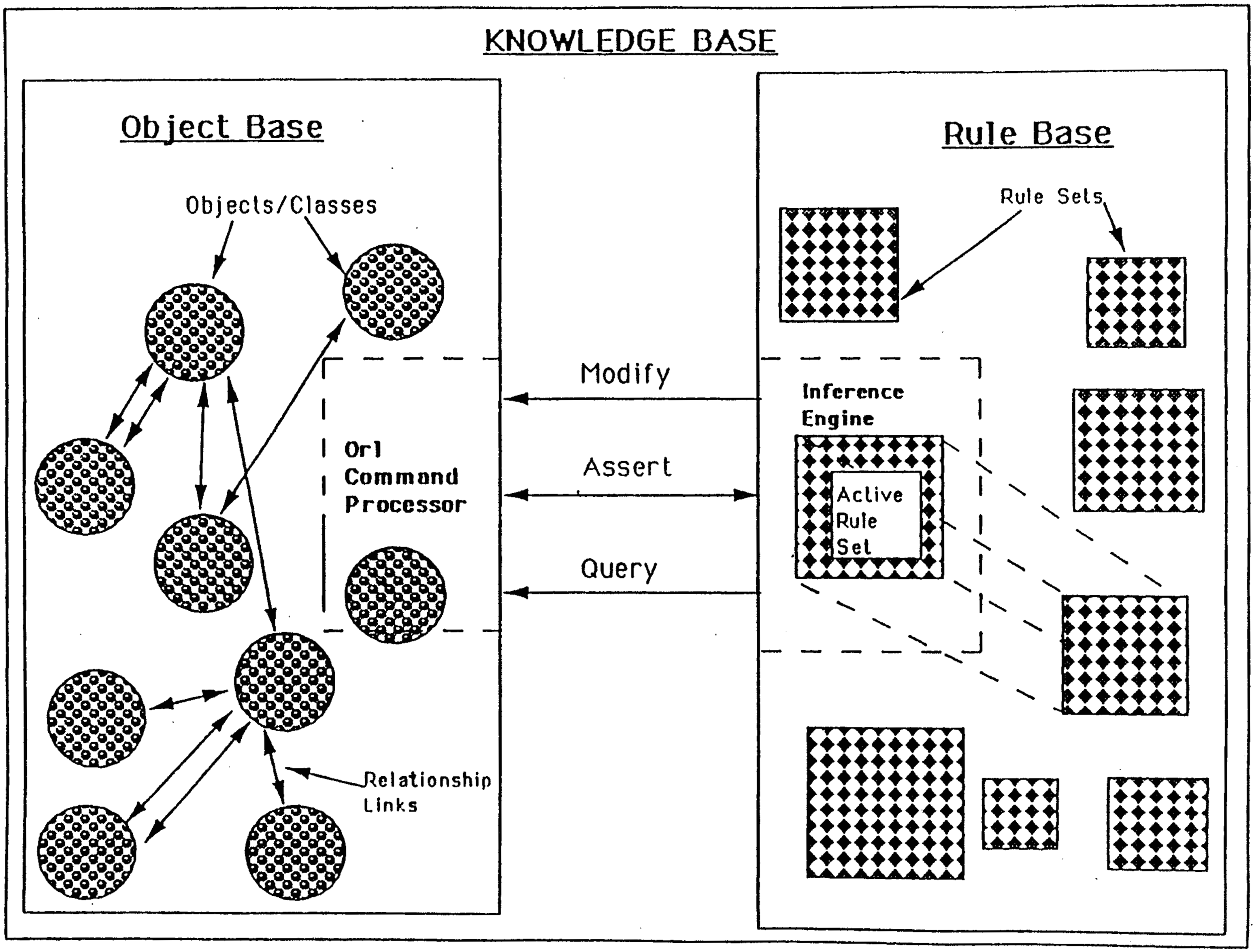


FIGURE 1. OBJECT-ORIENTED/RULE-BASED KNOWLEDGE REPRESENTATION SCHEME

527-61

N96- 12940

p. 9

358609

An Object Oriented Extension to CLIPS

Clifford Sobkowicz

Government of Canada, Dept. of the Environment
McGill University, School of Computer Science

April 25, 1990

Abstract

A presentation of a software sub-system developed to augment CLIPS with facilities for object oriented knowledge representation. Functions are provided to define classes, instantiate objects, access attributes, and assert object related facts.

This extension is implemented via the CLIPS user function interface and does not require modification of any CLIPS code. It does rely on internal CLIPS functions for memory management and symbol representation.

1 Introduction.

CLIPS (C Language Integrated Production System) is an expert system shell which represents knowledge by production rules which can be applied to asserted facts. Rules represent constant knowledge of relationships between antecedents and consequents, such as causes and effects. Facts specify current information and are either asserted initially, interactively, or as the consequents of rules.

Objects are abstractions of knowledge about hypothetical entities. They are represented as sets of attributes, which can take numeric or symbolic values, and methods for for manipulating them. Objects are members, or instances, of classes with common sets of attributes and methods. Each instance has specific values for the attributes associated with its class. As attributes and methods are qualified by specific objects polymorphism is provided for, whereby actions upon objects can be affected by different means depending on the class of the object.

The capabilities presented here provide for extending rule consequents to include object manipulation and allow for antecedents based on objects and their attributes as well as asserted facts. Also, assertion of facts about objects is facilitated.

2 CLIPS rules and facts.

Rules in CLIPS are composed of a set of antecedents termed the left hand side (LHS) and a set of consequents termed the right hand side (RHS). Facts are ordered sets of fields which can assume single word, numeric, or quoted character string values. The antecedents of rules are patterns which are matched against the current set of facts. They may include wildcard fields, variables which are bound to one or more field values from matching facts, and logical expressions for constraining field values. The consequents are actions such as asserting subsequent facts or side effects such as outputting messages and variable values.

An important feature of CLIPS is the facility for invoking external functions on either the LHS or the RHS of a rule. On the LHS functions can provide data for pattern expansion or be implemented as predicate functions to constrain fields or test conditions. On the RHS functions can perform side effects as consequents of rules. It is via these facilities that objects are created, manipulated and accessed.

3 CLIPS Objects.

Objects in this sub-system are sets of named attributes which can take word, numeric or string values. They can also take multiple values. Attributes are specified by an object name and an attribute name. The data type of an attribute is set dynamically.

Objects include methods which can manipulate the attributes. These are C functions which are integrated with CLIPS, like other external functions, via the usrfuncs routine. Methods are invoked by specifying the name of an object and a method selector in an invoke command, along with any parameters to be passed to the function. Also, functions can be attached to attributes and invoked automatically when the value is set or read. Different functions can be invoked from different objects by the same attribute name.

3.1 Implementation.

Functions such as creating objects and accessing their attributes, are implemented by external functions called from CLIPS rules.

Objects are designated by names which are CLIPS symbols and reside in the CLIPS symbol table. They are identified by hash pointers so string manipulation is averted. Objects are implemented as structures in a second table structured after the symbol table. The randomized bucket number from the symbol table entry is used in the object table so as to speed searching. The location of the last object referenced and the last object modified are retained, so performance can be optimized by grouping commands which reference the same object. The object structure includes the hash pointer of the name, a pointer to a list of attributes and a pointer to an inheritance list.

The attributes are stored in a linked list of structures which indicate the type of the attribute: class, instance, or method, the type of the current data: word, number, string, or multiple, the data itself: a pointer or floating point value, the attached functions: two pointers into the CLIPS function table, and some information related to inheritance.

Multiple field data is stored as a linked list referenced from an attribute value element.

4 Classes.

Classes specify sets of attributes and methods common to groups of objects referred to as instances of the class. They are represented by objects that are used as templates for instantiation. Attributes are either class or instance attributes. Class attribute values are maintained in the class object and are common to all instances of the class.

5 Inheritance.

Classes can inherit the attributes and methods of other classes. Thus general super classes can share their functionality with more specific sub-classes which can add additional functionality. Multiple inheritance is provided for in that a class can inherit from a number of classes allowing functionality from general utility objects to be mixed in with super class and local resources. Inherited classes can include inherited resources themselves to unlimited depth. Circular inheritance is disallowed.

A priority can be specified for each inheritance to resolve conflicts when the same name appears in more than one contributing class. The inheriting class carries priority 100 so that inheritances with priority less than or equal to 100 preserve the original resources while those with priority greater than 100 can replace them.

As an alternative to conflict resolution, methods can be declared as multiple in a class definitions. An object can then inherit a list of procedures under a single method name. All of the procedures will be called in sequence when the method name is invoked. It is the responsibility of these procedures to limit their results to non-conflicting side effects such as asserting facts or updating separate attributes or to implement a combining algorithm such as summing or appending results.

5.1 Implementation.

Each class includes an ordered list of inherited classes. The order is that in which the inheritances where specified. The list entries reference the inherited object and indicate the priority of the inheritance. Inherited class objects can inherit classes so the composition of a class is a tree structured set of class objects.

6 Instances.

Instances of classes represent specific objects by maintaining specific data in instance attributes. They inherit all attributes and methods of the class of which they are an instance.

6.1 Implementation.

When a class object is instantiated a new instance object is created. All of the attributes of the class object are copied to the new instance, and assigned a priority of 100. The tree of class objects inherited by the specified class is traversed depth first. For each attribute encountered, if the attribute is not yet present in the new instance it is copied and assigned the priority of the inheritance. If the attribute is already present but the priority of the inheritance is higher than the priority recorded in the instance it is overwritten.

In the event of equal priority, precedence is given to inherited objects according to the order in which the inheritance was specified and class objects are considered to include their inherited attributes.

When instance attributes are copied to the instance object the value in the class object is copied along with pointers to any when-read and when-set functions. Thus the attribute in the class object serves to provide an initial value and attached functions.

When class attributes are encountered a pointer is placed in the instance object which refers back to the class. Thus the data and attached functions remain common to all instances of the class.

A Appendix: CLIPS commands.

The following are illustrations of CLIPS statements which create and manipulate objects. Familiarity with CLIPS as documented in [1] is assumed.

A.1 Overview.

CLIPS interfaces with the object oriented programming extension by LHS functions and RHS commands implemented as external functions. The function arguments *object*, *class*, *instance*, *attribute*, *method*, or *function* refer to names which must be of type word. Attribute values can be of any type. Parameters for methods or attached procedures are subject to the protocols of the user supplied function.

Methods and attached procedures must be declared as external functions in `usrfuncs`. The *function* parameter refers to the CLIPS name declared for the function.

A.2 Object manipulation.

Classes are defined by a `defclass` construct which specifies attributes, methods, attached functions, and inherited classes.

Instance objects are created by instantiation of a class. They inherit all attributes, methods, and attached functions of the class.

Classes

```
(defclass class "comment "

(methods
(method function) (method function multiple) ...)

(instance-attributes
attribute[<-value] (attribute[<-value] [(when-set function)][(when-read function)]) ...)

(class-attributes
attribute[<-value] (attribute[<-value] [(when-set function)][(when-read function)]) ...)

(inherits
(class priority) class    ...)

)
```

Creates a new class with the given name and the specified methods and attributes. The given function names must be the CLIPS reference names as specified in `usrfncs`. Multifield values are enclosed in parenthesis. Inherited classes must be already defined.

Instances

```
(instantiate class instance [attribute<-value]... )
```

Creates a new instance object for the specified class giving it the specified name. Optionally specified attributes are initialized. The name of the new instance is returned as the function value so that `gen-sym` can be used to create instance names which can be bound to variables as the function value. The new instance becomes the current object and current instance.

(`delete-instance` *instance*)

Removes instance objects from the symbol table and releases their memory.

A.3 Attribute manipulation.

Valued attributes can be updated and referenced. Methods can be invoked.

Setting values

(`set-attribute` *object attribute [datum [parameter...]]*)

Sets the value of the specified attribute, or flags it empty if no data given. Any previous data is deleted.

The specified parameters are passed to any `when-set` procedures.

(`append-to-attribute` *object attribute datum*)

Appends the given datum to a multi-field value. If the attribute was not previously a multi-field value its contents, if any, becomes the first field. Appending does not invoke any attached procedures as it is not known if the multi-field value is complete.

Retrieving values

(`get-attribute` *object attribute [parameter ...]*)

Obtains the value of the specified attribute. Parameters if given are passed to any `when-read` attached procedures.

Invoking methods.

(`invoke` *object method [parameter...]*)

Invokes the function for the selected method of the specified object and passes it the given parameters. If the method was found in more than one inherited class only the first inherited with the highest priority is called unless the method was specified as `multiple` in which case all multiple designated functions are called.

A.4 Predicate functions.

Predicates about objects can be used to constrain patterns or as tests in the LHS of rules. They can thus prevent rules from making erroneous assumptions about objects.

Attribute of object?

(test is-attribute *object attribute*)

Determines if the named attribute is in fact an attribute of the specified object.

(test is-attribute-numberp *object attribute*)
(test is-attribute-wordp *object attribute*)
(test is-attribute-stringp *object attribute*)

Determines if the attribute is of a specified type. Returns *false* the specified object does not have the specified attribute.

Inheritance?

(test inherits *class object*)

Determines if the named object inherits the attributes of the specified class. The inheritance tree above the object is searched for the class.

A.5 Fact assertion

Facts about attribute values can be asserted into the CLIPS fact list so as to relate knowledge represented by objects to the inference engine. The facts are of the form (*object attribute value attribute value* ...).

Single assertion

(assert-attribute *object attribute* ...)

Asserts a fact giving the object name followed by ordered pairs of attribute name, and attribute value. For a multi-field value all fields are reported following the attributes name.

(assert-instance *object*)

As above for all attributes of an instance.

Multiple assertions

(assert-list *object attribute*)

Asserts a fact for each field of a multi-field attribute.

B Appendix: User function protocol.

This protocol must be followed when writing C functions which are to be invoked from the object oriented extension as Object methods. This includes procedures attached to attributes which are called when-set or when-read.

B.1 Overview.

Object methods are implemented as user defined C functions as specified in [2]. Information pertinent to the object oriented extension is provided via function call parameters.

The interface functions provided by CLIPS remain accessible. This includes the CLIPS parameter passing routines such as runknow.

Utility functions are provided for accessing and manipulating objects and attributes from user functions. These are used to access the invoking object or any named object.

when-set attached procedures are called before the attribute is updated. The new value is passed as the first CLIPS function parameter and is obtained using CLIPS interface routines. It is the responsibility of the function to update the attribute, possibly with a modified value.

All WORD or STRING valued parameters are represented by CLIPS hash pointers.

As well as including required CLIPS header files the source should include objects.h in order to access structures relating to the object oriented extension.

B.2 C Function parameters.

The object oriented extension calls methods and attached procedures with four parameters:

1. The name of the object which invoked the function. Type HASH_PTR.

2. The name of the attribute to which the the function is attached. NULL if not an attached function. Type HASH_PTR.

3. A pointer to the attributes data field. The field is a union of float, HASH_PTR, or MULDATUM.

4. The type of the data. An integer with possible values: WORD, NUMBER, STRING, or MULTIPLE as defined in constant.h.

5. The class from which the function was inherited.

The provided names allow for object specific and attribute specific processing. They can be used as parameters to object access utility routines.

The data field pointer can be used to access and update the attribute's value. If a when-set procedure the field will contain the previous value. The new value is obtained as the CLIPS parameter selected by parameter four. The when-set procedure is responsible for updating the value in the data field.

If the attribute is multi-valued the data field is a pointer to a linked list of the form:

```
typedef struct muldatum_fmt *  MULDATUM;

struct muldatum_fmt
{
          int data_type;          /* constant.h VALUE  */
          union {HASH_PTR symbolic; float numeric} data;
          struct muldatum_fmt *next;  /* LIST LINK         */
};
```

B.3 C Function returns.

when-read attached procedures return a resultant value which is reported as the attribute value. The procedure must be declared as an appropriate type and this type must be specified in usrfuncs.

B.4 Utility routines.

Certain object oriented commands are accessible to external C functions via function calls. The function names consist of the command prefixed by uf_ for "user function".

Parameters common to many routines:

Object identifier As with CLIPS function.

Attribute identifier As with CLIPS function.

type An int code indicating the type of an attribute value being affected. Can take the value NUMBER, WORD, STRING or MULTIPLE as defined in the file constant.h.

numeric A *float* attribute value.

alpha A HASH_PTR addressing either a CLIPS WORD or STRING attribute value.

B.5 Instance manipulation.

Creation

(uf_instantiate *class instance*)

Removal

(uf_delete_instance *instance*)

B.6 Attribute manipulation.

Setting values

uf_set_object_attribute(*object attribute [data [parameters]]*)
uf_append_to_object_attribute(*object attribute [data [parameters]]*)

Retrieving values

`(get_object_attribute` *object attribute [data [parameters]]*`)`

B.7 Predicate functions.

The truth of predicates is returned as CLIPS_TRUE or CLIPS_FALSE as defined in `constant.h`.

Attribute of object?

`uf_is_attribute(` *object attribute*`)`
`uf_is_attribute_numberp(` *object attribute*`)`
`uf_is_attribute_wordp(` *object attribute*`)`
`uf_is_attribute_stringp(` *object attribute*`)`

Inheritance?

`uf_inherits(` *class object*`)`

B.8 Fact assertion

Single assertion

`uf_assert_attribute(` *object attribute*`)`
`uf_assert_instance(` *object*`)`

Multiple assertions

`uf_assert_list(` *object attribute*`)`

References

[1] Giarratano, Joseph C. *CLIPS User's Guide.* COSMIC, The University of Georgia, 382 East Broad Street, Athens GA 30602.

[2] Artificial Intelligence Section, Lyndon B. Johnson Space Centre. *CLIPS Reference Manual.*

[3] Artificial Intelligence Section, Lyndon B. Johnson Space Centre. *CLIPS Architecture Manual.*

omiT

A12 Session:
Parallel and Distributed Processing II

C-4

S28-63

N96-12941

358610

p.14

MARBLE - a System for Executing Expert Systems in Parallel

Leonard Myers, Coe Johnson and Dean Johnson.

ICADS Research Unit
California Polytechnic State University
San Luis Obispo, California 93407

April 20, 1990

Summary.

This paper details the MARBLE 2.0 system which provides a parallel environment for cooperating expert systems. The work has been done in conjunction with the development of an intelligent computer-aided design system, ICADS, by the CAD Research Unit of the Design Institute at California Polytechnic State University[1].

MARBLE (Multiple Accessed Rete Blackboard Linked Experts) is a system of CLIPS shells that execute in parallel on a ten processor, shared-memory computer. Each shell is a fully functional CLIPS expert system tool. A copied blackboard is used for communication between the shells to establish an architecture which supports cooperating expert systems that execute in parallel.

The design of MARBLE is simple, but it provides support for a rich variety of configurations, while making it relatively easy to demonstrate the correctness of its parallel execution features. In its most elementary configuration, individual CLIPS expert systems execute on their own processors and communicate with each other through a modified blackboard. Control of the system as a whole, and specifically of writing to the blackboard is provided by one of the CLIPS expert systems, an expert control system.

Introduction.

The MARBLE project is a framework for executing simultaneous CLIPS expert systems in a tightly-coupled shared-memory parallel computer environment. Specifically, MARBLE modifies CLIPS 4.3[2] to implement a blackboard system[3,4] for control of narrowly focused expert systems that execute in parallel. The system is specifically intended to provide a platform for experimentation in the development of techniques for cooperative problem solving with multiple expert systems.

Cooperative problem solving approaches are of interest primarily for their promise to simplify the complexity of developing solutions to large ill-defined problems and because the use of multiple problem-solving agents can be mapped to parallel hardware architectures with the expectation of reducing execution time.

The Blackboard Model.

The design philosophy of MARBLE is based on the following hypotheses:

1. Large expert systems might be better engineered in the future as groups of independently developed specialized systems.
2. The control of cooperating expert systems can itself

be implemented as an expert system.

The two above ideas are quite simply based on the attempt to model a committee of experts, or a person who is advised by several experts. The expertise of the individual expert is distinct, and the manner in which the group operates is independent of the individual areas of expertise. Since committee work is common in human problem solving, it should be possible to model cooperative machine expert systems.

A chairperson, or project leader, focuses attention to specific sub-problems and maintains order. It is assumed that the meeting area provides a blackboard on which all important information is recorded. The chairperson uses the blackboard to provide a description of the current statements accepted by the team and to focus the attention of the team to the issues that must be considered to solve the problem. Team members are not permitted to communicate directly with each other. They must direct their comments to the leader, who uses the blackboard to provide the communication. Often, the blackboard is described as holding the current state of the solution to the problem and a history of its contents can be used to analyze the problem solving techniques of the team.

Although the blackboard model has been used in many projects over the last decade, the implementation of cooperation is in its infancy[5]. Therefore, it is important to develop a platform for experimentation with various approaches.

The basic architecture of MARBLE is illustrated in figure 1. The chairperson is replaced by the control expert system, and the team members are replaced by specific domain expert systems. The blackboard can be read by any expert system, but only the control expert system is permitted to modify it. In the simplest context, the control expert system examines suggestions from the other experts and summarizes the collective wisdom.

The PEBBLE Predecessor.

The MARBLE project follows the development of PEBBLE (Parallel Execution of Blackboard Linked Experts)[6]. PEBBLE is an initial attempt at executing multiple expert systems in a shared-memory parallel computer system under the blackboard model. It uses the C programming language to implement a simple expert system shell language in which the expert systems access a shared-memory blackboard. Communication between the experts is handled through action descriptors, which are small tables that protect their information from mutual access errors.

By compiling the PEBBLE expert system language and building a dependency graph from the conditions used in the production rules, efficient execution is obtained. PEBBLE also demonstrates the effectiveness of the action descriptor approach, but the limitations of its pattern matching make it inefficient to use in *the development of large expert systems.*

The powerful pattern matching capability of CLIPS and the ready availability of C-language source code make it an attractive candidate for replacement of the PEBBLE language. The use of CLIPS will also permit this research to focus on the cooperation of expert systems, rather than the continued development of the language itself. Thus MARBLE is born as the PEBBLE framework with CLIPS shells replacing the PEBBLE language shells.

The guiding principal in incorporating CLIPS into a PEBBLE-like configuration of parallel processing is to make the change as transparent to as much CLIPS code as possible. This is necessary in order to reliably make changes in the C code, which is an intricate fabric of interrelated functions and data structures, and to provide a platform that will make it possible to easily update the system with expected future versions of CLIPS.

Shaping MARBLE from PEBBLE.

Since the use of a blackboard is central to the intended application in the ICADS system, the primary problem is to implement a blackboard with CLIPS expert systems. The initial approach attempted to modify the CLIPS shell so that each expert could access the blackboard as an additional fact list kept in shared memory. This is complicated by the intimate connection between the fact list and the Rete network. As the coding changes to accomplish this transition were made, it became apparent that it would be necessary to make such basic alterations to CLIPS that it would jeopardize the ability to convenient replace the modified CLIPS shells with new versions.

It is important to understand why the PEBBLE shells cannot be directly replaced by CLIPS shells. In PEBBLE the facts are organized in a hashed table, similar to that of a symbol table for a compiler language. The rules reference the symbol table to obtain the addresses of variables used in their conditions. The blackboard facts are kept in a separate symbol table that is allocated in shared memory. Since all blackboard entries are uniquely defined by a "bb" prefix in their names, it is easy to make all of the references to blackboard values use the special symbol table while all other references use the symbol table that is local to the processor on which the rules are being executed.

In contrast to the organization of facts in PEBBLE, the facts in the CLIPS system are kept in a highly linked structure that specifically provides components to speed the execution of pattern matching. Each fact points to every condition with which it matches.

The rules in CLIPS are used to generate a pattern *network[7]. A node in the network represents the basic pattern of any fact that would satisfy a condition of a rule. When a fact matches a pattern, it is then further examined to bind the variables that may be used in more specific relations that must hold. After a fact is added to the working memory, or fact list as it is called in CLIPS, the fact is "pushed" through the pattern network. During this process "tokens" that represent matches of the fact with the patterns for the conditions in the rules are generated and distributed in the network. As a result, the network stores a knowledge of the "matches" that have been made at any particular point in time.*

The nodes are arranged in a manner so that each path in the network represents the set of conditions that are necessary to fire a rule. That is, if all of the nodes in a path were to have their conditions satisfied, the terminal node would identify a rule whose conditions have all been met. In essence, the network "remembers" what conditions have been met up to any particular point in time and processes new facts from that partial match. Thus new facts "add" to the partial match information and may result in the completion of requirements for a rule to fire.

This algorithm provides a very efficient way of determining the effect on the rules that should be produced when a fact is asserted. However, in order to obtain this efficiency the technique has deposited "memories" of the fact within the pattern network that represents the rules. If a fact is deleted, these "memories" must be removed; and in order to make the removal efficient, it is necessary to have pointers from the facts into the areas of the network where the "memories" are kept.

In order for MARBLE to provide a blackboard architecture similar to that used in PEBBLE, the domain expert programs and the control expert must execute their own CLIPS systems on their own processors. But this presents a real problem with respect to how separate CLIPS systems can share the facts that would be on the blackboard. For example, suppose rule 1 in one domain expert references the fact, "(bb wall 2 thickness 8)". Let us suppose that rule 2 in another domain expert references the same fact. If the fact is asserted onto the blackboard, then both of the domain systems need to "push" the fact through their respective pattern networks. This means that the address of the fact must be available to both domain systems. By using shared memory, the address could certainly be available to both. But the fact must also point into both of the networks to preserve the CLIPS code that keeps track of the matches that have been "remembered" in the network. This requires that the pattern network be in shared memory as well.

Placing the pattern networks and the blackboard into shared memory is not a very difficult task. But the C functions that implement CLIPS expect for the fact list to be one highly-linked structure. Since the blackboard facts need to be in shared memory, this implies that the local facts must also be in shared memory; or there could be two fact lists, one of which is in shared memory while the local fact list could be in local memory. In fact, an attempt to implement MARBLE with a shared memory blackboard, separate from a local fact list was attempted. The approach was abandoned as a series of changes to CLIPS code became necessary - changes that would compromise the ease with which new versions of CLIPS could be used for the system.

Instead of implementing the blackboard as one shared fact list, each expert shell now keeps a copy of the blackboard in its own fact list. At first, it may seem that this approach would produce a system inferior to the traditional blackboard model, in which the experts examine a common blackboard. However, just as distributed databases have achieved advantages over traditional localized databases, it will be noted later that there are some major conceptual advantages to the copied blackboard approach over the common blackboard implementation.

The implementation of MARBLE requires some basic changes to the manner in which CLIPS shells run. In order to simply the communication between the CLIPS shell that executes the control expert system and the CLIPS shells that execute the domain expert systems, all CLIPS data structures are stored in shared memory. Also, the run loop in each CLIPS shell is modified so that an exec function is called prior to the first rule firing, as well as immediately after each rule firing. In the event that an expert reaches a point where its agenda is empty, a special exec function is called repeatedly. The exec function invokes C functions that examine the action descriptor for the shell. As a result, every domain CLIPS shell checks for communication from the control expert, and the control expert CLIPS shell checks for communication from each domain expert after firing at most one rule.

Action Descriptors.

Action descriptors provide the link between the control expert system and the domain expert systems. Each domain system has an action descriptor in which it in can receive a request from the control unit, send a request to control and record its status to allow control to monitor its activities.

The interaction between the control expert system and the domain expert systems is configured by a finite-state machine diagram to make certain that there can be no deadlocks or uncontrolled interference. Whenever an action is initiated, the action descriptor is modified to show the state change that has occurred. When the action is finished, the action descriptor is again changed. Simple checking of the action descriptor guarantees that a change is proper, or the change is postponed. The control and the domain systems have their own areas within the action descriptor to permit concurrent action from both expert systems. The fields of the action descriptor and the possible values they represent are shown in Table 1.

MARBLE Architecture.

When the MARBLE system is started, it will do some initialization and then fork CLIPS loaders to each of the processors. In particular the action descriptors for the domain expert systems are initialized to indicate that they are "IDLE". The system waits until all of the loaders are ready to execute and then it begins to execute only the loader for the control expert system. The loader prompts the user for the filename of the CLIPS control expert. As a matter of form, the control expert must contain rules that use the function "activate" to initiate the loading of domain experts.

The control expert can start domain experts at any time. The function "any_idle" will tell the control expert if there are any CLIPS shells available for a new domain expert.

Loaders.

A loader is a modified CLIPS program. There are three versions of loaders, as follows:

- domain loader
- control loader
- I/O loader

The domain loader is a CLIPS shell that has been modified to examine the action descriptor of the processor on which the loader is executing. It will examine the action descriptor before each execution of an action on its CLIPS agenda. This makes certain that the domain expert system will pay immediate attention to the requests that come from control. If there are no items on the CLIPS agenda, the domain loader will continually examine the action descriptor, waiting for instructions from control. When the control system wishes to have the domain loader execute a domain expert system, it places the filename of the CLIPS domain ruleset into the action descriptor for the domain and then changes the action descriptor to indicate its wish for the domain loader to execute the domain ruleset. The domain loader reads the request in the action descriptor and executes a standard CLIPS "load" of the rules. Then, if the file loads without error, the domain loader changes the action descriptor to indicate it is beginning the execution of the domain expert system and executes a standard CLIPS startup, by asserting a CLIPS initial-fact. After this assertion, the domain loader runs as an enhanced CLIPS shell with a few new commands and the transparent examination of the action descriptor prior to the execution of each CLIPS command.

MARBLE User Functions.

MARBLE also requires the addition of several new user functions to the CLIPS language:

bb_assert, for the domain experts;
also, activate, any_idle, promote_fact, force_promote,
bb_retract and exit_marble, for the control expert.

The functions used to affect the content of the blackboard are bb_assert, promote_fact, force_promote and bb_retract. These four functions use a new parser which is a modified version of the CLIPS assert parser. This allows them to be called with the same syntax as the standard CLIPS assert. When a domain expert wishes to suggest a fact for the blackboard, it calls bb_assert with the fact as an argument. This new command sets the domain_action field of the action descriptor for the domain to REQUEST_ASSERT and places a pointer to the fact into the dargl field. When the control system inspects the action descriptor of the domain expert, it will perform the standard assertion code, using the address of the fact in the shared memory used by the domain expert, and assert the fact to the fact list, with "bb_consider" as the first argument. Facts beginning with "bb_consider" are only under consideration for posting to the blackboard.

By using the status, control_action and domain_action values of the action descriptor as a triple to identify the state of the action descriptor, a finite state transition graph can be constructed to show the valid sequences of operations. For example, in figure 2 when a domain expert is running with no communication pending, the state is 300. If the domain expert executes a bb_assert, the action descriptor will be changed to 301. This provides the request to the control expert. Then it is possible that the control expert will make a request for the domain to copy a value from the blackboard, before the control performs the domain request and changes the domain_action value. Thus the action descriptor might become 311. If the control did not make such a request, it would perform the bb_assert action and then reset the domain_action value back to 300.

By constructing the entire finite state transition graph from the point of view as to what should be possible, it is relatively easy to verify the code responsible for performing the actions associated with the action descriptor states. It is particularly important in the parallel environment to provide a proof of the conceptual plan to prevent invalid interactions between the various processes. In effect, the values in the action descriptors are used as semaphores to provide mutual exclusion in critical areas.

The control expert uses rules that evaluate the facts with "bb_consider" in their first fields, to determine if they should be promoted to the blackboard. If so, the control expert must choose between using the force_promote function and promote_fact. Both functions replace the fact with one that has a first field value of only "bb" and set the action descriptors of all active domain experts, to tell them to copy the new blackboard fact. The control_action value is set to ASSERT_BB and a pointer to the fact to be copied into the domain fact lists is placed into the cargl field. The functions differ in the form of the fact they send to be copied. Force_promote points to a fact beginning with "bb", while promote_fact sends a fact whose first field is "idt_consider". The first field value will tell the domain experts whether they must immediately assert the blackboard fact, or if they can delay in accepting it.

It is natural that the control expert system should "decide" what should be placed on the blackboard. In fact, a major reason for the control expert is to arbitrate between the domain experts when they make different recommendations for values of the same entity. However, it may at first seem unusual that this privilege is also extended to the domain experts. But consider the following! It is often the case that with a team of human experts, even after agreement has been reached by the group as a whole, an individual may continue to think differently. Moreover, if a domain expert feels that a particular attribute should have a specific value, important advice might be lost if the domain expert were forced to override its opinion. Control can ignore the continual suggestion of a value , but if the domain expert is "turned off" by a forced value, the control expert would not be receiving the best advice. Furthermore, the system can insulate itself from a cascade of trivial changes by allowing the domain experts to determine when to update their values of a blackboard fact. In the design environment, it is very possible that small changes should be ignored in the beginning phases of the design work, and that the domains can execute rules to identify different tolerances to use as the design progresses. It is also possible that when a major change is made in a drawing, the domains may be able to recognize this event and delay in accepting a quickly changing sequence of values for a blackboard fact until it is stable.

If the control expert uses force_promote exclusively, the domain experts will keep a very current copy of the blackboard. Remember that each domain expert can execute no more than one CLIPS action before checking its action descriptor, so the response is immediate. Also, since each processor will independently execute the assertion code to incorporate the fact into its own fact list, the entire process takes just a little over the time it would take to assert the fact into one fact list.

When a new domain expert system is loaded, it copies all of the blackboard values into its fact list. The control does not execute control rules until the copy is completed to guarantee the agreement of the blackboard contents between control and the domain, and to prevent any contamination of the blackboard. Thereafter, the blackboard assertions take place with a single fact at a time. Thus, the execution of the system is only slowed appreciably by the loading of new expert systems.

When a domain expert finishes its work, it performs a CLIPS halt. Then its CLIPS loader returns to the IDLE status, awaiting the loading of a new domain expert. When the control expert is finished, it calls the exit_marble function, which commands each domain process to exit. Exit_marble makes certain that all of the other processes have been killed before it exits.

Conclusions.

MARBLE has been used to implement a multi-person blackjack simulation in which the players execute in parallel. The design has also been used as a model for a distributed version of the blackboard that is currently being used with three networked computers for the first ICADS prototype system[8].

The most important result is that MARBLE provides a platform for experimentation in the development of techniques for synthesizing the efforts of concurrent expert systems. Moreover, the parallel environment provides this platform without the use of the complex scheduling algorithms that are needed in most blackboard systems. In addition, the use of shared memory eliminates the need for message passing, common to distributed blackboard systems.

When a CAD workstation that can execute the specific drawing system used in the ICADS prototype is added to the parallel system on which MARBLE runs, MARBLE will be used to execute the ICADS prototype with a greater degree of concurrence than the current networked system can provide.

References.

1. Pohl, J., A. Chapman, and L. Myers; 'ICADS: An Intelligent Computer-Aided Design Environment'; Proc. of ASHRAE Symposium on Artificial Intelligence in Building Design, St. Louis, IL., June 1990.

2. NASA; 'CLIPS Architecture Manual (Version 4.3)'; Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, May 1989.

3. Hayes-Roth, B.; 'A Blackboard Architecture for Control'; Artificial Intelligence, Vol. 26, 1985.

4. Nii, H.P.; 'Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures'; The AI Magazine, Summer 1986.

5. Klein, M.; 'Conflict Resolution in Cooperative Design'; Thesis, Computer Science Dept., University of Illinois, Urbana, IL., 1990.

6. Myers, L. Cheng, Erikson, Nakamura, Rodriguez, Russett and Sipantzi; 'PEBBLE: Parallel Execution of BlackBoard-Linked Experts'; Proc. SURF Conference, Newport Beach, CA., Sept. 1988.

7. Forgy, C.L.; Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem'; Artificial Intelligence, Vol. 19, No. 1, 1982

8. Myers, L. and J. Pohl, 'ICADS: DEMO1 - A Prototype Working Model'; Fourth Eurographics Workshop on Intelligent CAD Systems, Paris, France, April 1990.

FIELD	DESCRIPTION
status	current process status
control_action	action requested by control
domain_action	action requested by domain
proc	process id
carg1	fact pointer argument from control
carg2	string argument from control
darg1	fact pointer argument from domain

FIELD USAGE

FIELD	VALUE	DESCRIPTION	USE
status			
	-1	ERROR	error identification
	0	IDLE	free for new use
	1	READY_TO_LOAD	load sequence flag
	2	LOADING	domain is loading carg2 file
	3	RUNNING	domain is executing CLIPS
	4	STALLED	domain agenda is empty
	5	BB_COPY	domain requests blackboard
	6	HAS_EXITED	domain process is dead
control_action			
	-1	ERROR	error identification
	0	NONE_CURRENT	no current control command
	1	ASSERT_BB	control is sending new fact
	2	RETRACT_BB	control requests retraction
	3	COMMAND_EXIT	control commands an exit
domain_action			
	-1	ERROR	error identification
	0	NONE_CURRENT	no current domain request
	1	REQUEST_ASSERT	domain requests BB assert
	2	unused	(domains do not request retraction)
	3	DONE	domain CLIPS has exited

TABLE 1.ACTION DESCRIPTORS

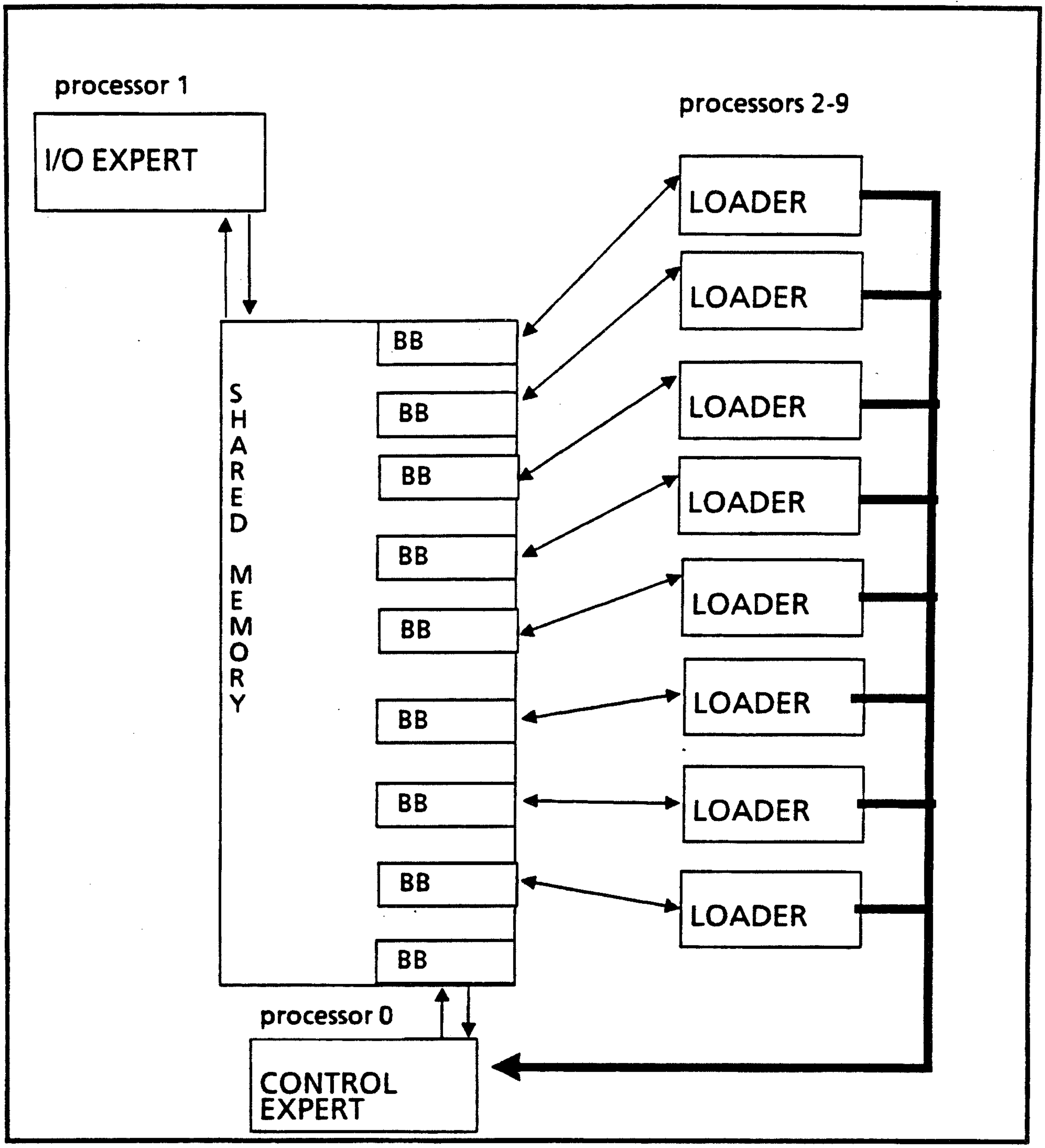


Figure 1: MARBLE Architecture

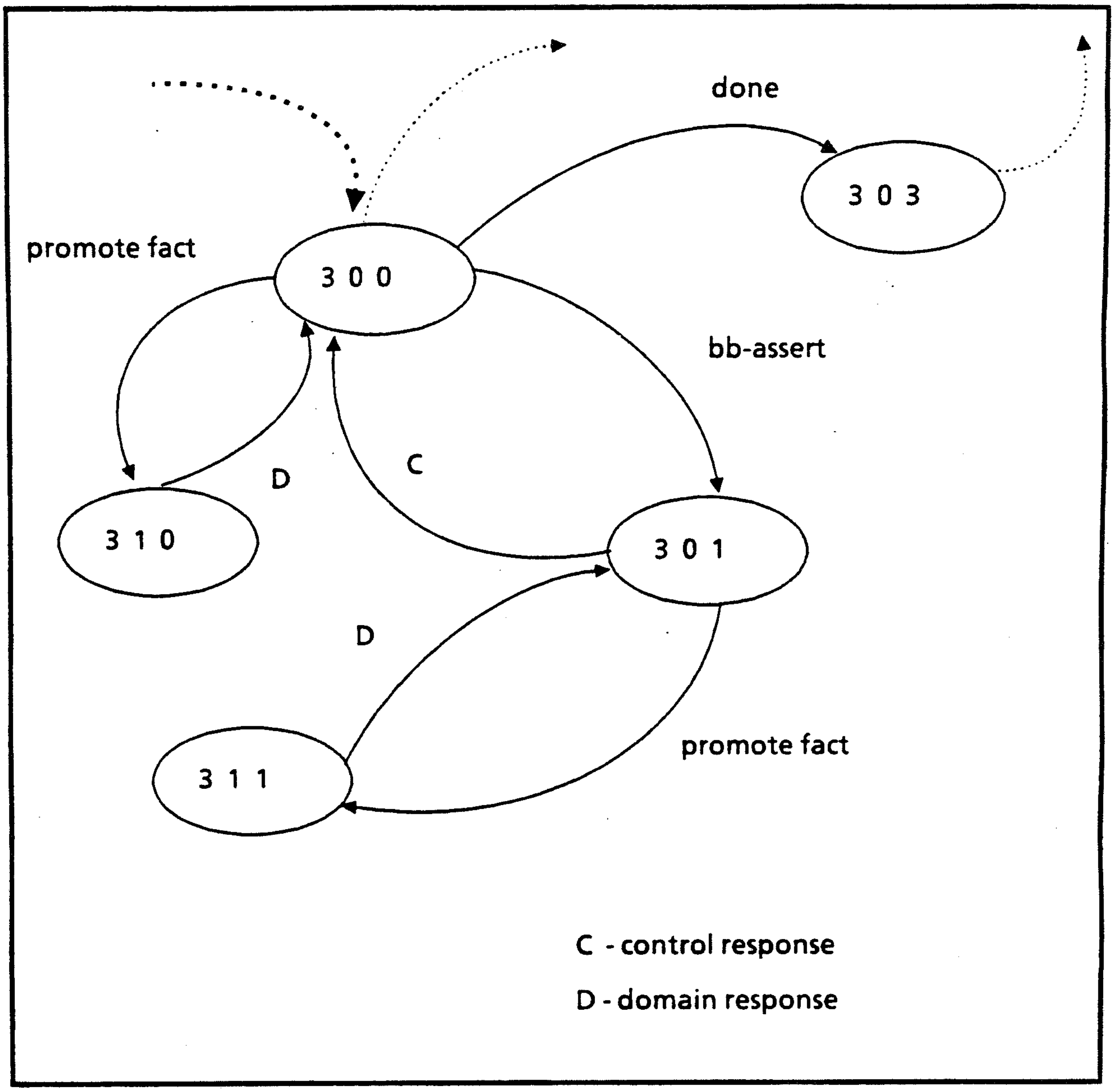


Figure 2: Partial State Transition Graph

529-61

N96- 12942

p. 8

Building Distributed Rule-Based Systems Using the AI Bus

Dr. Roger D. Schultz and Iain C. Stobie

Abacus Programming Corporation

358611

Abstract

The AI Bus software architecture was designed to support the construction of large-scale, production-quality applications in areas of high technology flux, running on heterogeneous distributed environments, utilizing a mix of knowledge-based and conventional components. These goals led to its current development as a layered, object-oriented library for cooperative systems.

This paper describes the concepts and design of the AI Bus and its implementation status as a library of reusable and customizable objects, structured by layers from operating system interfaces up to high-level knowledge-based agents. Each agent is a semi-autonomous process with specialized expertise, and consists of a number of knowledge sources (a knowledge base and inference engine). Inter-agent communication mechanisms are based on blackboards and Actors-style acquaintances. As a conservative first implementation, we used C++ on top of Unix, and wrapped an embedded Clips with methods for the knowledge source class. This involved designing standard protocols for communication and functions which use these protocols in rules. Embedding several Clips objects within a single process was an unexpected problem because of global variables, whose solution involved constructing and recompiling a C++ version of Clips. We are currently working on a more radical approach to incorporating Clips, by separating out its pattern matcher, rule and fact representations and other components as true object oriented modules.

1. Introduction

The AI Bus is a software architecture and toolkit which supports the construction of large-scale, production-quality cooperating systems in areas of high technology flux. It was first developed as an approach to integrating the Space Station software, and more recently has been applied to the Advanced Launch Systems project (ALS). Both applications share requirements of a long life-time, during which new technological advances should be seamlessly incorporated, and high degrees of autonomy. These two classes of requirements - the software engineering need for flexible methods for combining heterogeneous components, and the functional need to coordinate a mix of knowledge-based and conventional systems - led to the development of the AI Bus as a layered, object-oriented, distributed architecture.

This paper describes the concepts and design of the AI Bus and its current implementation as a Unix C++ library of reusable objects. After an introduction to distributed processing and a discussion of the facilities needed to build cooperating systems, we present the mechanisms provided by the AI Bus for these facilities. Particular emphasis is placed on supporting high-level models of cooperation and problem-solving, implemented via semi-autonomous agent processes with knowledge-based communication and control. Finally we describe our approach to using Clips as a common knowledge representation language for the prototype.

2. Overview of Distributed Cooperative Systems

A distributed system may be characterized as a collection of separate processes together with an interaction medium. This separation and the interaction medium may be physical, as in processors connected by a network, or logical, as in modules with semantically disparate representations. Although developments in the last fifteen years have taken advantage of hardware advances by distributing data and processing, the control has remained centralized in master-slave relationships. Machines are now "talking" to one another, but the question for cooperative systems is deciding what to say, when, and by whose authority. Just as humans form organizations in order to function more effectively - the whole is greater than the sum of the parts - the promise of cooperative systems is that they can tackle problems beyond the capabilities of current architectures.

Cooperative systems use advances in distributed processing - algorithms for load balancing, efficient network routing, error recovery procedures, synchronization mechanisms, etc. - but build on them by treating the distribution as part of the problem solving which needs to be represented and reasoned about. For example, a distributed database should appear coherent to its users, but maintaining its global consistency is impossible without synchronizing transactions, and this may be prohibitively slow. The promise of cooperative systems is that such problems are amenable to techniques of modelling the users' goals and plans, handling uncertainty and inconsistency gracefully, and adaptively allocating tasks and resources (Ref. [1,2]).

If an agent is to help another it must have a way to represent that agent's goals and plans, if it is to receive help it must know which agents are able to provide assistance and hence must model their abilities and resources, and if it is simply interested in avoiding conflict it must be aware of their planned use of shared resources. Thus facilities are needed for modelling capabilities and interests, above simple interface specifications, and knowledge-based protocols for negotiation. Some approaches to realizing these goals are (Ref [3]):

- Distributed Object-Oriented Systems (DOOS): A natural way to model cooperative systems uses the object-oriented paradigm of autonomous modules communicating via messages. Extending this paradigm to distributed environments involves difficult problems of several threads of control and no single shared space of objects. (Ref. [4,5,6)
- Blackboards: In contrast to the message-passing model of DOOS, blackboards are an organizational mechanism whereby agents share their current problem solving state. (Ref. [7])
- Integrative Frameworks: Systems which combine a number of different mechanisms to support various paradigms for developing and experimenting with large scale applications. (Ref. [8,9,10]).

3. Facilities Provided by the AI Bus for Building Cooperative Systems

3.1 Overview of Goals and Features

The AI Bus is an integrative framework for building cooperating systems with the following requirements:

- Technology Transparency: the architecture is open to allow integration of future advances and is portable across disparate platforms.
- High Performance: the emphasis is on production quality, rather than experimentation.
- Support multiple coexistent problem solving paradigms: DOOS, blackboards, expert systems.
- Standard interfaces for combination of components and communication between subsystems.
- Mixed conventional and AI Approaches: through standard interfaces; included is the ability to incorporate off the shelf commercial tools.
- Support for verification and validation: integrated tools include dynamic audit probes (which can feed diagnosis and repair modules) and static compile-time checking of interfaces.

3.2 Software Engineering Principles

The components are divided into layers based on their abstraction level: at the bottom are the physical entities, then the operating system components, then conventional tools such as databases and user interfaces, followed by knowledge-based tools such as inference engines, and at the top are generic applications such as diagnosis shells which simply need to be customized for a specific application. Each layer provides services to higher layers; since the internal details of a layer are hidden from others, software changes are localized and modules are easily replaceable; for performance reasons a layer is permitted to call a lower non-adjacent layer rather than strictly stepping through the intermediaries. The AI Bus is defined as a set of object classes, and implemented as a class library, which again enables the internal implementation of objects to be hidden from other objects. Off the shelf components can be integrated by wrapping them in a suitable interface, but clearly the degree of support they receive from other AI Bus services is proportional to their "white-box" nature. The layers and representative object classes are illustrated in Figure 1, while the inheritance between a few classes is shown in Figure 2.

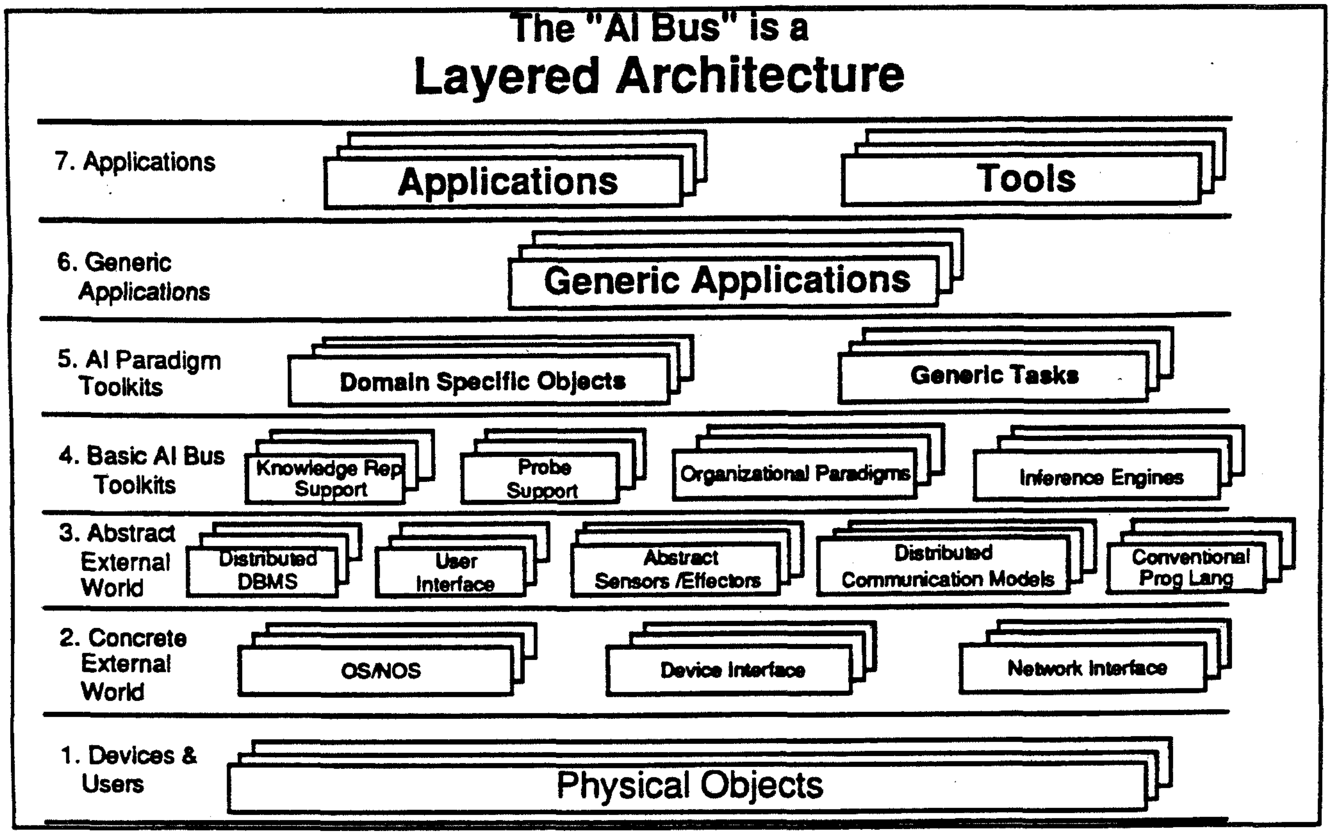
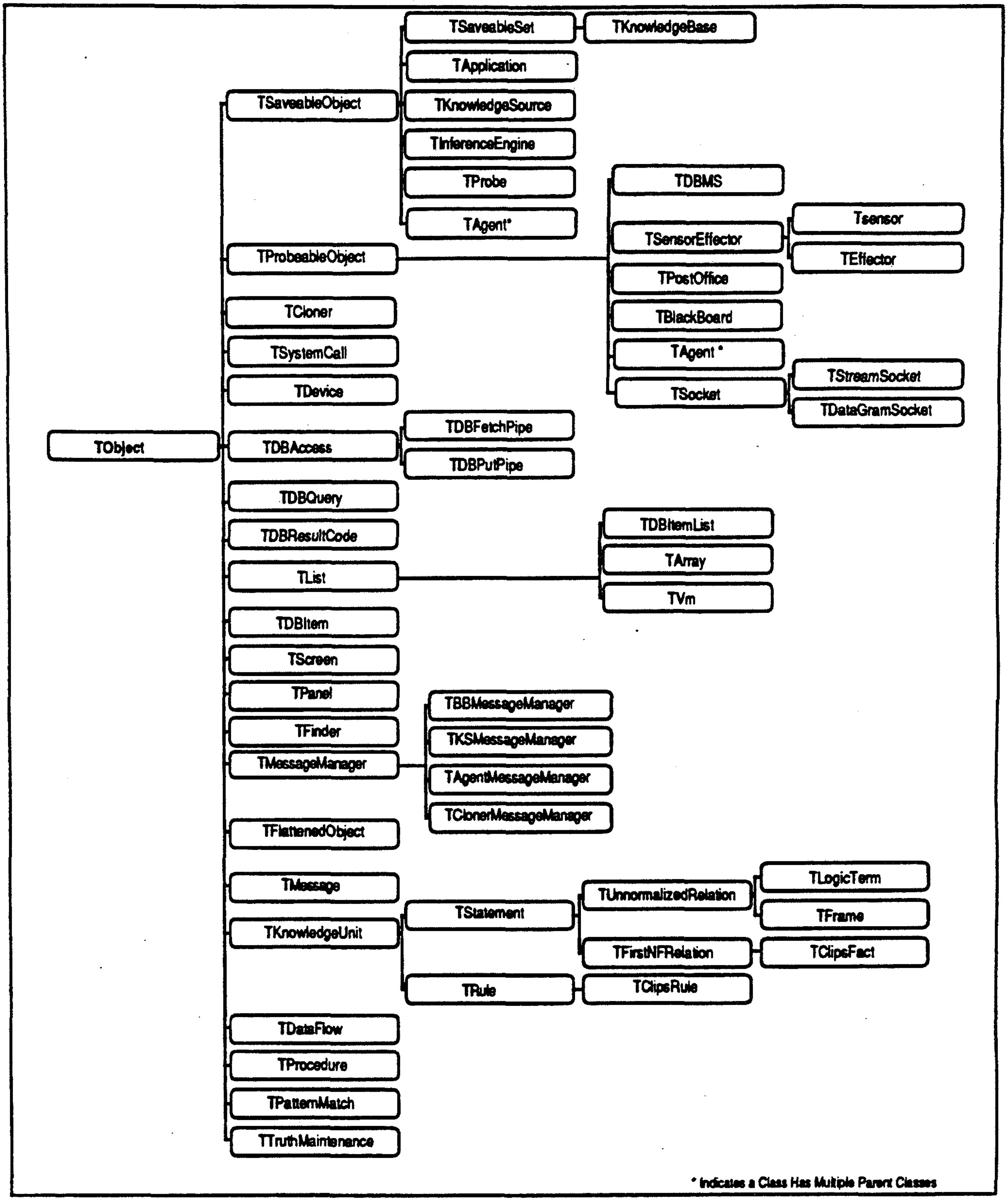


Figure 1. The layers of the AI Bus



* Indicates a Class Has Multiple Parent Classes

Figure 2. A Subset of the AI Bus Class Hierarchy

3.3 Probes, Messages, Agents and Organizations

3.3.1 Probes and Event-Driven Programming

The AI Bus follows the distributed object oriented model of interaction between software modules, here considered to be loosely-coupled agents. This not only supports the above software engineering principles, but also the open, continuous processing that is a characteristic of cooperative systems. Whereas event handlers in conventional systems, such as X Windows or database transactions, are invoked from a dispatch table using simple masks or triggers, the AI Bus extends this paradigm in its Probe object (Ref. [11,12]). A probe is activated based on matching patterns of events and conditions and routes information about subsystem activity to interested parties which can install and modify them dynamically. A probe's history can be used to maintain partial matches for efficiency (e.g. in the blackboard), its priority can be used to order the actions of several probes. A standard event, condition and action language allows the evaluation and interpretation of probes to be implemented by the probed object - a class of probeable objects is specified, and includes databases, network communication, blackboards and agents; there are corresponding subclasses of probes.

Probes can be used to support validation in a testbed environment and to monitor resource usage and each other. Since they are implemented by the probed object and installed by request, this access does not violate the secure boundaries of active objects. A subclass of probes called abstract sensor/effectors can be used in hierarchical process control applications - like probes they provide data, retain state and do some filtering, but in addition they recognize alarm situations and provide direct pathways between each other for fast response.

3.3.2 Communication Substrate

A layer of services exists between the operating system and the programming tools which allows the developers to concentrate on problem-solving rather than worrying about actual physical locations. Of course, for some applications, physical parameters are part of the problem definition (e.g. communication delays, noise and failures) and so are available for querying. Each agent has a Post Office object, which queues incoming messages and permits addressing by name, rather than location. The Post Office uses a distributed Finder object, which keeps track of the addresses of active objects and maps them to their globally unique names. Furthermore, agents can advertise certain attributes (see later section) which are also registered with the Finder and permit communication by knowledge rather than just syntactic names.

The interaction medium is the message, the glue which enables the transfer of data and control between the agents. A message contains fields which identify the sender and receiver, an object (such as a question or answer) an optional time tag and list of attributes, which may include its expiration date or other application-specific information. Control is passed via messages which represent remote procedure calls - they are intercepted by an agent's Message Manager, which is responsible for converting messages to procedures, and keeps a queue of questions received together with their askers (for subsequent direction of replies). Remote procedure calls by default are asynchronous - the caller doesn't block and wait for its completion - but may be synchronous if required. The question of whether the receiving agent blocks until it processes the request depends on the organization used: if the agent does, it is under the control of the sender (a client-server relationship), if not it is autonomous. Of course, requests to lower-level services (such as a database manager) are processed synchronously - only high-level agents can own a thread of control.

3.3.3 Agent

The agent is the fundamental active entity in the AI Bus, encapsulated as an object which communicates by messages. Currently an agent and its message manager occupy a Unix process, so its boundary exists not only as a software object but is also enforced at the operating system level. An agent is defined as a collection of knowledge sources and an organization; these knowledge sources may be implemented as expert systems (an inference engine and a knowledge base) or a conventional system - just so long as the specified interface is followed. Each knowledge source has a list of capabilities and interests - which match questions it can answer and information it would like to be told - the agent advertises these attributes with the Finder and keeps a cache of other agents' capabilities and interests for subsequent communication.

An agent's specification thus permits implementation along several sizes of granularity. Internally, it can be a whole organization of problem solvers, or just a simple procedural program. It has a scheduler component for control of its knowledge sources and is not necessarily serial (it may be realized as one or several processes or threads). Its state may be dormant or active, but currently most agents are eternally vigilant or waiting for a reply. For efficiency reasons in Unix-like environments a large grain may be preferred, and this can be used at the next layer up as a generic task - an agent which is a specialist in one area of problem solving (Ref. [13]).

An agent's capabilities and interests represent a model of its goals, plans, abilities and needs that other agents can use for cooperation. An agent can choose not to cooperate by not advertising this model, but in general they can build up more extensive models of each other by starting with the originally advertised capabilities and interests and then learning from experience by caching results: for example, two agents may have a capability to do arithmetic, but by trying each the faster one is identified and will be preferred in future requests. An agent can have a reflective ability by installing probes in itself (for example, to measure the number of rules fired by a knowledge source's inference engine); this allows it to monitor its progress and interrupt if necessary. The combination of agents into a cohesive problem-solving team is achieved by creating an organization. One example of the internal organization of a complex agent is illustrated in Figure 3.

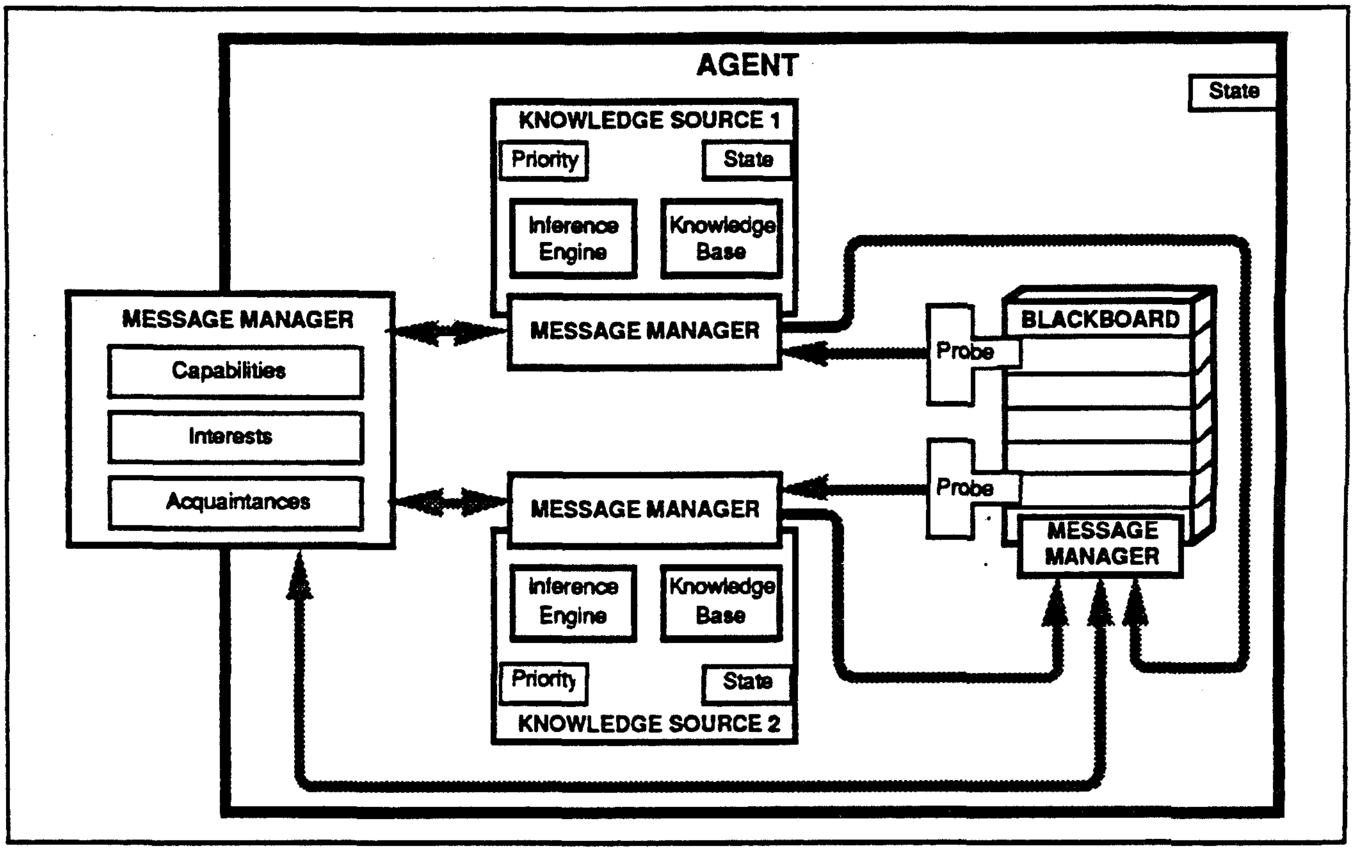


Figure 3. An Example of an Agent Composed of Several Layer 4 Objects

3.3.4 Organization

An organization is simply a collection of agents who know each others' capabilities and interests - this is an implicit specification by knowledge existing in each agent. In contrast to structural definitions of organizations, this model is adaptive, since agents can compute who knows how to answer a question it cannot itself process, and thereby new relationships form within the organization. One agent can be programmed to act as a manager, who delegates work to other agents according to their advertised capabilities, monitors their progress using probes and adjusts their position in the organization .

A final method to combine agents is more indirect, by sharing access to a blackboard. A blackboard is realized in the AI Bus as a restricted subclass of agent - it is a passive server which is interested in everything (or at least whatever it is programmed for). Agents post information on the blackboard by sending it messages, they install probes on it to gather information resulting from matching events plus several current and historical conditions. A blackboard is thus a semi-permanent communication space, but also acts as a mechanism for loosely-coupled organization whereby several agents can combine partial results without repeated inter-agent communication. It is more than a global database, in that the probes' histories provide a short-term memory and record of partial matches, so that new additions and requests can be processed quickly (in the style of the Rete algorithm for rule-based systems); in contrast, database queries are processed one at a time. This is an object-oriented version of the blackboard concept, and it is important to contrast it with blackboard systems which contain a centralized scheduler in control of the serial execution of agents: in the AI Bus the agents are autonomous. Although logically centralized, a blackboard may be physically distributed for performance reasons: in this case, consistency must be maintained using techniques (e.g. multiple copies, deadlock avoidance) borrowed from distributed databases. An illustration of the different methods of communication and cooperation is shown in Figure 4.

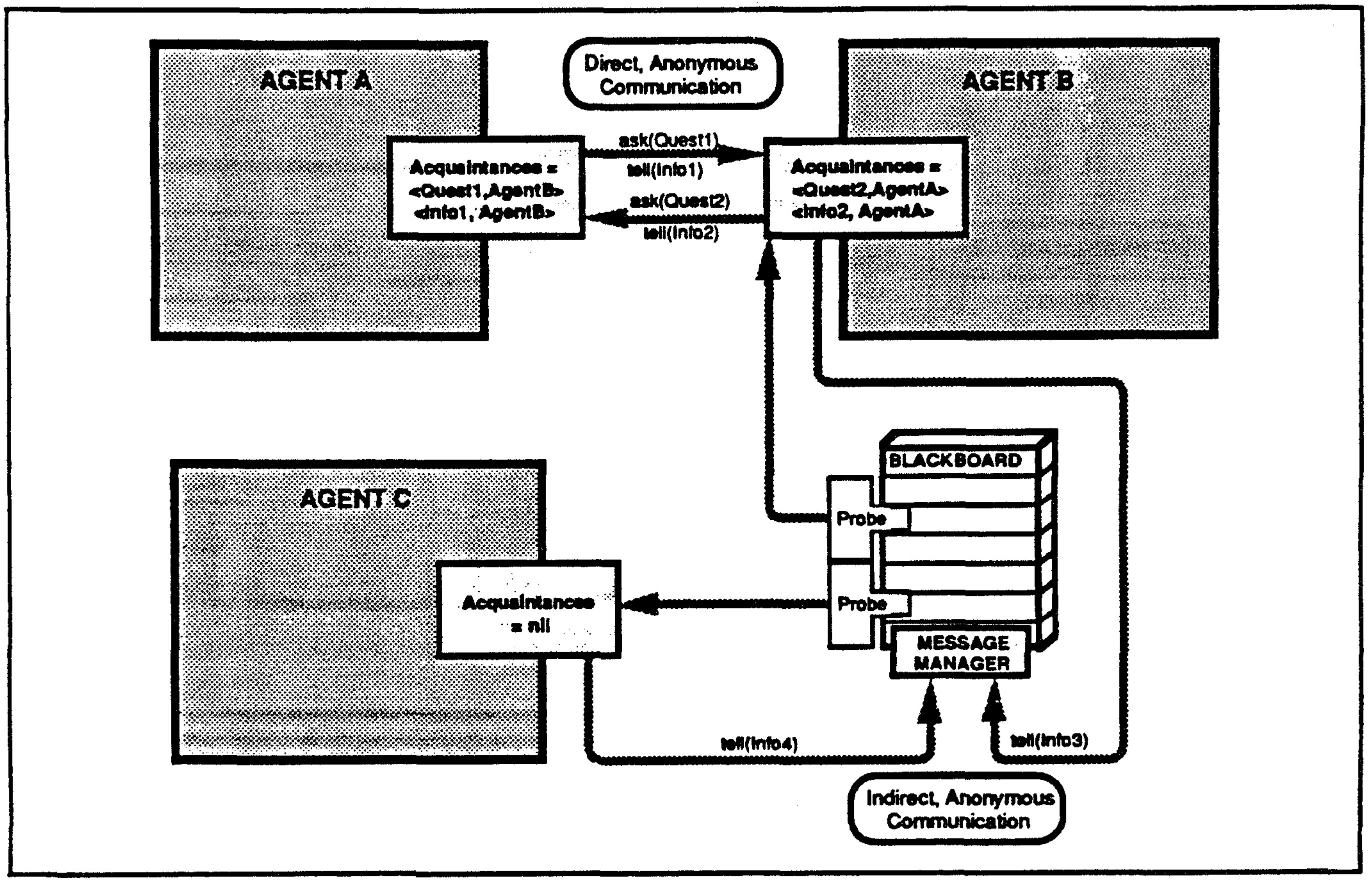
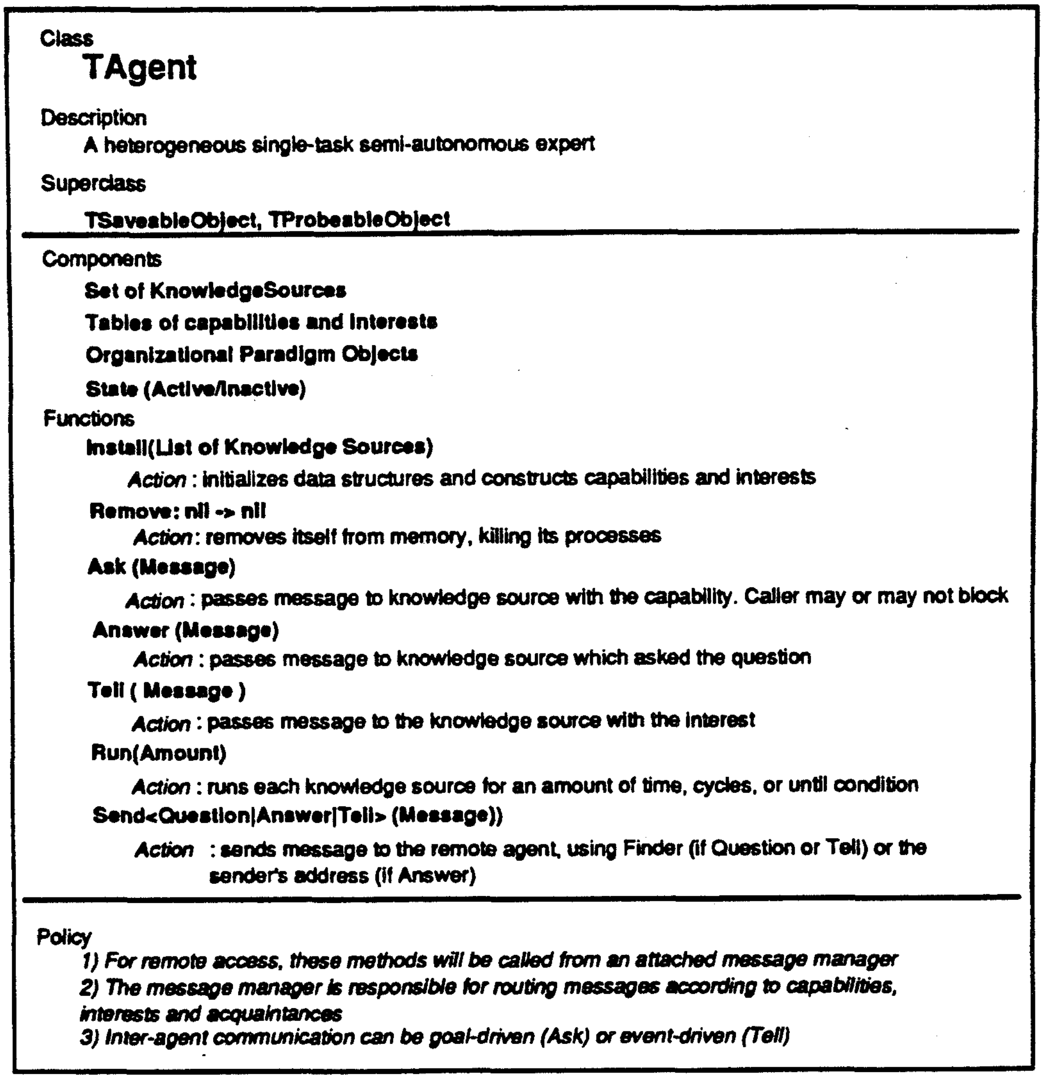

AGENT A
Direct, Anonymous Communication
AGENT B
Acquaintances = <Quest1,AgentB> <Info1, AgentB>
ask(Quest1)
tell(Info1)
ask(Quest2)
tell(Info2)
Acquaintances = <Quest2,AgentA> <Info2, AgentA>
AGENT C
BLACKBOARD
Probe
Probe
Acquaintances = nil
MESSAGE MANAGER
tell(Info4)
tell(Info3)
Indirect, Anonymous Communication

Figure 4. Agents Communicate directly with their Acquaintances and indirectly via Blackboards

4. Development of the AI Bus

The design of the AI Bus was first summarized in a set of abstract-data-type class specifications, intentionally kept language-independent in order to avoid restricting the design. See Figure 5.



Class

TAgent

Description

A heterogeneous single-task semi-autonomous expert

Superclass

TSaveableObject, TProbeableObject

Components

Set of KnowledgeSources

Tables of capabilities and interests

Organizational Paradigm Objects

State (Active/Inactive)

Functions

Install(List of Knowledge Sources)

Action : initializes data structures and constructs capabilities and interests

Remove: nil -> nil

Action: removes itself from memory, killing its processes

Ask (Message)

Action : passes message to knowledge source with the capability. Caller may or may not block

Answer (Message)

Action : passes message to knowledge source which asked the question

Tell (Message)

Action : passes message to the knowledge source with the interest

Run(Amount)

Action : runs each knowledge source for an amount of time, cycles, or until condition

Send<Question|Answer|Tell> (Message))

Action : sends message to the remote agent, using Finder (if Question or Tell) or the sender's address (if Answer)

Policy

1) For remote access, these methods will be called from an attached message manager

2) The message manager is responsible for routing messages according to capabilities, interests and acquaintances

3) Inter-agent communication can be goal-driven (Ask) or event-driven (Tell)

Figure 5. Class Description for an Agent

4.1 Initial Implementation Approach

For the implementation, we chose C++ and Unix because of the performance benefits of a relatively low-level language and its wide availability: a fundamental goal was to build a production quality system, not an experimental testbed. For the common knowledge representation language (a KnowledgeUnit class) we chose Clips because it is distributed with source code and hence is amenable to customization. Message passing between Clips agents was easily accomplished by writing three user-defined C++ functions (aibus_ask, aibus_tell, aibus_answer) that are called from the right hand side of a Clips rule and in turn invoke the encapsulating agent's methods (Figure 5) to interface with remote objects. The communication services were built on top of the RPC protocol.

This choice of implementation tools resulted in the compromise that an agent could only contain one knowledge source: a C++ object resides in one process, but having several Clips instantiations in one process is impossible because of its global variables. Furthermore, we had hoped to isolate out the inference engine components (pattern match, agenda scheduling, etc.) from the Clips source code for reuse in other Layer 4 objects such as probes and blackboards; however C++'s strongish typing caused problems in handling free-style C, even with the help of Abacus' automatic translation system, MetaPack. As a result we had to write our own procedures for these purposes and just treat Clips as a black-box KnowledgeSource object rather than a composite object (see Ref. [14] for the approach used in the Joshua system).

4.2 Current Implementation Direction

We are currently pursuing the direction outlined above, of decomposing the functionality of a Clips based inference engine into object-oriented modules. Agents could then have several Clips components, rules could inherit conditions and actions from other rules, rule-bases could inherit rules from other rule bases and the distinction between rule-based and frame-based languages would disappear. For hard real-time situations, the Rete net's good average-time but unpredictable worst-time performance is unsuitable and alternative implementations are necessary. For example, linear searching with compiled-out pattern matching (e.g. L*Star, Ref. [15]), or search algorithms like iterative deepening which always maintain the best-solution-so-far.

We are also working on incorporating non-linear fact and pattern representations (e.g. Prolog's recursive structures), and providing more support for probe access to AI Bus objects, especially for dynamic validation. At the cooperative systems level, we are experimenting with negotiation protocols, and providing agents with learning capabilities.

References

[1] Lesser,V. Corkill,D. *The Distributed Vehicle Monitoring Testbed.* AI Magazine, Fall 1983

[2] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. *Coherent Cooperation Among Communicating Problem Solvers.* IEEE Transactions on Computers, C-36:1275-1291, 1987

[3] Alan H. Bond and Les Gasser. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[4] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986

[5] Yutaka Ishikawa and Mario Tokoro. *Orient84/K: An Object Oriented Concurrent Programming Language for Knowledge Represetnation* in Object Oriented Concurrent Programming, Yonezawa & Tokoro, eds, MIT Press, 1987

[6] A. Yonezawa, J-P. Briot, E. Shibayama. *Object-Oriented Concurrent Programming in ABCL/1.* in [3]

[7] Penny Nii. *Blackboard Systems.* AI Magazine Volume 7, nos. 3 and 4

[8] R. Bisiani, F. Alleva, A. Forin, R. Lerner, M. Bauer. *The Architecture of the Agora Environment* in Distributed Artificial Intelligence, Michael N. Huhns, Ed., Morgan Kaufman, 1987

[9] Lee D. Erman, Jay S. Lark, and Frederick Hayes-Roth. *ABE: An Environment for Engineering Intelligent Systems.* IEEE Transactions on Software Engineering 14(12), December 1988

[10] Les Gasser, Carl Braganza, Nava Herman. *Implementing Distributed AI Systems Using MACE.* in [3]

[11] Roger D. Schultz and A. Cardenas. *An Approach and Mechanism for Auidtable and Testable Advanced Transaction Processing Systems.* IEEE Transactions on Software Engineering, SE-13 (6), June 1987

[12] Roger D. Schultz and A. Cardenas. *An Expert System Shell for Dynamic Auditing in a Distributed Environment.* ACM SIGSAC '87 Conference Proceedings

[13] B. Chandrasekaran. *Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design.* IEEE Expert, Fall 1986

[14] S. Rowley, H. Shrobe, R. Cassels, W. Hamscher. *Joshua: Uniform Access to Heterogeneous Knowledge Structures*. AAAI-87

[15] T. Laffey, P. Cox, J. Schmidt, S. Kao. J. Read. *Real-Time Knowledge-Based Systems*. AI Magazine, Spring 1988

S30-63

N96-12943

P-10

358761

Executing CLIPS Expert Systems in a Distributed Environment

James Taylor
IntelliCorp, Mountain View, California, USA

Leonard Myers
CAD Research Unit, California Polytechnic State University
San Luis Obispo, California, USA

This paper describes a framework for running cooperating agents in a distributed environment to support the Intelligent Computer Aided Design System (ICADS), a project in progress at the CAD Research Unit of the Design Institute at the California Polytechnic State University. Currently, the system aids an architectural designer in creating a floor plan that satisfies some general architectural constraints and project specific requirements. At the core of ICADS is the Blackboard Control System. Connected to the blackboard are any number of domain experts called Intelligent Design Tools (IDT). The Blackboard Control System monitors the evolving design as it is being drawn and helps resolve conflicts from the domain experts. The user serves as a partner in this system by manipulating the floor plan in the CAD system and validating recommendations made by the domain experts.

The primary components of the Blackboard Control System are two expert systems executed by a modified CLIPS shell. The first is the Message Handler. The second is the Conflict Resolver. The Conflict Resolver synthesizes the suggestions made by domain experts, which can be either CLIPS expert systems, or compiled C programs. In DEMO1 [1], the current ICADS prototype, the CLIPS domain expert systems are Acoustics, Lighting, Structural, and Thermal; the compiled C domain experts are the CAD system and the User Interface.

COMMUNICATION FRAMEWORK

The communications framework supports multiple hierarchies of connections among both C and CLIPS processes. Each connection provides an independent two-way stream communication path between processes using UNIX sockets [2]. The current network of connections demonstrates some of the possibilities (Fig. 1). From the point of view of the Blackboard Message Handler (MH), the Conflict Resolver consists of a single connected component. However, to increase performance, the rule set of the Conflict Resolver was divided into three independent rule sets and distributed as separate processes across the network. The User Interface has also been divided into two processes to take advantage of the organizational power of the Rete Network in CLIPS and the graphical display capabilities of the X Windows Tool Box.

MESSAGE HANDLER

The part of the Blackboard called the Message Handler (MH) is a CLIPS expert system with additional functions for message passing. The MH has two primary functions. First, it initializes the system by starting each IDT. Second, it distributes modified values to IDTs that request them. The MH initializes the system in two phases. During the first phase, the MH establishes a connection with the IDT to allow message passing, and receives the input requests specifying the blackboard values the IDT needs to produce its results. During the second phase, the MH builds a hash table and transmits it to each IDT to reduce future message sizes. An important prerequisite in this framework is that all system components use the same naming convention. Without a consistent naming convention, too much time would be spent converting between different representations. This common naming scheme is provided by a frame-based representation developed as part of the ICADS project [3].

REPRESENTATION

The particular-frame based representation used in ICADS is implemented as a set of CLIPS facts. A frame is a collection of information about a class or object. The information is represented in CLIPS with a frame header fact and any number of slot facts. Slots can define a particular value of the class or identify a "has-a" relation to another class.

A frame header is a fact of the form:

(FRAME <class> <instance>) where
FRAME is a keyword,
<class> is the name of the class of this frame, and
<instance> is the frame identification number.

The FRAME header is useful in performing operations on the entire frame (ie. deleting the frame), but is not needed to access the slots within the frame.

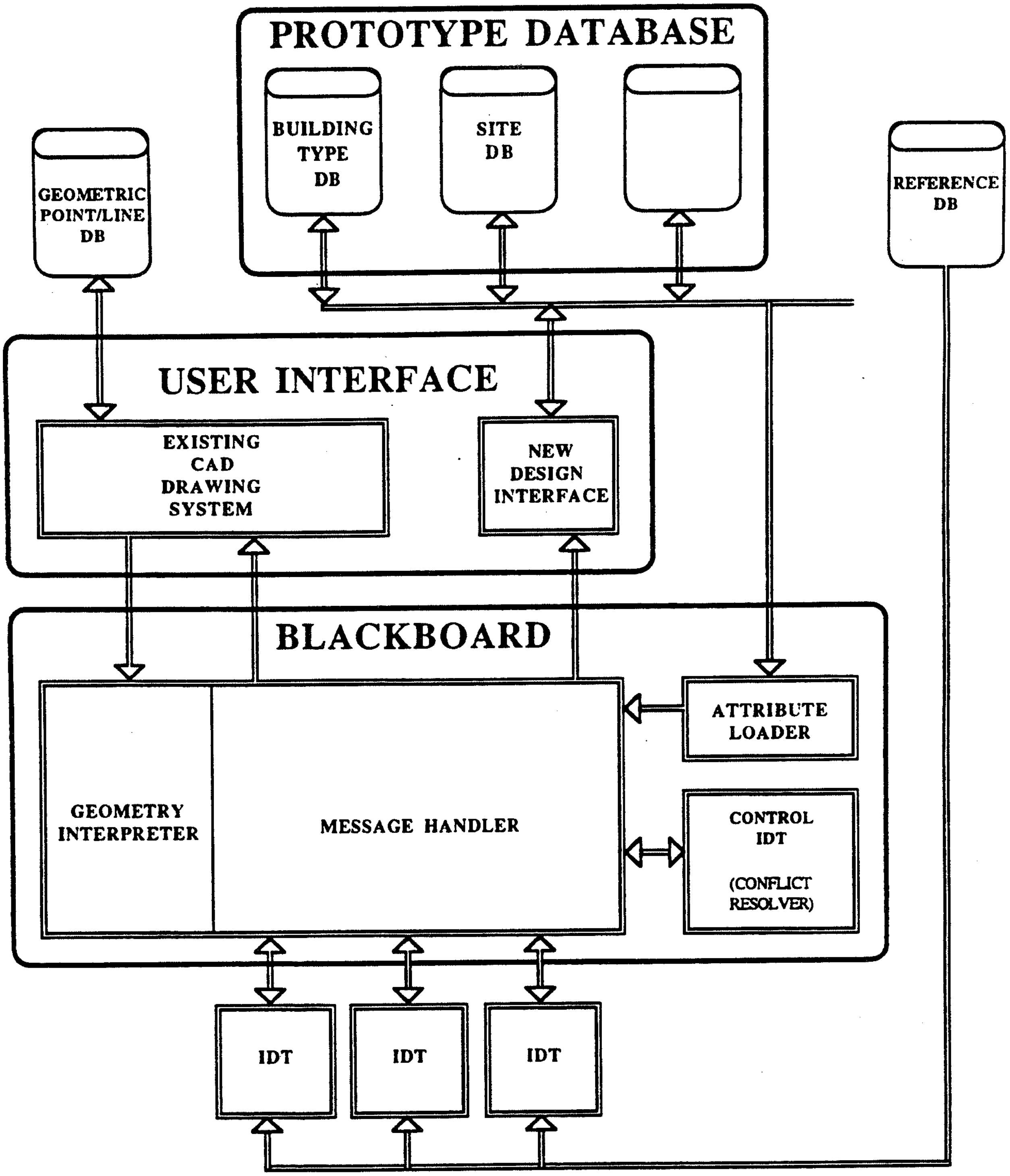


Figure 1: ICADS System Diagram

A value slot is a fact of the form:

```
(VALUE <class> <attribute> <instance> <value>) where
VALUE is a keyword,
<class> and <instance> are the same as in the frame header,
<attribute> is the slot name or attribute, and
<value> is the actual value of the slot.
```

The <value> field is one or more values, depending on the nature of the slot. For example, a slot for the coordinate of a point would have two values, whereas a slot for the length of a wall would only have one value.

A relation slot is a fact of the form:

```
(RELATION <class1> <class2> <instance1> <instance2>) where
RELATION is a keyword,
<class1> and <class2> are the names of classes, and
<instance1> and <instance2> are the frame identification
numbers of <class1> and <class2> respectively.
```

An example of an architectural object is the room or space object. Shown below is an instance of the class 'space' with an id number of 15, a name of LOBBY, a center coordinate of (128, 384), a perimeter of 108 feet, and four walls:

```
(FRAME space 15)
(VALUE space name 15 LOBBY)
(VALUE space center 15 128 384)
(VALUE space perimeter 15 108)
(RELATION space wall 15 1)
(RELATION space wall 15 2)
(RELATION space wall 15 3)
(RELATION space wall 15 4)
```

Changes to existing frames are made by inserting an action as the first field of the slot. Slots can be added, deleted, and modified using the keywords ADD, DELETE, and MODIFY. The ADD action asserts the slot. The DELETE action retracts the slot, and the MODIFY action retracts the existing slot and asserts the new slot. For example, if the above instance of a 'space' class exists and (MODIFY VALUE space area 5 216) is asserted, then the following actions occur:

```
retract (VALUE space area 5 108)
assert (VALUE space area 5 216)
retract (MODIFY space area 5 216)
```

When the DELETE action is asserted with the frame header, the entire frame (ie. all slots and the header) is retracted.

EXTERNAL FUNCTIONS

The external functions added to CLIPS to implement message passing are divided into two categories -- initialization and transmission. Messages are composed of any number of slots (ie. CLIPS facts), and are received explicitly with an external function that asserts the slots in the message. Messages are built with commands that have been added to the standard CLIPS command set and have the same syntax as the CLIPS 'assert' command.

INITIALIZATION FUNCTIONS

The functions used during initialization are briefly described below:

(new_server <name of process>):
Called by the MH and IDTs to create a server to allow future connection. Returns zero if no errors occurred.

(connect_bb [<name of message handler>]):
Called by an IDT to establish a two way connection between the IDT and the MH. Returns IDT identification number. If no argument is present, the IDT identification number is returned.

(accept_idt):
Called by the MH to establish a two way connection between the MH and an IDT. Returns IDT identification number.

(unaccept_idt <IDT id number>):
Called by the MH to terminate the connection between the MH and the IDT specified. Returns zero if no errors occurred.

(insert_hstring <field1> <field2> ...)
Called by the MH and IDTs to add a string composed of the concatenated fields to the hash table. Returns zero if no errors occurred.

TRANSMISSION FUNCTIONS

The functions used during the transmission of facts are briefly described below:

(receive_message [<IDT id number>]):
Called by MH and IDTs to receive a message in FIFO order and assert the facts in the message. Receives a message from only the MH, if zero is supplied as the IDT id number. Receives a message from only the IDT specified, if IDT id number is supplied. Returns zero if no errors occurred.

(bb_assert (<fact 1>) [(<fact 2>) ...]):
Called by IDTs to add facts to the message buffer. Uses the same syntax as the CLIPS 'assert' command. Returns zero if no errors occurred.

(bb_end_message):
Called by IDTs to send the message buffer built with the bb_assert command to the MH. Returns zero if no errors occurred.

(idt_assert <IDT id number> (<fact 1>) [(<fact 2>) ...]):
Called by MH to add facts to the message buffer of the IDT specified. Separate message buffers are maintained to allow messages for different IDTs to be built simultaneously. Returns zero if no errors occurred.

(idt_end_message <IDT id number>):
Called by MH to send the message buffer built with the idt_assert command to the IDT specified. Returns zero if no errors occurred.

INITIALIZATION

The Message Handler (MH) has two phases of initialization. In the first phase, it starts each IDT, establishes a connection to allow message passing, and receives input requests specifying the slots an IDT requires as input. Each IDT sends its input requests as its first message in the form of 'input' value slots in an 'idt' frame. The following example demonstrates the actions performed by the MH and two IDTs during the first phase:

```
        MESSAGE HANDLER
(new_server "mhandler")
(system "sound.start")
(receive_message (accept_idt))
(system "light.start")
(receive_message (accept_idt))
```

```
        SOUND IDT
(new_server "sound")
(bind ?no (connect_bb "mhandler"))
(bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME space)
        (ADD VALUE idt input ?no FRAME space name)
        (ADD VALUE idt input ?no FRAME space area))
(bb_end_message)
```

```
        LIGHT IDT
(new_server "light")
(bind ?no (connect_bb "mhandler"))
(bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME wall)
        (ADD VALUE idt input ?no VALUE wall length)
        (ADD VALUE idt input ?no RELATION wall window))
(bb_end_message)
```

As shown above, an optional argument is supplied to receive_message to specify that the next message be received only from the most recently started IDT. This prevents messages sent by previously started IDTs from being mistakenly received and interpreted as the input requests for the most recently started IDT.

In the second phase of initialization, the MH builds a hash table to decrease the percentage of time spent transmitting messages by reducing the amount of information sent across the network. This technique reduces message sizes by a factor of four or five. The MH builds the hash table from the input requests of the IDTs. The keyword and class name fields of the input request slots are concatenated into a string and entered into a hash table. Then, when an instance of that slot is added to the message buffer with bb_assert or idt_assert, the string of consecutive words starting with the second field is converted to a hash code, transmitted across the network as an integer, and then converted back to the original string of words upon receipt. If the string cannot be found in the hash table, each field is transmitted as a sequence of separate words. To insure that the hash code is correctly converted back to the original fields, the MH and all IDTs must have identical hash tables. Thus, even though an IDT may never receive a particular slot, the slot name is still contained in the hash table of the IDT.

Using the example from Phase I, the following strings would be entered into the hash table of the MH, the sound IDT, and the light IDT:

```
        (insert_hstring FRAME space)
        (insert_hstring VALUE space name)
        (insert_hstring VALUE space area)
        (insert_hstring FRAME wall)
        (insert_hstring VALUE wall length)
        (insert_hstring RELATION wall window)
```

When the slot shown below is added to the message buffer, the second, third, and fourth fields (ie. VALUE space name) are converted to a single integer hash code, sent across the network, and converted back to the original three fields upon receipt of the message.

```
        (bb_assert (MODIFY VALUE space name 5 RECEPTION))
```

DISTRIBUTION

After initialization, the basic loop of the MH receives the next available message, distributes the slots of the message to the IDTs that request them, and then retracts the slots. The following rules accomplish this for VALUE slots:

```
(defrule receive-message
        (declare (salience 40))
        ?f <- (RECEIVE)
        =>
        (retract ?f)
        (receive_message)
)



(defrule build-message
        (declare (salience 30))
        (VALUE idt input ?no VALUE ?class ?attribute)
        (?action VALUE ?class ?attribute ?instance $?value)
        =>
        (idt_assert ?no (?action VALUE ?class ?attribute ?instance $?value))
        (assert (SEND FRAME idt ?no))
)

(defrule send-message
        (declare (salience 20))
        ?f <- (SEND FRAME idt ?no)
        =>
        (retract ?f)
        (idt_end_message ?no)
)

(defrule loop-rule
        (declare (salience 10))
        (not (RECEIVE))
        =>
        (assert (RECEIVE))
)
```

Similar rules send the FRAME header and RELATION slots.

Assertion of (DELETE FRAME idt <IDT id number>) causes the MH to retract the frame and terminate the connection of the IDT specified. This fact must be asserted for an IDT to exit prior to receipt of (KILL) without causing an error. Assertion of (KILL) causes the MH to distribute this fact to all of the connected IDTs and then exit. The IDTs exit upon receipt of this fact.

COMMUNICATION ARCHITECTURE

There are three levels of C modules below the actual IDT in the communication architecture (Fig. 2).

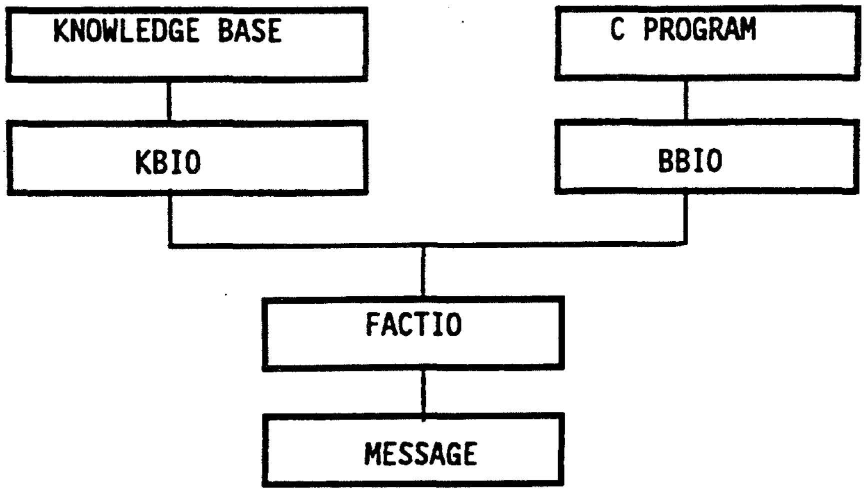


Figure 2: Levels of C Modules in Communication Hierarchy

At the lowest level in the hierarchy is the MESSAGE module which implements transmission of information between distributed processes using UNIX sockets. This module takes care of mapping the logical name supplied by a process into a network address, creating and binding the socket to this address, establishing multiple connections to a single socket, and receiving facts from distributed processes in first-in-first-out order. The next level in the hierarchy is the FACTIO module which implements reading and writing of the elements in a CLIPS facts. This module hides the representation and means of transmission of the fact. The next level in the hierarchy depends on the language in which the IDT is written. CLIPS knowledge bases use KBIO, while C programs (ie. CAD system, User Interface) use BBIO. Both modules implement establishing a two way connection between the MH and an IDT, and the hashing and unhashing of the static fields of frame slots. The KBIO module allows facts to be transmitted using the same syntax as the CLIPS 'assert' command. The BBIO module allows facts in the frame format to be transmitted with a single C function call.

CONCLUSION

ICADS DEMO1 is currently very stable. However, for the system to become usable in a professional setting, the response time needs to be much faster. Presently, the response time is slow because of the large size of the knowledge bases. The response time could be increased by dividing the large IDTs into multiple rule sets, and adding an expert system to coordinate them. The communications framework supports this creation of multiple hierarchies of expert systems.

An IDT should be divided into rule sets that are as independent of each other as possible. This will minimize the transmission and subsequent assertion of local facts between the sub-IDTs. In addition, one slow sub-IDT will not affect the calculation of results from the other sub-IDTs. Optimumly, the facts produced by the sub-IDTs will be blackboard values to be passed directly from the coordinating IDT back to the Blackboard Message Handler.

The IDT would control its sub-IDTs using the same technique as the Blackboard Message Handler. The multiple rule sets would be coordinated by their own message handler. All communication among the rule sets would go through this message handler. Only this message handler would be connected to the Blackboard Message Handler, allowing the IDT to continue to be treated as a single connected component.

Based on run-time profiles of ICADS DEMO1, the percentage of time spent in communication (5 percent) is insignificant compared to the percentage of time spent managing expert system execution (75 percent). The functions which are taking the highest percentage of time are join_compute, find_id, and request_block. The execution time of all these functions would decrease with smaller rule sets. The savings gained from dividing large knowledge bases outweighs the added overhead for the necessary communication.

The slowest and thus the most logical system to divide is the Conflict Resolver. This knowledge base is the largest with over 250 rules. It would be divided into three relatively independent rule sets: no conflict, direct conflict, and indirect conflict. The no conflict division would have rules to post a blackboard value which only one IDT produces. The direct conflict division would have rules to decide the blackboard value based on suggestions for that value from more than one IDT. The indirect conflict division would have rules to infer a blackboard value from a set of other blackboard values. The coordinating expert system for these divisions would be implemented using the same rules contained in the Blackboard Message Handler.

The Conflict Resolver is the largest and most complex knowledge base, and thus would need to be divided first. However, in the future, each IDT will be expanded to produce more in depth analysis and simulation, and thus become larger and slower. When this time comes, these expanded IDTs will also need to be divided.

REFERENCES

1. Pohl, J., L. Myers, A. Chapman, J. Cotton (1989); ICADS: Working Model Version 1; Tech. Report, CADRU-03-89, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, California.

2. Kernighan, B. and R. Pike (1984); The UNIX Programming Environment; PrenticeHall.

3. Assal, H. and L. Myers (1990); An Implementation of a Framebased Representation in CLIPS; Proc. First CLIPS Users Conference, Houston, Texas.

B12 Session:
Enhancements to CLIPS – Graphics/X-Windows

S31-61

N96-12944

358769

p- 10

Integrating Commercial off-the-shelf (COTS) Graphics and Extended Memory packages with CLIPS

Andres C. Callegari

Computer Sciences Corporation
16511 Space Center Blvd.
Houston, Texas 77508

Abstract

This paper addresses the question of how to mix CLIPS with graphics and how to overcome PC's memory limitations by using the extended memory available in the computer. By adding graphics and extended memory capabilities, CLIPS can be converted into a complete and powerful system development tool, on the most economical and popular computer platform. New models of Pcs have amazing processing capabilities and graphic resolutions that cannot be ignored and should be used to the fullest of their resources. CLIPS is a powerful expert system development tool, but it cannot be complete without the support of a graphics package needed to create user interfaces and general purpose graphics, or without enough memory to handle large knowledge bases. Now, a well known limitation on the PCs is the usage of real memory which limits CLIPS to use only 640 Kb of real memory, but now that problem can be solved by developing a version of CLIPS that uses extended memory. The user has access of up to 16 MB of memory on 80286 based computers and, practically, all the available memory (4 GB) on computers that use the 80386 processor. So if we give CLIPS a self-configuring graphics package that will automatically detect the graphics hardware and pointing device present in the computer, and we add the availability of the extended memory that exists in the computer (with no special hardware needed), the user will be able to create more powerful systems at a fraction of the cost and on the most popular, portable, and economic platform available such as the PC platform.

I. Introduction

Programmers who use CLIPS (C Language Integrated Production System) to design large PC applications with or without graphics have encountered the problem of being left with insufficient memory to run the application in a guaranteed and productive way.

This memory problem does not come as a surprise considering that DOS normally uses 640 KB of RAM to allocate the operating system, drivers, buffers, TSRs (Terminate and Stay Resident programs), and for loading and executing programs. DOS memory limitation constitutes a barrier that impedes applications to use the full potential of CLIPS and the standard features of the new generation of PCs such as: extended memory, higher resolution graphics cards and displays, etc. It is important to realize that graphics and image manipulation are usually memory intensive, and that CLIPS memory requirement varies according to the size of the knowledge base used.

Now, PC's are the most popular, portable, accessible, and every time more powerful computer platform available, and it will be a shame that having such excellent hardware power and software development tools such as CLIPS and high quality off-the-shelf software packages, there should still be problems using or developing large PC's applications.

Fortunately for us, two events have happened. The first event is that one of CLIPS blueprint goals was to create a highly portable and low-cost expert system tool that could be easily combined with external systems. This goal facilitates the integration of CLIPS with any external software package(s). The second event is the fact that the PC software market has been flooded with high quality off-the-shelf software packages. These software packages has been written for almost every need, and by using the right combination of software tools (off-the-shelf graphics packages and

DOS memory extenders packages), the problems of using and creating large applications utilizing CLIPS with or without graphics can be solved.

II. CLIPS Problem Areas

The solutions to graphics and memory problems that arise when developing large applications using CLIPS and applications mixing CLIPS with off-the-shelf graphics packages can be grouped into three main areas: CLIPS and extended memory, CLIPS and graphics, and CLIPS using graphics and extended memory. One of several ways to solve each of these problems will be analyzed next. All these solutions have been implemented, used, and tested in an application or demo.

1. CLIPS and Extended Memory

Many PC programmers using CLIPS to build their applications find that they run out of memory while designing, testing, or executing their programs. Once this problem occurs, the only thing left is to restructure the application in order to optimize memory usage. But, as painful as it sounds, sometimes there is no way around. Sometimes, the knowledge base becomes too big, and CLIPS will not have enough memory to operate. In most cases, the only solution is to have access to more memory, but, luckily for us, there are straight and easy solutions for these kind of problems.

1.1 Using extended memory

On the PCs, CLIPS runs on computers using the old Intel 8086 chip as CPU or using a chip which can emulate the operation of this chip. When a program runs on a chip using this emulation mode, it is said that the program is running in **real mode**. Now, the new family of Intel chips (80286, 80386, and 80486 chips) were designed with two working modes (dual-mode chips). The first mode provided full compatibility with older chips so that existing programs will still run in the new computers. The second mode was designed to give on-chip memory management, task management, and protection tools to new and more powerful operating systems (multitasking and multiuser operating systems). When a program runs in the second mode, it is said that the program is running in **protected mode**. Rules for programs running in protected mode are more strict than those programs running in real mode. Protected mode was designed to support multitasking and multiuser systems, so direct access to the hardware and to the operating system has to be restricted in order to eliminate any possible interference with other running processes or with the operating system itself. A crucial advantage of a program running in protected mode is that it gains access to all the extended memory available in the computer.

Normally, when CLIPS runs in real mode, DOS will provide CLIPS with specific services: input/output, file system management, memory management, processor management, etc. In general, all programs will request any of these services from DOS or will bypass DOS and access the hardware directly.

Now, A.I. Architects, Inc. created a very interesting software package which provides to a program running in protected mode (and, therefore, able to access directly all the extended memory available in the computer) with all the services that DOS normally gives to a program running in real mode. This approach permits programs running on 80286 systems a direct memory addressing of 16 MB with 64 KB segments. On 80386 systems, the program can directly access up to 4 GB, with segment sizes as large as the memory installed in the computer.

1.2 Processing CLIPS

If the A.I. Architects package is installed in our compiler package (there is a large list of compilers and assemblers supported) and CLIPS source code is correctly processed, CLIPS will be able to run in protected mode. With CLIPS being able to run in protected mode, CLIPS will have access to all the extended memory available in the computer (15 MB on 80286 systems and 4 GB on 80386 systems). With access to extended memory, CLIPS will be able to handle large knowledge base systems; moreover, the size of the knowledge base that CLIPS could handle will depend on the amount of extended memory available in the computer.

In general, to make a C, assembly language, or FORTRAN program run in protected mode will normally imply the following steps: compiling or assembling the program (.OBJ), linking with the special patch libraries provided for each compiler brand, and maybe postprocessing it by using a special program which creates the final protected mode executable. After these steps are performed, one should load the kernel and load the program into protected mode by using a special real-mode program called loader, which tells the kernel to manage and load the program into protected mode.

This enhanced version of CLIPS does not have the memory limitations and problems that CLIPS and PC users have suffered for so long. From now on, CLIPS will be able to fully use the extended memory normally available in the new powerful generation of machines found in today's market (machines based on Intels' 80386 and 80486 chips). Powerful and highly productive expert systems can be built at a very low cost, and they will be able to use all the graphics power, portability, low cost, and availability characteristic of PC platform's machines.

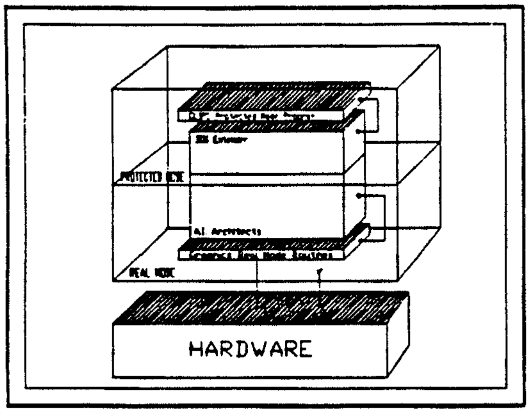


Figure 1. All graphics routines are run in real mode; CLIPS is run in protected mode, and the DOS extender provides the communication links between protected and real mode.

1.3 8086 Emulation

There is another solution which is not as complete as the one discussed before but is very simple to use and to implement. This option is only available for 80386 based systems, the 80386 chip has a virtual V86 mode, which emulates real-mode of an 8086 or 80286 in virtual address space. This emulation permits specially processed executables to run in virtual V86 mode and to use direct addressing in the device space. This approach gives the processed executable a total linear addressing space of 1 MB of RAM. Thus, if CLIPS is properly processed, it will have the capability to directly address up to 1 MB of memory.

One of the biggest advantages of this method is that the operating system, TSRs, and drivers will all run in real mode, so the application has a whole 1 MB linear address space for itself to use. Spawn processes don't take memory from the program, since each spawn process generates a new virtual V86 1 MB linear address space for the new process to use. Another advantage of this mode is that each process runs in a real-mode emulation, which means, they do not have the restrictions imposed by protected mode; they can bypass the operating system and access the hardware directly.

In a few words, a program running in real mode can have only 450 KB or less of free RAM memory left for execution. The operating system, TSRs, buffers, drivers, devices, etc. coexist in the same linear address, while a process running in V86 mode uses practically 1 MB of RAM exclusively for its execution and use.

In order for a program to be successfully processed, it must not used any unsupported DOS calls, and the programs should not be tied to specific physical addresses. The beauty of this solution is that the executable (.exe) can be processed, and there is no need to have the source code.

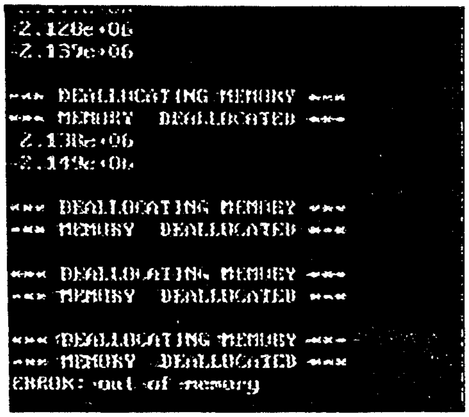
1.4 Performance

Now, the performance of a program running in a 80386 CPU in protected mode is faster than when it is run in real mode. In a 80286 based system, the performance is slightly slower because the 80286 chip needs to be reset (logic reset) every time it switches from real mode to protected mode, and it requires several overhead calls in order to return control to the running program (shutdown logic).

1.5 Restrictions

When a program runs in protected mode, it is subjected to more restrictions. First, the access to physical memory is no longer direct; in this case, indexes to descriptor tables are used instead of addresses. Access to the physical address is made through these descriptor tables when paging is not enable, and the segment register contains a symbolic representation of the address called selector.

A second difference is that memory can not be allocated in an arbitrary way. Third, one can not write to a code segment, and one can not write past the end of a segment. Fourth, a program can not interfere with the operating system. This protection is implemented to keep the operating system in optimal and healthy conditions at all times. These restrictions are necessary because 80286, 80386, and 80486 chips are design to support multitasking and multiuser operating systems.



```
2.128e+06
2.139e+06

*** DEALLOCATING MEMORY ***
*** MEMORY  DEALLOCATED ***
 2.138e+06
 2.149e+06

*** DEALLOCATING MEMORY ***
*** MEMORY  DEALLOCATED ***

*** DEALLOCATING MEMORY ***
*** MEMORY  DEALLOCATED ***

*** DEALLOCATING MEMORY ***
*** MEMORY  DEALLOCATED ***
ERROR: out of memory
```

Figure 2. After running a memory exhaustive test program, CLIPS issued a memory allocation error message after using 2.1 MB. of extended memory.

In Figure 2, there is a picture of a CLIPS program processed so that it can run in protected mode. The CLIPS source program being run from the processed CLIPS executable has been designed to exhaust all the extended memory available in the computer. This test program continuously created CLIPS data forcing CLIPS to request more memory from the operating system until the system run out of memory. The picture shows that CLIPS requested 2.1 MB of memory from the operating system before the system run out of memory.

1.6 Limitations

The EMACS-style editor could not be used. It's code seems to violate some of the restrictions, discussed earlier, imposed over programs running in protected mode. However, one can create a user-defined function to call another editor until a cure is found. A redefinition of the "system" command is necessary. From now on, spawning is reserve for executable files only (.exe) not command files (.com). This means that in order clear the screen, one can not use the command [system "cls"] anymore. The solution is to create a small routine to clear the screen and added to the user defined functions. All of these problems can be fixed in the future, but it is very important to notice that unmodified CLIPS source code is being used and mixed with the A.I. Architects DOS memory extender package.

1.7 Conclusions

The ability of being able to run CLIPS in protected mode and being able to access all the extended memory available in the computer permits the application programmer to create large applications that can handle large knowledge bases. The new generation of PCs based on the Intel 80386 chips have processing speeds near the 8 MIPS mark, and computers based on the 80486 chip have speed around the 15 MIPS bench mark. With CLIPS breaking the PC DOS memory barrier which constrained CLIPS from being used to develop large PC applications, CLIPS will now be able to use all the power and portability of the new PC generations. Now, for example, powerful and complete expert systems can be run in a small but powerful laptop computer, which can be taken and run on practically any possible physical environment. This combination of performance, portability, graphics power, plus the intrinsic capabilities of CLIPS is what CLIPS programmers have been awaiting for. PCs are very powerful and fast but if the operating system can't give programs enough memory to work with, then all the good qualities and power of the PCs are useless. From now on, the situation is different; applications can use large amounts of memory and can use all the new features of the PC's (extended memory, higher resolution graphics cards, mass storage, etc.).

2. CLIPS and Graphics

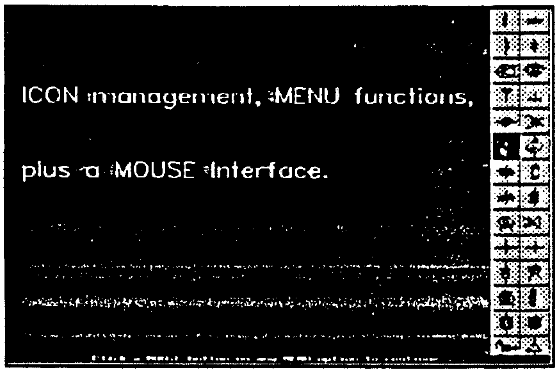
Sometimes an application needs to express some or all of its output information in a graphical form or needs to have a specialized graphical user interface to interact with the user (icon menus, cascade menus, dialog windows, etc.).

In the following paragraphs, two ways of mixing CLIPS with graphics will be discussed. The first method is to mix CLIPS and two graphical packages. In this first case, a driving program controls the execution of the routines. The second method consists in embedding graphics package(s) into CLIPS and to define a complete set of user-defined graphics functions into CLIPS. That is, adding to the original CLIPS language a complete set of graphics commands so that any graphic output or image manipulation process can be performed by issuing commands from these extended language set. Each of this methods have their own advantages and disadvantages.

2.1 Embedding CLIPS (First method)

CLIPS was designed so that it can be embedded within other applications; therefore, when this happens, it needs a driving program which calls CLIPS as a subroutine. This driving program controls CLIPS activation and normally can control most of the graphics output of the application.

CLIPS can interact and interchange data with the driving program in many ways: declaring user-defined functions, passing variables from CLIPS into external functions, passing data from external functions to CLIPS, etc. It is very easy to integrate CLIPS with external functions, which gives CLIPS the capability to execute user-defined graphics commands (C language, etc.) whenever it is needed. In this way, both the controlling program and CLIPS will be able to process, modify, or send graphical information to the screen.



ICON management, MENU functions,

plus a MOUSE Interface.

Figure. 3 This is a menu created by a extended graphics CLIPS command. It displays an icon menu activated by the mouse, and it uses all available extended memory.

2.2 Using off-the-shelf packages

There are many off-the-shelf graphics packages that can be used. Two of them (one from Metagraphics Software Corporation and one from Ithaca Street Software, Inc.) have excellent graphics packages that combined provide the following features: complete graphics environment support, a complete set of utilities for developing multi-window desktop applications, independence over graphics peripherals, icon manipulation routines, plus a complete and powerful set of graphics drawing functions. These features provide most of the tools needed to build any kind of graphical information, graphical objects, and complete user interfaces. These software provide most of the necessary routines needed to build higher user interface tools like pop-up menus, windows, image processing routines (frame animation, etc.), icon manipulation, automatic graphics hardware detection of graphics cards and mouse, etc.

The result of combining CLIPS with the graphics tools provided by these software packages is a **complete and powerful set of software development tools.** Computer Sciences Corporation created for NASA an application which mixes these graphics packages (from Metagraphics Software Corporation and Ithaca Street Software, Inc.) with CLIPS using Borland's C compiler as the blending environment.

Figure 4 shows a screen of the application which was develop using CLIPS 4.3 and the tools provided by the packages described above. A complete user interface (popup menus, Icon menus, help windows, etc.) and automatic hardware detection capabilities were created or provided by the former packages. In addition to this, a set of specialized graphics functions aimed to manipulate graphical objects on the screen were built too.

2.3 CLIPS Graphics Version (Second Method)

In the second method, CLIPS possesses all the graphical capabilities to create and manipulate (using its new set of graphical language commands) any graphical object on the screen: menus, image manipulation, icon manipulation, graphics functions, etc. In Figure 2, 5, and 6, there are examples of applications that use all the extended memory available in the computer and that use a mouse to activate the icon menus. These icon menus were created using the new set of CLIPS graphics commands (icon management and graphics environment provided by Metagraphics Software Corporation and Ithaca Street Software, Inc.). When an option is chosen, a fact specifying the chosen option will be asserted into the CLIPS fact list. Figure 5 gives a demonstration of text management, size, and the different kind of fonts available in the extended graphics CLIPS version.

The advantages of this method is that all programs will be written as part of an extended CLIPS language, they will run in interactive mode (easy to maintain, perform tests, or debug), and they will not need a driving program. The best part is that after modifying the code, there is no need of recompiling or relinking the program. This will give the expert system total and continuous control over the process. Sometimes, if there is a driving program(s), information has to be passed to CLIPS to update any change in the state of the system that happened while the driving program was in control.

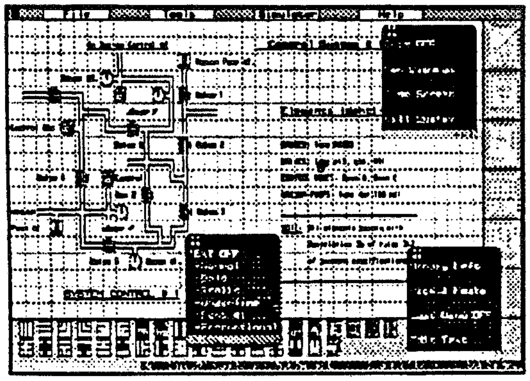
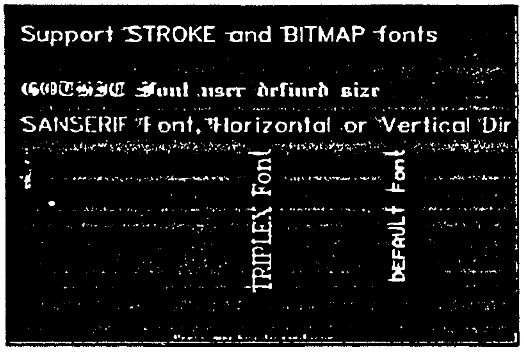


Figure 4. This figure shows the display of an ICAT system created by mixing CLIPS and graphics in a C environment.

Graphics commands behave and are issued exactly like any other CLIPS internal command, and rules containing graphics commands will behave like any other rule does.



Support STROKE and BITMAP fonts

GOTHIC Font user defined size

SANSERIF Font, Horizontal or Vertical Dir

TRIPLEX Font

DEFAULT Font

Figure 5. This figure shows font management and the available fonts (provided by Borlands C compiler) used in the extended memory/graphics CLIPS version.

2.4 Performance

The applications described above were tested on a PC running at 25 Mhz., 100 ns RAM memory, with coprocessor, and with a VGA card/monitor. The applications didn't have any problems in what speed pertains; CLIPS, the user interface, and the graphics responses run smoothly and pleasantly fast.

2.5 Memory Limitations

For systems that use CLIPS and a moderate amount of graphics (does not need a complete user interface or image manipulation routines), these graphics packages will provide the perfect development solution, and the application will almost have the same limitations as a normal CLIPS program (just a little less memory free for CLIPS).

The application in Figure 4 possesses a complete graphical user interface, works on graphical objects, and does a lot of image manipulation. Therefore, it is anticipated that after loading the program, there will not be much memory left for CLIPS to work with. This fact directly implies that there will be a strict limitation in the size of knowledge base that can be loaded and/or used by CLIPS.

If the knowledge base consumes most of the free memory left in the computer, then it will be very probable that CLIPS will run out of memory at run time. This is why an application using CLIPS and intensive graphics can not run in a guaranteed (knowledge base can grow and consume all the memory) and productive way (if the knowledge base is limited to a certain size, the application main goal will be restricted too).

2.6 Conclusions

Thanks to CLIPS special features and design, CLIPS can be easily integrated with off-the-shelf's graphics packages. These added graphics capabilities give CLIPS the power to express output in graphical form, which is needed in a large number of applications (simulations, training, charting, etc.), or in those applications that need a specialized graphical user interface (image manipulation, icon menus, etc.).

For large applications that use CLIPS and intensive graphics manipulations, a second package (DOS memory extender) has to be added. This package will permit CLIPS to run in protected mode and the graphics part to run in real mode. In this way, CLIPS knowledge base can grow as big as it needs and the graphics part of the application will have enough memory to operate without any problems.

3. CLIPS, Graphics, and Extended Memory

3.1 Problems

Developing an application that uses CLIPS in extended memory and graphics involves a deeper understanding of how real mode and protected mode work. First, off-the-shelf graphics packages provide only libraries and object files. Most packages do not provide the source code; therefore, it will not be possible to process the code so that it can run in protected mode. Second, there are software whose code access directly the hardware (direct screen write, etc.); protected mode will not let these programs access the hardware directly.

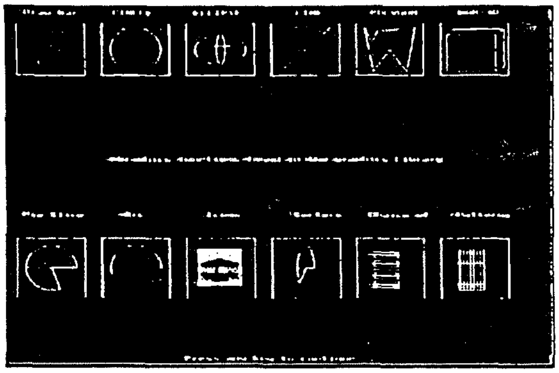


Figure 6. This is a sample of some graphics features available in the extended memory/graphics version of CLIPS.

3.2 Solution

The solution consists in running the hardware dependent routines in real mode, where they can access the hardware directly, and to run all non hardware dependent code in protected mode. A.I. Architects developed mechanisms for interprocessor communication. A routine running in protected mode can pass data to a routine in real mode which will process the data and will return data to the protected mode application. There are two more ways how a real procedure can communicate with a protected-mode application. The real procedure can signal a protected-mode handler, or one can make use of interrupts.

If the application is going to use graphics (graphics run faster if the graphics routines can access the hardware directly), the best solution will be to run all the graphics routines or hardware dependent routines in real mode. The required communication links will be established with the protected mode application so that the application can issue any graphics command. Figures 3, 5, and 6, show screens of an application created by the extended graphics version of CLIPS. This version of CLIPS runs graphics in real mode and runs all other CLIPS routines in protected mode. When CLIPS needs to issue a graphics command, it will make a call to the real procedure and will pass the needed data so that the real procedure can execute the graphics commands. This extended CLIPS version that includes graphics and extended memory was built using packages from Borland, A.I. Architects Inc., Metagraphics Software Corporation, and from Ithaca Street Software Inc.

3.3 Limitations

The transaction buffer size is 4 KB. This buffer is used to pass data when the protected mode application calls a real-mode procedure. One can get around this problem by using interrupts or real procedure signals. Second, the number of real procedures can not exceed 32. Considering that DOS only uses 640 KB of memory, 32 real procedures will be sufficient for most purposes. If the real procedure is called as an overlay, then the CS:IP in the EXE header is needed; therefore, the executable must be an .EXE file not a .COM file.

3.3 Conclusions

In appendix A, there is a list of graphics commands used by the extended version of CLIPS. This graphics version of CLIPS uses extended memory and was created using the packages mentioned in section 3.2. Appendix B contains two rules that cut a portion of an image on the screen and will slide it randmoly around the screen. One excellent feature provided by the off-the-shelf packages is the ability to detect the graphics hardware and pointing device present in the computer. The PC platform has a wide variety of hardware and is difficult to keep track of all the different brands and models. This feature frees the user from the trouble of configuring the application for the particular system in which it will run.

Applications created using this method has the advantage that all programs are written in the extended CLIPS language (expert system, graphics output, and user interface). In order to run different applications, the only executable needed is the extended version of CLIPS. If the application needs to be modified, only the application code needs to be changed without needing to compile or link the source code again. Moreover, all the tools like fonts, drivers, and graphics routines will altogether coxist in one CLIPS executable package. Applications will only consist of source code and data; thus, large amounts of storage space will be saved permiting even PCs with small storage devices to store complete applications.

III. Concluding Remarks

Section 1 shows a way overcome CLIPS memory problems by using extended memory; section 2 shows a way to mix graphics with CLIPS, and section 3 shows a way to use CLIPS in extended memory and how to mix it with graphics. Again, this is only one way to solve these problems, but if you are designing a large application on a PC, you will surely have one of the problems discussed in this paper, so if you have any of these problems and don't have a solution, or you are thinking in designing a large application, this paper provides you with the information needed to solve that problem.

The best outcome of the whole process is that PC's applications, built using CLIPS and graphics, can overcome the 640K memory limitation imposed by DOS, and applications using CLIPS will be able to handle large knowledge bases. This capability allows developers to use the PC platform as an application development and delivery platform, and will permit users to enjoy the power, low cost, portability, and accessibility of the new generation of PCs.

APPENDIX A

Extended CLIPS version (Graphics and Extended Memory)

List of Graphics Commands:

initialize-graphics	draw-sector	write-grstringxy	erase-mode
draw-bar	draw-linerel	write-grstring	set-palette
memory-left	draw-line	set-textstyle	draw-icon-bar
change-bkcolor-to	draw-polygon	set-textjustify	free-mouse-draw
set-viewport	draw-lineto	hide-mouse	cursor-shape
draw-rectangle	pause	show-mouse	draw-menu
clear-viewport	draw-bar3d	mouse-waitchange	save-image-file
clear-screen	status-message	set-usercharsize	read-image-file
draw-ellipse	pause-keyortime	mouse-waitrelease	write-image
draw-circle	fill-pattern	mouse-waitpress	read-image
draw-pieslice	status-message-top	mouse-RightPressed	grid-pick
draw-point	set-linestyle	mouse-LeftPressed	release-image
mouse-in-rectangle	move-mouse	mouse-AnyPressed	set-rectangle
pt-in-rectangle	random-number	mouse-MiddlePressed	close-graphics
set-active-page	move-cursorrel	mouse-positiony	set-nosound
limit-mouse	move-cursor	mouse-positionx	set-sound
set-draw-mode	set-visual-page	max-screenx	spawn-process
key-pressed	get-scankey	clear-text	draw-textbox
max-screeny	Initialize-Animation	close-animation	bell
ed	rectangle-animation	text-attribute	text-xy
draw-arc	set-usercolorpattern	mouse-visible	set-delay
set-timer			

APPENDIX B

Extended CLIPS version (Sample Source Code)

These are two rules that when fire, they will cut an image from the screen and will move it randomly around the screen. The move will be performed in the specified number of steps. Before running the program, the command "**initialize-graphics**" has to be run interactively from CLIPS or added to the CLIPS initialization rule.

```
;
;   This Rule Reads an image from the screen and initializes the animation procedure.
;

(defrule copy-rectangle
  (initial-fact)
=>
  (limit-mouse 0 0 639 479)                ; Limit mouse movements to specified box
  (read-image 251 251 349 349 1)           ; Read image in specified box (cut)
  (clear-screen)                           ; Clear the graphics screen
  (write-image 255 255 1)                  ; Write image to specified position (overwrite)
  (release-image)                          ; Free image resources
  (Initialize-Animation 251 251 349 349)   ; Initialize animation for given box.
  (clear-screen)
  (assert(count 80)
         (animation-start)
         (do animation)
         (coord 251 251)
         (rectangle-start)))


;
;   This rule moves the image ramdomly around the screen 80 times.
;

(defrule do-animation
  ?one<-(do animation)                     ; Begin process
  ?two<-(coord ?a ?b)                      ; Retrieve actual coordinates
  ?three<-(count ?cc)                      ; How many times more do we need to fire this rule
  (test (> ?cc  0))                        ; Stop moving..?
  (test (key-pressed))                     ; Test if key was pressed, if pressed don't fire rule.
=>
  (retract ?one ?two ?three)
  (bind ?x (random-number 539))
  (bind ?y (random-number 379))            ; Create random numbers within the screen size
  (bind ?w (+ ?a 98))
  (bind ?z (+ ?b 98))
  (bind ?cc (- ?cc 1))                     ; Working variables.
  (rectangle-animation ?a ?b ?w ?z 5 ?x ?y 0)  ; Move image using given arguments
  (assert (do animation)
  (coord ?x ?y)(count ?cc)))               ; Repeat until count is reached.
```

532-61

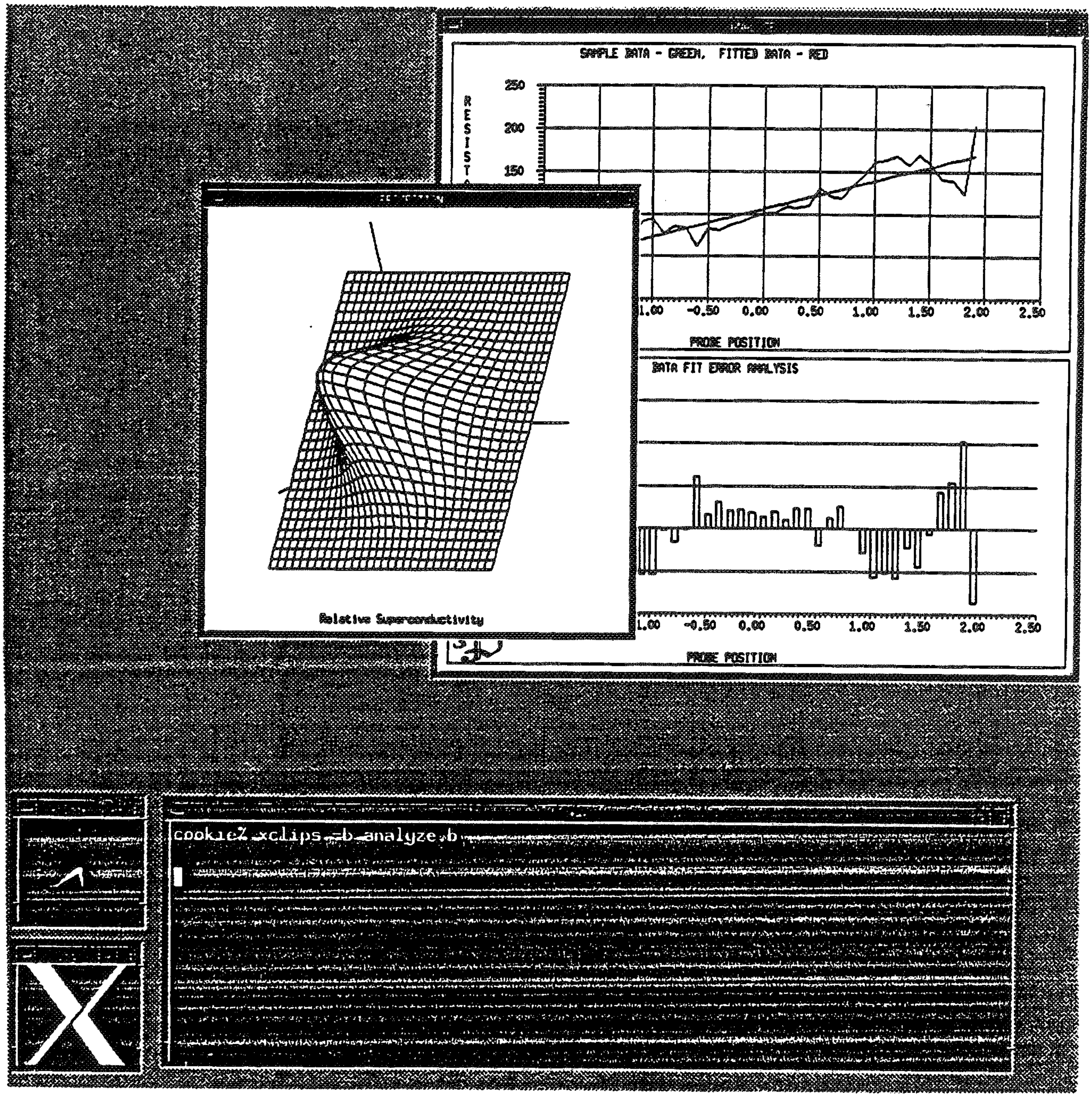
N96-12945

p. 25

358781

Constructing Complex Graphics Applications With CLIPS and the X Window System

Ben M. Faul
TRW Defense Systems Group
Carson, CA 90746



```
cookie% xclips -b analyze.b
```

1. ABSTRACT

This article will demonstrate how the artificial intelligence concepts in CLIPS used to solve problems encountered in the design and implementation of graphics applications within the UNIX-X Window System environment. The design of an extended version of CLIPS, called XCLIPS, is presented to show how the X Window System graphics can be incorporated without losing DOS compatibility.

Using XCLIPS, a sample scientific application is built that applies solving capabilities of both two and three dimensional graphics presentations in conjunction with the standard CLIPS features.

2. INTRODUCTION

The CLIPS language provides most of the control functions required for building expert systems. Two areas of the language identified that could use improvement are in the areas of advanced graphics presentation and data analysis functions.

To apply CLIPS to the solution of very complex scientific or business applications, the language requires extensions to handle extended data analysis and graphics presentations problems normally encountered in these systems.

In designing extensions to the CLIPS system to handle these kinds of problems, a survey of several scientific and presentation graphics systems was done to determine the new features.

The survey of these other systems yielded the following capabilities that would be most desirable in the extended CLIPS expert system shell:

Inter-process communications - Many problems are better solved by the ability to use a server/client architecture.

2D & 3D Charting/Graphics - A picture is worth a thousand words.

Printing of Charts/Graphics - Hard-copy is needed, in order to publish the charts and graphs.

Data Smoothing - Reduces noise in a set of experimental data.

Curve Fitting - Polynomial and cubic splines curve fitting to a set of values.

Simultaneous Equations - Solves systems of linear equations.

The remainder of this document describes the philosophy of how CLIPS was extended to incorporate these new features and how well the resultant XCLIPS performs in solving a non-trivial problem.

3. APPROACH

The design approach for extending the CLIPS language involves two distinct tasks.

The second first involved designing a graphics system for XCLIPS to use. While the X Window System was chosen as the graphics sub-system, linking XCLIPS directly to X would obviate the expert systems from ever being used on DOS; the X Window System is not available on DOS, nor is it ever likely to be available on DOS.

The second task involved linking the XCLIPS language to the data analysis algorithms and graphics sub-system. For the most part, interfacing XCLIPS to these sub-systems follows the method defined in section 2, in the CLIPS 4.3 Advanced Programming Guide. However, the interface to some of the data analysis functions requires the use of vectors and matrices as parameters. Because the data types of standard CLIPS are not convenient for representing matrices, the language had to be extended in a non-standard manner.

4. DATA ACQUISITION AND MANIPULATION

Advanced data handling capabilities required by XCLIPS fall into two categories: inter-process communications and the mathematicics based tasks such as curve-fitting and the like.

4.1 Inter-process Communications Data

Many applications are better implemented as separate cooperating entities - using a server/client architecture.

A familiar server/client architecture may be found in large database management systems. Typically, the only program that actually updates the database is the "server" process. The user "client" programs communicate their requests for processing to the server, that handles the requests and returns the appropriate responses. In such a way, access to the database is maintained through a single process.

In the XCLIPS system, the inter-process mechanism used is the TCP protocol. Using TCP, an XCLIPS program may communicate directly to any process within the same machine, or any process on any machine that the user can access via the local or wide-area network.

In order to open a communications path between programs, the caller and receiving programs first have to be ready to make connections. The two programs that will be communicating agree beforehand which communications channel (or "socket") will handle the call. The program that will be called prepares to receive by making a function call to place the socket into the "accepting state". The function that places the program into the accepting state returns immediately with status indicating whether any other program is ready to communicate. In this way, the program can continue processing, without the need for waiting for a connection to complete.

Periodically the accepting program checks the status of the socket to find out if a connection has been accepted.

The program that wishes to place a communications call to another program specifies the address of the program to be called. This address consists of the Internet name and socket number. This action puts the calling program in the "opening state".

When the opening program makes the function call to open the communications socket, the function waits for the call to complete before returning; however, if the call does not complete within 5 seconds the call returns with an error.

When the open completes, the opening program gets a return code indicating success. Also, the called program, which is periodically checking the socket for a completed call likewise gets a return code indicating that the communications channel is now open for communications.

Once established, bi-directional communications is as easy as reading and writing to a file.

To promote efficiency, when a socket is read by a rule in XCLIPS, the socket read returns immediately, whether or not data is actually available. The socket read call returns the number of bytes it read. When the return code is greater than zero, data is ready to be processed.

To demonstrate how easy implementing inter-process communications within XCLIPS programs can be, consider the following rules for sending and receiving messages across a network.

In the rules defined below, the process listens on socket 3000; when successful, the socket descriptor 1 is used for reading a message from the network and then printing it on the terminal.

```
(defrule listen "Listen for network open"
        (not (socket opened))
        =>
        (if (> 0 (NetAccept 3000 1)
        then
                (assert (socket open)))

(defrule read-socket "If data in socket, then print"
        (socket open)
        =>
        (bind ?string (NetRead 1))
        (if (neq ?string "")
        =>
                (printout t ?string t)))
```

In the following rules the process opens a connection to a process on machine "shasta" at socket

3000 (the previously described rules). Once the connection is open the "write-socket" rule reads from the terminal and sends the message to the other process on machine shasta.

The "read-socket" rule of the other process reads the data sent by the "write-socket" rule and then prints this data on the terminal.

```
(defrule setup "Setup the call"
        (not (call setup))
        =>
        (GetHostByName "shasta")
        (assert (call setup)))

(defrule check-socket "Check socket for open success"
        (not (socket open))
        (call setup)
        =>
        (if (> 0 (NetOpen 3000 1))
        then
                (assert (socket open)))

(defrule write-socket "Write to socket"
        (socket open)
        =>
        (NetWrite (read) 1)))
```

These two programs may be on the same machine, on different machines on the same local-area network, or on different machines separated across the world on a wide-area network.

4.2 Language Interface For Inter-process Communications

In the UNIX environment, the programmatic interface to the TCP layer is done through file descriptors. However, in a DOS system, TCP sockets are separate from the file descriptors. Because this bifurcation of file/socket descriptors is a given on DOS, in the spirit of keeping DOS and UNIX versions of XCLIPS equivalent, this bifurcation of file/socket descriptors is retained in the UNIX version. Note that while file I/O on both DOS and UNIX is of the blocking variety, Network I/O on XCLIPS is of the non-blocking type.

As can be seen in the XCLIPS programs of the previous section there are several new language constructs introduced. Actually, this network capability is accomplished by the introduction of only five new commands to the language.

(GetHostByName) - Identifies the program to be called, by its Internet address.

(NetAccept) - Place a specified socket in the "accept" state. (Listen for a call.)

(NetOpen) - Place a specifies socket in the "opening" state. (Place a call.)

(NetWrite) - Send data to the other program.

(NetRead) - Receive data from the other program.

The ability of XCLIPS rules to communicate across a network, in a transparent, real-time fashion opens up new vistas for CLIPS applications.

4.3 Mathematical Data

The XCLIPS language includes many functions (over 75) for easily handling and analyzing large volumes of data. Section 2 of this document details the kinds of functions available for data analysis.

4.4 Language Interface For Data Analysis

All of the data analysis functions involve operations on floating point arrays or matrices. While CLIPS has a vector data type, it is not suitable for handling large amounts of data, nor are these vectors shareable across rules.

To accommodate easier handling of single and two dimensional arrays, as well as for the ability to share this kind of data across rules, two new data types are introduced -- Vector (single dimension) and Matrix (two dimensions). These new data types are accessed by name as string variables. The new data types have their own actions for assigning and evaluating elements.

As representative of the class of data analysis functions available in XCLIPS, the curve-fitting functions are briefly discussed below:

In the curve-fitting section of the XCLIPS language there are three functions available.

(PolyCurveFit) is a function that fits a polynomial with linear coefficients to a dependent - independent variable set of data.

(CubicSplines) is a function that fits a set of polynomial equations to a discrete set of data.

(CalcSpline) is a function that will calculate the cubic spline interpolation of a y-given value given an x-value of the cubic splines coefficient matrix calculated by the function CubicSplines.

These curve fitting functions are representative of the power and flexibility of the functions available within XCLIPS. For sample uses of these functions refer to Section 6.4.

5. GRAPHICS

5.1 Two Dimensional Graphics

The XCLIPS language provides both plot and chart graphics, as well as object oriented drawing. Graphical representations are often the best method for conveying information derived from a mathematical analysis; the pictorial representation of a sine wave carries more information to the reader than an equation or columns of numbers.

There are approximately fifty functions available for 2D graphics. The following table details the kinds of features available. The extensions were written in a machine-independent manner, all of these graphics functions are available under both DOS and UNIX versions of XCLIPS.

Automatic Axes and Scaling
Automatic Grid Drawing
Line Plotting
Bar Plotting
Contour Plotting
Pie Charting
Patterning
Text Printing
World <-> Real Coordinate Translations
Color Selection
Object Oriented Drawing

5.2 Three Dimensional Graphics

Building on the 2D graphics capabilities, XCLIPS implements 3D projections using 2D functions. There are thirty 3D graphics functions available in XCLIPS. The following table summarizes the capabilities available in the language:

World <-> Actual Coordinate Translation
Concatenation
3D Rotation
Perspective Selection
3D Scaling
Color Selection
Solid Drawings

5.3 Language Interface For Graphics

To facilitate an XCLIPS product portable to both DOS and UNIX, XCLIPS uses an arbitrarily defined that is neither specific to DOS or UNIX. The XCLIPS language interfaces to this arbitrary window system. In this manner, the language is independent of the native DOS graphics or the X Window System based graphics. The graphics commands include both low-level (draw line, point, etc.) to very high-level (auto-axes generation, draw contour, draw 3D in 2D projection, draw object and the like) commands.

5.4 Definition of XCLIPS Windows

The window system used internally by XCLIPS is an arbitrary one designed to be portable to both the DOS and UNIX operating systems.

The DOS version of XCLIPS works on Pcs using CGA, EGA, VGA, and Hercules graphics cards. The XCLIPS programs are independent of the graphics card used in the PC. Of course, color application's output is converted to black and white on monochrome displays; nonetheless the XCLIPS application still run. On the DOS screen up to 10 "windows" may be created by the application. Each of these windows is separately accessible by the XCLIPS program. The windows may or may not overlap as the programmer desires. These windows are accessed with a "world" coordinate system, defined by the user program.

In the UNIX-X Window environment, XCLIPS creates an X window that corresponds to the DOS screen. Within this X window, up to 256 sub-windows (instead of 10) may be created by the XCLIPS program. If the user program desires, the resolution of the XCLIPS window may correspond to a resolution found under DOS on CGA, EGA, VGA or Hercules graphics adapters. However, the XCLIPS program may select a base window to be of any size that the X Window System display can support. The UNIX based XCLIPS program uses the same base color scheme as the DOS system uses. However, the XCLIPS program may utilize all the colors available to the X Server, if the developer so desires; but, such programs are not backwards compatible under DOS. Even though the UNIX based extended XCLIPS has higher resolutions, more colors, virtual memory in its favor, the XCLIPS programs will still run under DOS, subject to DOS's memory restrictions.

To demonstrate how well the arbitrary window interface works, consider the following XCLIPS rules that describes a wire-frame house in a 3D perspective as displayed on a DOS screen and a UNIX X Window.

```
(defrule main "Initialize the system"
        (not (system initialized))
        =>
        (tInit3)
        (Init3D 6)
        (SetWorldCoordinates -10 -10)
        (SelectColor 3)
        (WorldScale 1 2)
        (WorldRotate3 10 0 1)
        (Persp 15)
        (assert (system initialized))
        (assert (draw house)))
```

```
(defrule draw-house "Draw the wire-frame house"
        ?rem <- (draw house)
        =>
        (retract ?rem)
        (SelectColor 15)
        (Move3Abs 1)
        (Line3Abs 1 -1)
        (Line3Abs 1 -1 -1)
; right side
        (Line3Abs 1 -1)
        (Line3Abs 1)
        (Move3Abs -1)
        (Line3Abs -1 -1 -1)
; left side
        (Line3Abs -1 -1)
        (Line3Abs -1)
        (Move3Abs 1)
; front top
        (Line3Abs -1)
        (Move3Abs 1 -1)
        (Line3Abs -1 -1)
; front bottom
        (Move3Abs 1 -1)
; back top
        (Line3Abs -1 -1)
        (Move3Abs 1 -1 -1)
; back bottom
        (Line3Abs -1 -1 -1)
        (Move3Abs 1)
        (Line3Abs 0 1.5 1)
; roof
        (Line3Abs -1)
        (Move3Abs 1 -1)
        (Line3Abs 0 1.5 -1)
        (Line3Abs -1 -1)
        (Move3Abs 0 1.5 1)
        (Line3Abs 0 1.5 -1))
```

The following two figures show the results of the XCLIPS programs running under both DOS and UNIX. In Figure 1, the DOS screen is displayed. In Figure 2, the UNIX screen is shown, running the exact same program.

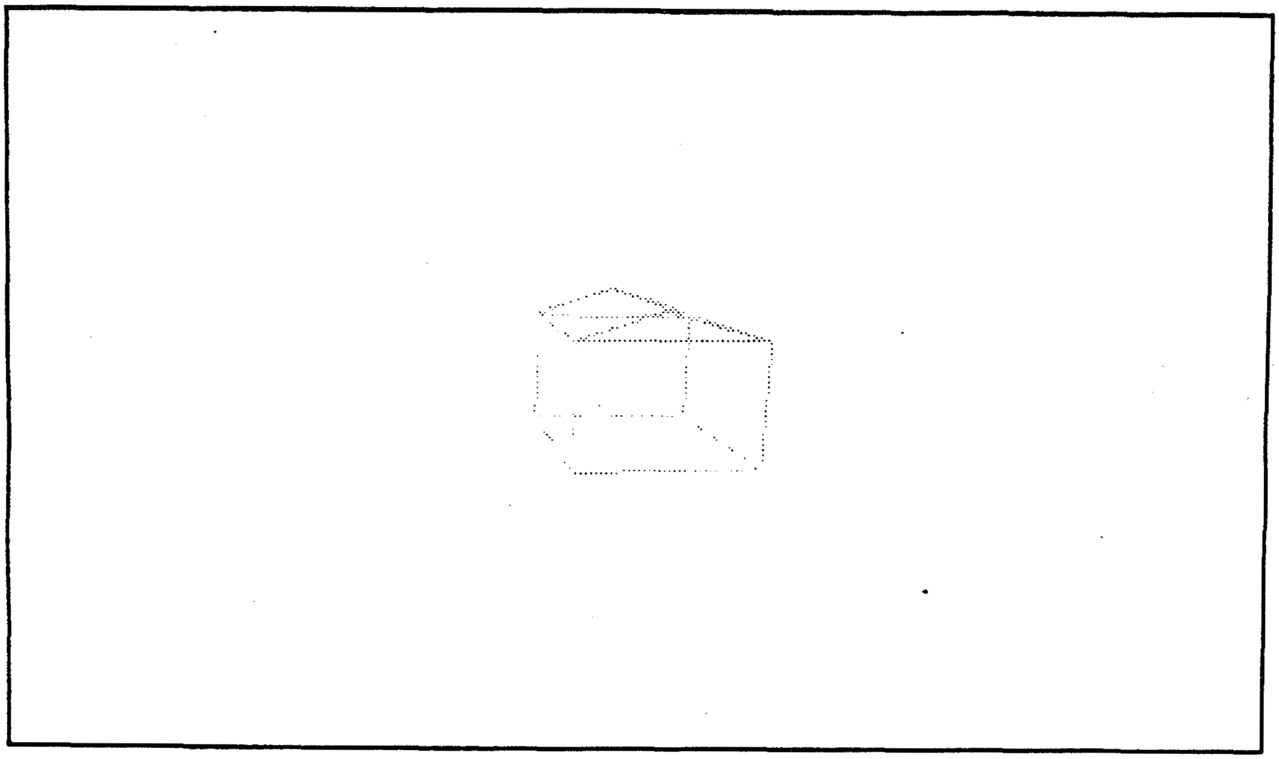
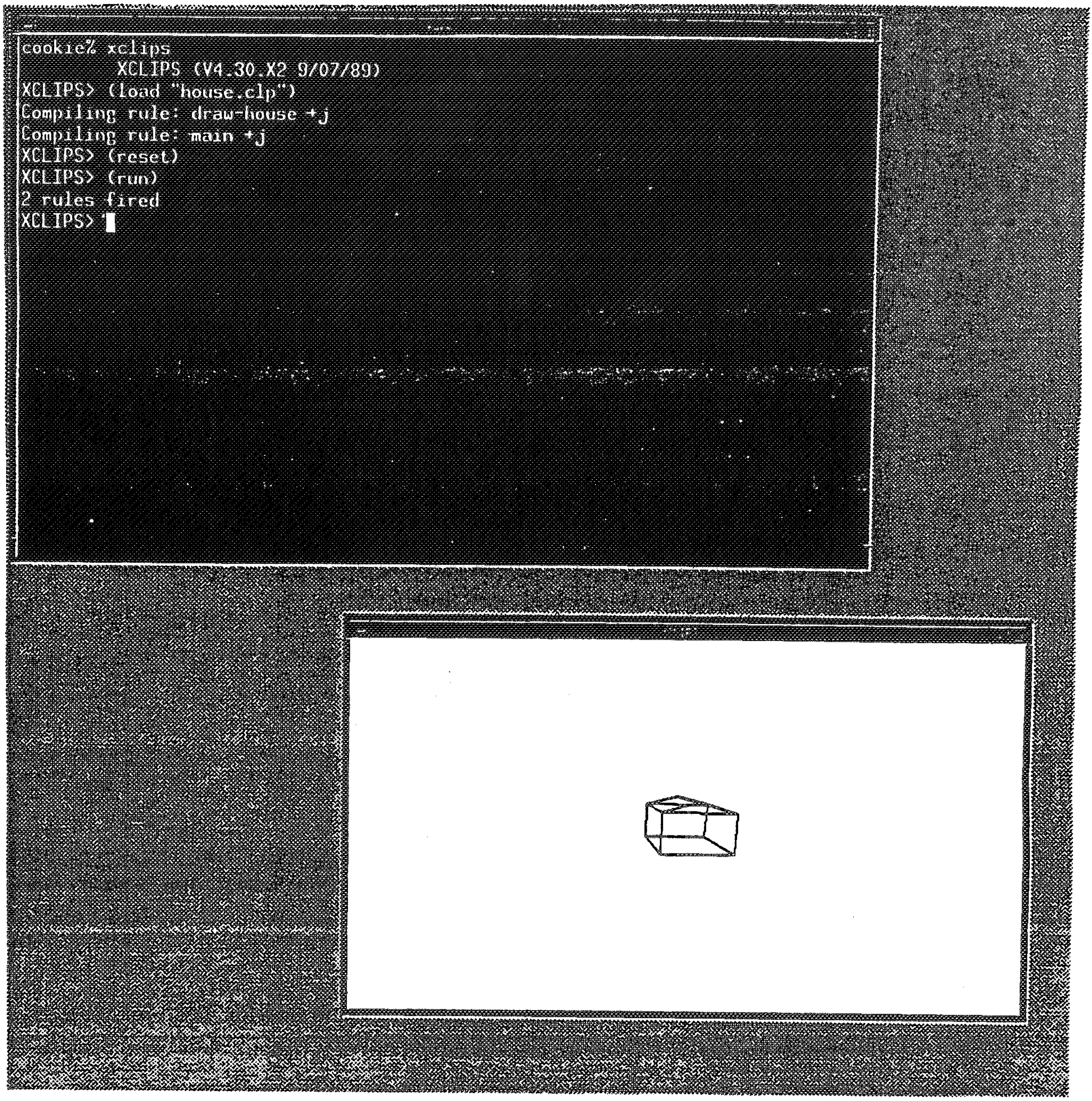


Figure 1, Screen Dump of DOS Version of "3D House".



```
cookie% xclips
         XCLIPS (V4.30.X2 9/07/89)
XCLIPS> (load "house.clp")
Compiling rule: draw-house +j
Compiling rule: main +j
XCLIPS> (reset)
XCLIPS> (run)
2 rules fired
XCLIPS> 
```

Figure 2, UNIX Version of "3D House".

As Figures 1 and 2 show, by using an arbitrary windowing system the XCLIPS programs are easily made machine and operating system independent.

5.5 Printing

Users of graphics systems need hard-copy output as well as screen outputs. Since a range of printers would be used by any given set of users, XCLIPS supports some of the more popular printers. The language interface to the printer drivers is via a single call, with parameters used to inform XCLIPS which printer is selected, where to spool the output, and landscape/portrait modes.

Table 1. Printers Supported by XCLIPS
EPSON MX, FX, LQ Dot Matrix Dot Matrix Printers.
Toshiba Dot Matrix Printers
HP Laser Jet Printers
Plotters implementing Hewlett-Packard Graphics Language (HPGL).

A future enhancement will include Post-Script support, as this output format is readily becoming the standard for publishing.

5.6 XCLIPS Interface to the UNIX X Window System

The graphics sub-system used by XCLIPS is the X Window System. Because the X Window System is divided into two distinct parts, with all of the device dependent code isolated in the server, XCLIPS is inherently machine-independent.

XCLIPS utilizes the Xlib programming library for all its graphics requirements. Xlib provides all of the primitive graphics capabilities needed by XCLIPS; however, since Xlib calls are very low-level, a separate library called "seglib" was created that supplies high level functions to the XCLIPS language, such as auto-axes, bar and pie charting, etc., that are of more interest to the expert system user.

Seglib is organized using a layered approach, making it usable with the Microsoft "graph.lib" library under MS-DOS and on UNIX under the X Window System. XCLIPS expert systems utilizing graphics capabilities work without modification on either UNIX (using X) or DOS (using standard DOS graphics).

In Figure 3, the operating system independent graphics architecture of XCLIPS is described. Notice that the top layer, the XCLIPS language interface and high-level graphics, is common across both the DOS and UNIX versions of XCLIPS. The middle layer, also common to both DOS and UNIX versions, is an interface to the machine dependent graphics layer (bottom layer). The middle layer is divided into to two parts. The top half of the middle layer is an arbitrary graphics system that communicates to a graphics library with a compatible calling sequence to Microsoft's "GRAPH.LIB" library. The bottom half of the middle layer is an implementation specific module depending on which operating system is being utilized. On DOS, this bottom half is merely a coupling to GRAPH.LIB. On UNIX this bottom half is a module that translates GRAPH.LIB calls to X-Window System Xlib calls.

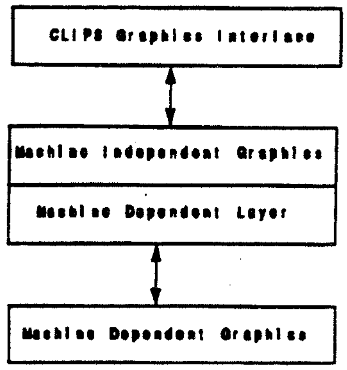


Figure 3
Layered Graphics Architecture

The bottom layer is operating system specific. On DOS, this layer is merely the Microsoft graph.lib library. On UNIX, this bottom layer is the Xlib X Window System library. Using this architectural approach, XCLIPS remains true to its operating system independence heritage.

Note that this DOS to X Window System library interface at level-2 has the potential to allow DOS programs written to the Microsoft library to be easily ported to the UNIX/X Window System interfaces.

6. XCLIPS APPLIED TO A REAL PROBLEM

6.1 Problem Definition

To demonstrate the effectiveness of applying non-procedural languages, such as XCLIPS, to solving graphics related problems, a fairly sophisticated application is described, and then implemented. This XCLIPS system will demonstrate the uses of the 2D and 3D graphics described in Sections 6.1 and 6.2, the inter-process communications mechanisms described in Section 5.1, and the curve-fitting data analysis functions described in Section 6.3.

The demonstration system will be a simulated resistance/superconductivity analysis station. This analysis station will have the following features:

Table 2. Capabilities of the Superconductor Analysis Station

Obtains resistance samples from the surface of the object being analyzed.
In a split window display the resistance samples on the object. This display will show the actual samples along with a prediction of the resistance curve, and a running computation of the prediction confidence.
In a separate window, display a 3D projection of the resistance found across the object's surface. The X and Y axis represent the surface of the object and the Z axis represents the resistance at that point on the object.

6.2 System Architecture

There will be two instances of XCLIPS running on the UNIX computer that communicate via inter-process communications mechanisms.

The first XCLIPS system is called "ANALYZE". This expert system is responsible for communicating with the resistance probe, over a TCP socket. ANALYZE will also handle all computations involving data gathering and analysis (curve fitting) as well as all 2D graphs). Also, ANALYZE will handle any user input

The second XCLIPS system is called "PROJECTION". This expert system is responsible for generating the 3D projection of the object's resistance. The data for the 3D projection will be a contour map. This contour map is send to PROJECTION by the ANALYZE expert system.

The following figure describes the processing architecture of the complete resistance analysis work-station.

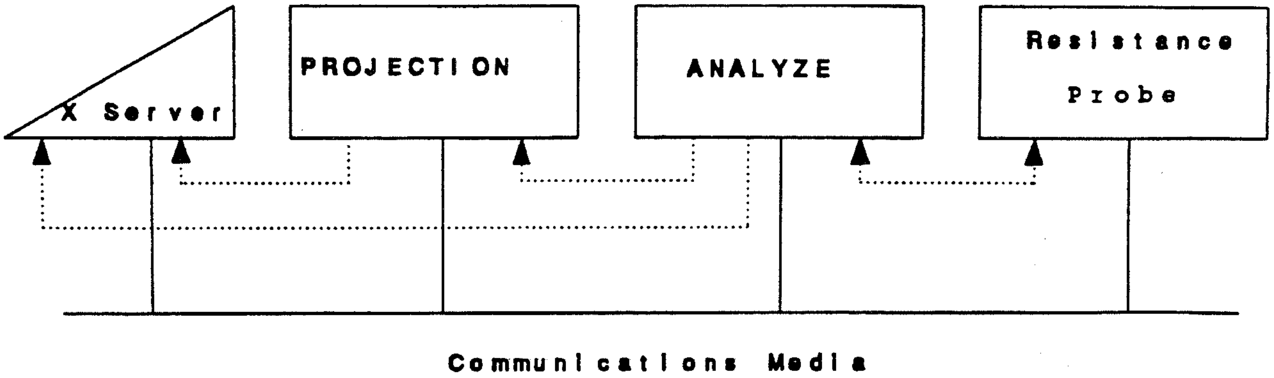


Figure 4, Superconductor Analysis Work Station Architecture

As can be seen in Figure 4, the central server is the process called ANALYZE. ANALYZE controls input and output to the probe, and the PROJECTION system. The graphics output of both ANALYZE and PROJECTION is sent to an X Server.

Due to its inherent server/process architecture, all the processes could be on the same machine, or each process could be on a different machine across a network. This transparent distributed architecture is flexible, without burdening the user with having to know the specifics of how the system operates.

When the analyst starts the session, the ANALYZE expert system is started. ANALYZE then establishes connections with the resistance probe. Once the connections are successfully started, ANALYZE then requests user input as to where to place the probe. Once ANALYZE has the coordinates to analyze it sends the appropriate information to the probe. When ANALYZE receives the data from the probe it displays the information in its 2D windows as line-plots. When the analyst wants to view a 3D projection of the object's resistance the user pushes an icon with the mouse button. ANALYZE then starts the PROJECTION expert system, establishes a TCP connection with it, and passes the data to be displayed as a contour map.

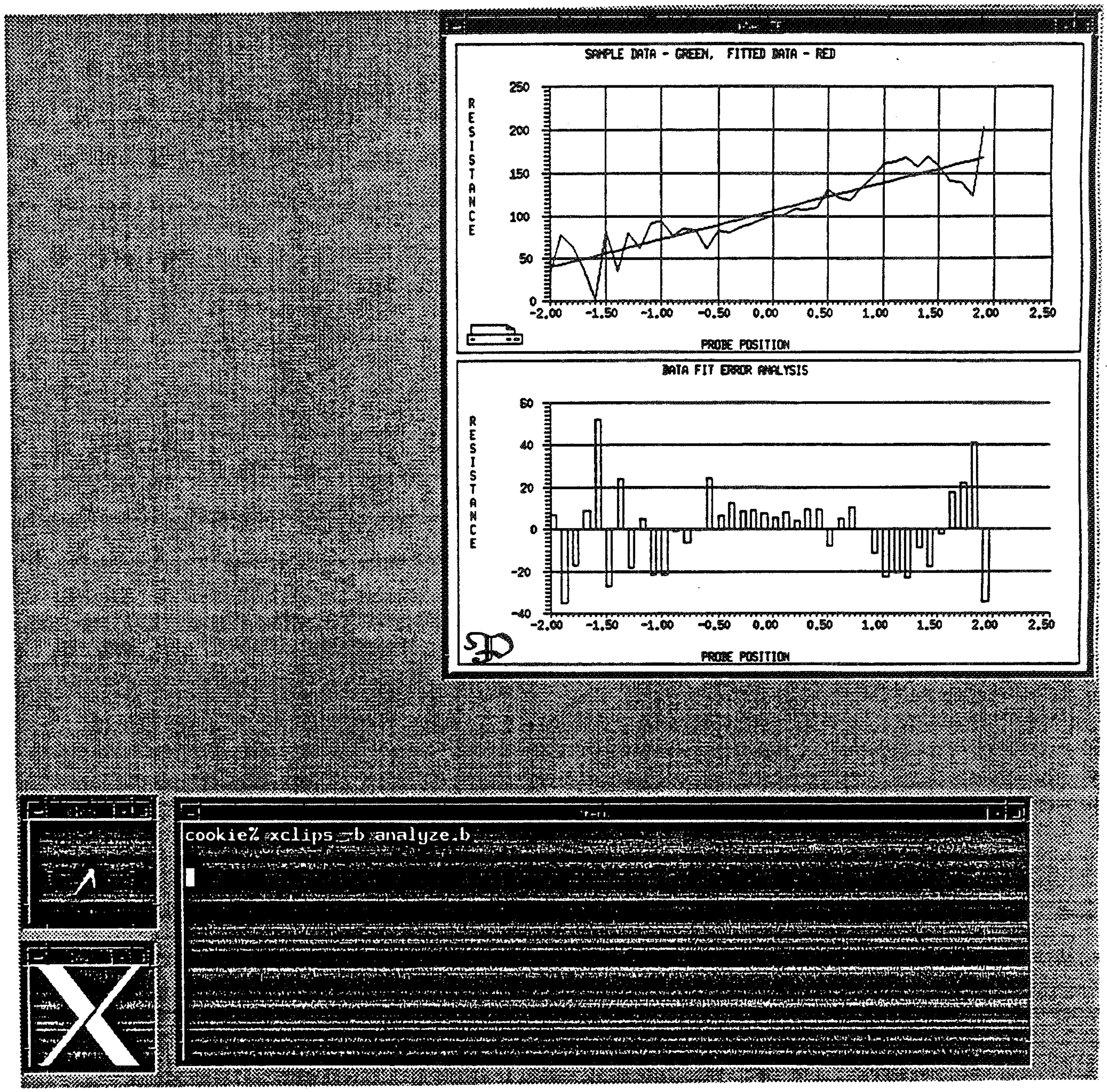
PROJECTION then computes the 3D projection of the contour map and displays the projection.

6.3 Demonstration

The resistance analysis work-station is started by typing "xclips -b analyze.b" at the UNIX shell prompt.

After ANALYZE begins running, the coordinates of the object are sent to the probe. After the necessary data is retrieved from the probe, the graphs are then presented. In the figure on the next page, the 2D chart of the resistance of the material under test along with a prediction of the material's resistance appears in the upper window. In the lower window the error analysis of the predictions appear as a line-plot.

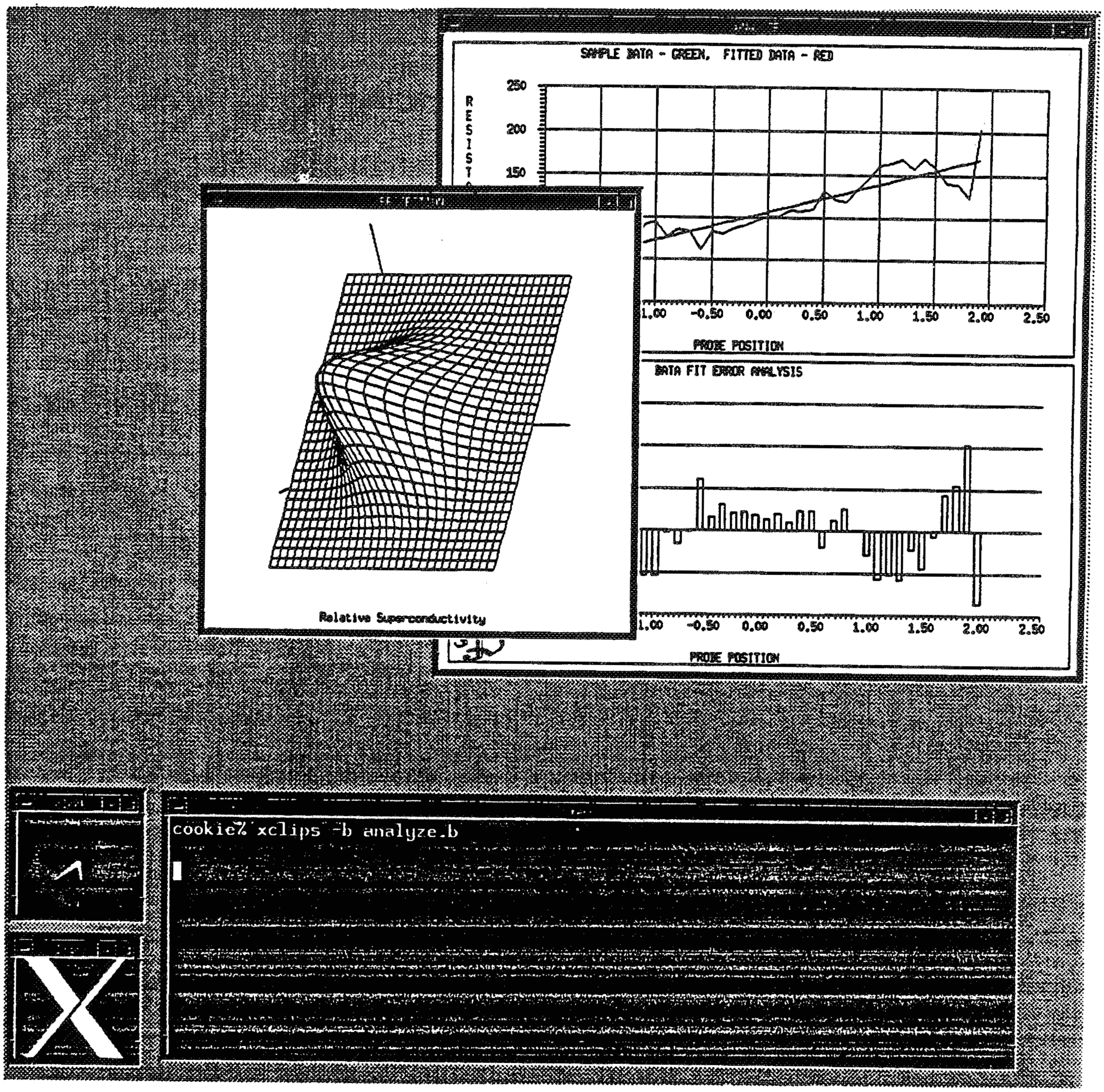
Notice the "printer" icon in the lower left hand side of the upper graph window, and the Gothic "P" icon in the lower left hand side of the lower graph window. Depressing the mouse button while the mouse rests on the "printer" icon causes the rule to fire that prints the window on a dot matrix printer. Depressing the mouse button while the mouse cursor rests on the Gothic-P icon starts the PROJECTION expert system.



```
cookie% xclips -b analyze.b
```

Figure 5, ANALYZE (2D) Display

The following figure shows the result of depressing the Gothic-P icon, that results in the 3D projection of the contour map.



```
cookie% xclips -b analyze.b
```

Figure 6, ANALYZE (2D) and PROJECTION (3D) Display

6.4 XCLIPS Programs

In this section the two expert systems source code is listed in sections 6.4.1 and 6.4.2. In Section 6.4.3 a glossary of the XCLIPS extensions used in this article are presented.

6.4.1 ANALYZE Source Code

```
(defrule initialize "Initialize the ANALYZE expert system"
        (not (system initialized))
        =>
; allocate storage
        (Vector "xdata" 50)
        (Vector "ydata" 50)
        (Vector "indvar" 50)
        (Vector "depvar" 50)
        (Vector "coef" 50)
        (Vector "coefsig" 50)
        (Vector "yest" 50)
        (Vector "resid" 50)
        (Vector "numobs" 1)
        (Matrix "contour" 50)
; connect to probe & get data
        (GetHostByName "probe")
        (bind ?test 0)
        (while (= ?test 0)
                (bind ?test (NetOpen 3000 1)))
        (NetWrite 1 "-2 0 2.5 0")
        (bind ?test 0)
        (while (= ?test 0)
                (bind ?test (NetRead 1)))
        (Assign "numobs" 0 ?test)
        (XTitle "ANALYZE")
        (InitSEGraphics 600600)
        (assert (system initialized)))
        (InitSEGraphics 6)

(defrule display-and-fit ""
        (system initialized)
        =>
        (bind ?numobs (Evaluate "numobs" 0))
; fit data to 1st order polynomial
        (NetRead 1 (Address "depvar"))
        (NetRead 1 (Address "indvar"))
        (PolyCurveFit "indvar" "depvar" ?numobs
                "order" "coef" "yest" "resid" "coefsig")
        (SetCurrentWindow 3)
        (BorderCurrentWindow 2)
        (SelectColor 6)
        (SetAxesType 0)
        (AutoAxes "xdata" "ydata" ?numobs 1)
        (LinePlotData "xdata" "ydata" ?numobs 3 0)
        (SelectColor 3)
        (TitleWindow "SAMPLE DATA - GREEN,  FITTED DATA - RED")
        (TitleXAxis "PROBE POSITION")
        (TitleYAxis "RESISTANCE")
         (bind ?i 0)
         (while (<= ?i ?numobs)
                (Assign "ydata" ?i (Evaluate "yest" ?i))
         (bind ?i (+ ?i 1)))
; draw the curve
        (LinePlotData "xdata"  "ydata"  ?numobs  4 3);
        (DrawGrid 10)
         (assert (display errors))
         (Register (Transform(PutObject "printer") 0 0) "print"))

(defrule display-error analysis
         (display errors)
         =>
        (SetCurrentWindow 3)
        (BorderCurrentWindow 2)
        (SelectColor 6)]
        (SetAxesType 0)
         (bind ?i 0)
         (while (<= ?i numobs)
                (Assign "ydata" ?i (Evaluate "resid" ?i))
              (bind ?i (+ ?i 1)))
        (AutoAxes "xdata" "ydata" ?numobs 1)
        (BargraphData "xdata" "ydata" ?numobs 0.05 1)
        (TitleWindow "DATA FIT ERROR ANALYSIS")
        (TitleXAxis "PROBE POSITION")
        (TitleYAxis "RESISTANCE")
        (DrawGridY 10)
        (assert (watch mouse))
         (Register (Transform(PutObject "Gothic-P") 0 0) "project"))

(defrule watch-mouse "Watch the mouse, and do what it says"
         (watch mouse)
         =>
         (while (= 0)
                (if (= (MouseHit) 1)
                then
                        (GetMouse) (bind ?object (Analyze (Pick)))
                        (if (eq ?object "print")
                        then
                                (ScreenDump "/usr/ben/spool" "epson-lq" 1 1 0))
                        (if (eq ?object "project")
                        then
                                (system "xclips -b project.b &")
                                (GetHostByName "shasta")
                                (while (eq (NetOpen 3000 2) 0))
                                (NetWrite 2 50)
                            (NetWrite 2 15)
                                (NetWrite 1 "contour_map")
                                (bind ?test 0)
```

```
                                        (while (= ?test 0)
                                                  (NetRead 1 (Address "contour")))
                                        (NetWrite 2 (Address "contour"))))))
```

6.4.2 PROJECTION Source Code

```
;*****************************************************
;         Draw 3D From Contour Plot
;*****************************************************

(defrule create-function "Create the contour map"
  (create function)
  =>
  (Vector "elements" 2)
  (while (= (NetRead 1 (Address "elements") 0))
  (bind ?num (Evaluate 1 "elements"))
  (bind ?alloc (Evaluate 2 "elements"))
  (Matrix "contour.x" ?alloc ?alloc)
  (Matrix "contour.y" ?alloc ?alloc)
  (Matrix "contour.z" ?alloc ?alloc)
  (NetRead 1 (Address "contour.x"))
  (NetRead 1 (Address "contour.x"))
  (NetRead 1 (Address "contour.y"))
; draw the contour map
  (Vector "pv.x" 5)
  (Vector "pv.y" 5)
  (Vector "pv.z" 5)
  (bind ?lower (- 1 ?num))
  (bind ?i (* -1 (- ?num 1)))
  (while (<= ?i ?num)
          (bind ?j (* -1 (- ?num 1)))
          (while (<= ?j ?num)
                    (Assign "pv.x" 0 (Evaluate "contour.x" (+ ?i ?lower) (+ ?j ?lower)))
                    (Assign "pv.y" 0 (Evaluate "contour.y" (+ ?i ?lower) (+ ?j ?lower)))
                    (Assign "pv.z" 0 (Evaluate "contour.z" (+ ?i ?lower) (+ ?j ?lower)))

                    (Assign "pv.x" 1 (Evaluate "contour.x" (+ ?i ?num) (+ ?j ?lower)))
                    (Assign "pv.y" 1 (Evaluate "contour.y" (+ ?i ?num) (+ ?j ?lower)))
                    (Assign "pv.z" 1 (Evaluate "contour.z" (+ ?i ?num) (+ ?j ?lower)))

                    (Assign "pv.x" 2 (Evaluate "contour.x" (+ ?i ?num) (+ ?j ?num)))
                    (Assign "pv.y" 2 (Evaluate "contour.y" (+ ?i ?num) (+ ?j ?num)))
                    (Assign "pv.z" 2 (Evaluate "contour.z" (+ ?i ?num) (+ ?j ?num)))

                    (Assign "pv.x" 3 (Evaluate "contour.x" (+ ?i ?lower) (+ ?j ?num)))
                    (Assign "pv.y" 3 (Evaluate "contour.y" (+ ?i ?lower) (+ ?j ?num)))
                    (Assign "pv.z" 3 (Evaluate "contour.z" (+ ?i ?lower) (+ ?j ?num)))

                    (Assign "pv.x" 4 (Evaluate "contour.x" (+ ?i ?lower) (+ ?j ?lower)))
                    (Assign "pv.y" 4 (Evaluate "contour.y" (+ ?i ?lower) (+ ?j ?lower)))
                    (Assign "pv.z" 4 (Evaluate "contour.z" (+ ?i ?lower) (+ ?j ?lower)))
```

```
                (PolyFill3D "pv.x" "pv.y" "pv.z" 9 4 5)
                (bind ?j (+ ?j 1)))
        (bind ?i (+ ?i 1)))
  (assert (watch mouse)))

(defrule watch-mouse "Watch the mouse for click, exit if found"
  (watch mouse)
  =>
  (while (eq 0 (GetMouse))
         (Service))
  (CloseSEGraphics)
  (exit))

(defrule init "Initialize Comm Port, Draw the Axes"
  (not (system initialized))
  =>
  (while (= (NetAccept 3000 1) 0))
  (XTitle "PROJECTION")
  (Tinit3)
  (Init3D 6)
  (SetWorldCoordinates -10.0 -10.0 10.0 10.0)
  (WorldRotate3 20.0)
  (WorldRotate3 -45.0 1)
  (SelectColor 15)
  (Draw3DAxis 10)
  (assert (system initialized))
  (assert (create function)))
```

6.4.3 XCLIPS Extensions Glossary

The following list summarizes the XCLIPS extensions used in this presentation.

Address - Returns the address of the object specified.
Assign - Assign a value to matrix or vector at index specified.
AutoAxes - Draw axes from data.
BarGraphData - draw a bargraph from specified data.
BorderCurrentWindow - Border the current window.
CloseSEGraphics - Close the graphics window.
Draw3DAxis - Draw the 3D axis from specified values.
DrawGrid - Draw a grid in the window.
Evaluate - Return a value from a vector or matrix at specified index.
GetHostByName - Given a host name, initialize the network parameters.
GetMouse - Get the mouse position.
Init3D - Initialize 3D graphics routines.
InitSEGraphics - Initialize the 2D graphics routines.
Line3Abs - Draw a line from current 3D point to specified 3D point.
LinePlotData - From specified data, draw the plot.
Matrix - Create a 2D matrix.
Move3Abs - Move to specified point in 3D.

Move3Abs - Move to specified point in 3D.
NetAccept - Accept connections from the network.
NetOpen - Place a call on the network.
NetRead - Read data from a process across the network.
NetWrite - Write data to a process across the network.
Persp - Set the 3D perspective.
Pick - Determine if an object is registered at this mouse position.
PolyCurveFit - From specified data, create a curve.
PolyFill3D - Draw a 3D image from a contour map.
PutObject - Put specified object in window.
Register - Register an object at specified position.
ScreenDump - Print the window contents.
SelectColor - Select current color by an index.
SetAxesType - Set the axes type to use on subsequent calls.
SetCurrentWindow - Set operations to point into specified window.
SetWorldCoordinates - Set the world coordinates as they relate to the screen.
TitleWindow - Place a title on the current window.
TitleXAxis - Place a title on the X axis in current window.
TitleYAxis - Place a title on the Y axis in current window.
Transform - Transform world coordinates to real.
Vector - Allocate storage for a one dimensional array.
WorldRotate3 - Rotate a point in a 3D space.
WorldScale - Scale the window by specified world coordinates.
XTitle - Title the window for use by the X Window System window manager.

7. CONCLUSIONS

XCLIPS is readily applicable to solutions that require graphical expressions. The XCLIPS rule is a very flexible control mechanism for handling icons, mouse and keyboard devices, and drawing simple to very complex pictures.

XCLIPS based graphics solutions to very complex problems tend to be straight-forward and compact. With the addition of communications support, transparent distributed XCLIPS applications are as easy to build as monolithic systems.

The one area where XCLIPS is difficult to apply to data intensive problems is in the area of performing complex computations. If the language employed an operator precedence grammar, this difficulty would be eased, but the language would become less uniform.

8. FURTHER READING

A good understanding of the general graphics principles used in extended CLIPS is contained in the following reference: Newman, M.N., Sproull, R.F., *Principles of Interactive Computer Graphics*, McGraw Hill Book Company, New York.

For information on graphics programming in the PC environment under MS-DOS, two books are especially helpful: Wilton, R., *Programmer's Guide to PC & PS/2 Video Systems*, Microsoft Press, Redmond Washington and in Microsoft Corporation (1987), *Microsoft C 5.1 Optimizing Compiler Run Time Library Reference*, Microsoft Press, Redmond Washington.

The following book was used a reference for X Window System graphics programming: Nye, A. (1988), *Xlib Programming Manual Vol. 1*, O'Reilly & Associates, Sebastapol CA.

The following book describes the algorithms used for solving systems of simultaneous equations and polynomial curve fitting: Chapra, S.C., Canale, R.P. (1985), *Numerical Methods for Engineers*, McGraw-Hill Book Company, New York.

The following document describes advanced programming topics in CLIPS: Giarratano J.C., *CLIPS Reference Manual*, Johnson Space Center, Houston TX.

A Graphical Interface to CLIPS Using SunView

Terry Feagin
University of Houston - Clear Lake

Abstract

The importance of the incorporation of various graphics-oriented features into CLIPS is discussed. These features, which have been implemented in a version of CLIPS developed for a popular workstation, are described and their usefulness in the development of expert systems is examined.

Introduction

When developing expert systems that are intended to interact heavily with the user (as opposed to those systems that operate in a primarily independent manner), it is essential to provide an interface that enhances and accelerates the process, that allows meaningful dialog with the least effort, that provides clear and unambiguous two-way communication, that expedites the handling of sensitive or emergency situations, and that provide intuitive mechanisms for giving commands to and for receiving responses from the expert system. Computer graphics has long been recognized as a valuable aid in facilitating the flow of information between users and their computer-based applications. Instead of allowing only a few one-dimensional streams of characters (i.e., input and output files and a command line interface), modern computer graphics admits the possibility of interacting with the user via a number of two-dimensional color images that can move or be influenced directly by the user using a mouse, light pen, keyboard, joystick, or other graphic input device. The images are often organized into windows and user-interaction is often provided via menus that are mouse-selectable.

There are, of course, several enhanced versions of CLIPS that provide support for graphic-based interactions. However, these are primarily provided to enhance the giving of commands or setting conditions at the top-level (which level of support, it might be added, is also provided by the system described below).

For example, the Macintosh interface allows users to clear, load, reset, run, etc. by making conventional menu selections. However, there is no direct support for opening windows or generating menus from within an executing expert system.

Because of the extensible nature of CLIPS, it is not difficult to develop such support by adding user-defined functions. This paper, as well as several other papers presented at this conference, offers the expert system developer the ability to support directly such graphic-oriented interfaces to the user.

In an expert system, it is often desirable to convey to the user a set of conditions that may be true or false or indeterminate (inactive, disabled, etc.). Additionally, it may be important to show a precise measurement or reading. In the traditional command-line versions of CLIPS, these conditions would normally be exhibited via printed messages. In a graphic-oriented interface, the natural vehicle for conveying such information would be to provide an image or icon that might be immediately recognized by the user and to alter the image in a way that might graphically depict the level of the condition. For example, in an expert system developed to assist the operators in a nuclear power plant, excessive temperature conditions might be indicated by flashing red in an image depicting a thermometer. In an expert system developed to control a chemical reaction, the pH of a solution might be indicated by showing a dial. If the pH exceeds certain limits (either high or low), then the dial could be repainted, for example, in red for low pH and in blue for high pH values. If a serious or emergency condition holds, a graphic-oriented interface might be set up to flash the whole screen or window in color, to set off audible alarms (such as a beep or buzzer sound), to present the operator with an alert window with various possible actions or options designated, and to allow the operator to view and evaluate the consequences of his/her actions on the system via explanatory text revealed in windows and additional diagrams of equipment, meters, or fault nets.

Specific Advantages of a Graphical Interface

Most of the advantages of providing a graphical user interface to an expert system are obvious. Human operators are usually more receptive to a new environment if it is intuitive, pleasing to the eye, and easy-to-learn. A well-designed graphical interface assists the operator in *visualizing* the problem at hand, the relationships between entities in the system or variables in the problem, the ways in which she/he can or cannot affect the behavior of the system or the solution of the problem, the current state of the system or the solution process, the distinction between essential and non-essential characteristics or conditions, and any hierarchical organizational relationships.

Any change in the system can be identified almost immediately and the more significant changes can be allowed to trigger the more visually stimulating graphical effects (such as flashing lights or images and alert windows). This kind of separation of more significant from less significant events is difficult to accomplish as effectively with a simple command line interface.

Animation of windows or objects within windows can be used to represent higher level concepts such as a sequence of actions or events that are particularly difficult to represent in a simple command line interface. This is especially helpful if the speed of the animation can be varied.

In a graphical user interface, it is easier to control and restrict the user's input when it is important. Typographical errors can be eliminated. Other types of user input errors such as clicking on the wrong object are possible, but can be readily monitored and the user can be requested to confirm any unusual input.

Even a simple presentation of images within windows can be effective. The images can be used to present aspects of the problem or system that are otherwise difficult or impossible to present. In many of the science and engineering disciplines, there are times when some kind of two-dimensional image is the best way to represent a problem or method of solution. For example, in an expert system that might be used to help solve heat transfer problems, it may be desired to show the temperature distribution over the surface of some physical object like a flat plate of copper. The actual temperature distribution could be displayed as a color-coded image within one of the windows and used to show progress toward a solution.

In an object-oriented approach within CLIPS, one may wish to identify specific graphic objects or items that represent the objects about which the system is reasoning. As the attributes of the objects change, the graphic representations (position, size, shape, color, motion, etc.) could be made to change as well, thereby giving the user a view of the reasoning process that might be difficult to provide with the more usual command line interface. As the new object-oriented version of CLIPS emerges, this advantage may become even more significant.

Another advantage of a graphical approach concerns explanation facilities. If one creates an expert system in which the user can simply depress a mouse button over an object to signify that the user requires an explanation of the reasoning process or simply requires help regarding the meaning of the object, then the expert system can be more readily understood and may even be used for training new users. Also, if a user enters an unusual, expensive, hazardous, or dangerous request of the system, the system can ask for confirmation with an alert window complete with a cautionary warning. Security can be enforced by requiring passwords at critical points before actions are taken.

A graphics environment is less tiring to the user. Graphical output is generally able to convey more information with less eyestrain than simple text. In a command line interface, a user may miss an important detail that can become lost in line after line of alphanumeric characters. For input, many users often find that using a mouse is easier than typing.

Some Graphics Primitives Useful in Creating Expert Systems

The kind of graphics primitives that one might select for creating an expert system will undoubtably vary from one application to another. A general expert system shell that proposes to support the user in all of the ways mentioned above must therefore be able to support a wide variety of graphic objects and functions. In this section, several kinds of graphic objects and functions are described and some examples of how they might be used in an expert system are given.

WINDOWS - A CLIPS programmer should be able to call functions that cause windows to be created, opened, closed, hidden, exposed, and destroyed. The size and position of any window should be adjustable from within CLIPS or directly by the user. Other useful attributes might include scrollbars, labels, and colors. The windows should have the same appearance as non-CLIPS windows. It should be possible to retrieve most window attributes directly.

ITEMS - There should be the ability to support a number of graphic objects or items within each window. It should be possible to create, hide, show, and destroy items. It should be possible to label the items, and move them about under user control or program control. Several especially useful types of items might be identified. For example, a button item would be useful for selecting conditions or indicating operator actions. Most graphical interfaces provide for this type of object. Such items should be displayable as general graphical images in color or as simple labeled buttons. Another useful type of item would be text items. These items could be used to prompt the user for input with text strings and enable the user to enter filenames, passwords, or other text input. Such objects could also be used for short messages to the user. Other types of items might also be defined. Animation of items would be also useful, particularly in simulations (another area where the use of CLIPS is growing rapidly).

Items should be selectable and the result of a user selecting such an object should be the assertion of a fact describing the event. It would also be useful for items to be highlighted when selected, thereby providing positive feedback to the user. It would also be useful to be able to get most item attributes directly.

MENUS - Menus should be supported for both windows and items within windows. Menus should be displayed according to the conventions supported by the windowing system in general. Whenever a menu selection is made, a fact should be asserted describing the selection. It should be possible to remove a menu and create a new menu for an item or a window.

DRAWING PRIMITIVES - Certain simple drawing primitives should be provided as a minimum, including the ability to draw lines, draw polygons, fill regions , load images from raster files into windows, and save window images (all in color, of course). It would also be important to be able to get the pixel value at a particular location within a window.

OTHER FEATURES - Other helpful features of a graphical interface to CLIPS would include the ability to change the color definitions in the colormap segment for a window, to cycle a window's colors, to repaint a window, to cause an item to be highlighted, to change the menu for an item, to remove all items in a window, to remove all windows, and to remove all the items in all the windows.

There are also a number of functions that might be provided for debugging purposes such as the ability to print a list of all the windows or a list of all the items in a window for examination.

Implementation : Some Questions

Most of the functionality for graphical objects described above is supported for C programs executing on Sun™ workstations under the windowing system SunView™. Making this functionality available to CLIPS programs is somewhat complicated by a number of issues:

How many of the hundreds of options available under SunView are really important for supporting the development of expert systems ? What features not currently supported by SunView should be added ?

Should the central control loop remain within CLIPS or should control be given to the main loop in SunView and returned to CLIPS only for the handling of predesignated events ? The latter callback mechanism is the one normally used when developing SunView applications. Also, how should multiple simultaneous input streams be treated ?

Should the SunView distinction between a window for images or drawings (known as a "canvas" in SunView) and a window for button and text items (known as a "panel") be maintained or should a "new" type of window be adopted that would incorporate the essential features of both panels and canvasses ? The second approach would be less confusing and present additional power to the CLIPS programmer.

Implementation : Some Answers

Out of the hundreds of options available to general SunView applications, it was decided that only those most useful to the expert system developer would be supported. In the current version of the system, the most significant functions supported are as follows:

create_window - causes a window to be created with attributes as specified,
remove_window - causes a window to be destroyed, releasing resources used,
remove_all_windows - causes all windows to be removed,
hide_window - causes a specified window to be hidden from view,
show_window - causes a hidden window to be exposed,
open_window - causes a closed, iconified window to be opened,
close_window - causes an open window to be iconified or closed,
set_window - allows resetting of a window's attributes,
get_window - allows retrieval of a window's attributes,
set_window_color - allows redefinition of the particular colors used in a window,
set_window_fg - sets the window's foreground color,
set_window_bg - sets the window's background color,
draw_window - allows drawings to be created within a window,
load_window_image - allows a user-specified image to be loaded in a window,
save_window_image - causes the window's image to be saved in a file,

create_item - causes an item of specified type to be created in a window,
remove_item - causes an item to be removed permanently from a window,
remove_all_items - causes all the items in a window to be removed,
remove_all_items_in_all_windows - causes all items to be removed,
hide_item - causes an item to be hidden from view,
show_item - causes an item to be exposed,
set_item - allows for resetting of attributes of an item, including its image,
get_item - allows for retrieving attributes of an item,
highlight_item - allows the item to be highlighted for a flashing effect,
animate_item - permits the item (i.e., its image) to appear to move within a window at a specified rate of speed,

create_item_menu - causes a user-specified, item-dependent menu to be created for the item,
remove_item_menu - causes menu to be removed from the item,
set_item_menu - allows menu attributes to be established,

get_alert_window - causes an alert window to be displayed and blocks user from entering input (except to indicate a response to the alert),
cycle_window_colors - allows the colors in a window to be cycled,
repaint_window - allows the window to be repainted,
snooze - causes CLIPS to sleep for a user-specified number of milliseconds,
list_windows - causes a list of the presently defined windows to be produced,
list_items - causes a list of presently defined items to be produced,

In almost all of the above functions, the number of arguments has been limited in order to make the syntax of each function easier to remember. The arguments are generally ordered in such a way that the most significant arguments appear first, thus allowing the CLIPS programmer to omit some of the less significant arguments (thereby implicitly specifying default values for such arguments).

In addition to the explicit function calls listed above, a user can interact with the system in a number of ways, primarily by making mouse movements and mouse button selection over windows, items, and menus. Text entry is also supported.

Regarding the issue of control, it was determined that the central control loop of CLIPS would be maintained and that SunView's Notifier (the dispatcher which allows client programs to register event handlers and receive notifications later when the respective events occur) would be called explicitly after each rule firing and implicitly during any blocking or non-blocking read. This allows the user to obtain good response to graphics input events while CLIPS is firing rules and also when the user is entering commands or function calls directly to the CLIPS prompt.

It was also determined that the distinction between a canvas and a panel in SunView would be superfluous. The features of both have therefore been combined by adding the essential features of a canvas (namely, most two-dimensional graphics primitives such as drawing lines, constructing images, setting colors, and getting pixel values) to the panel type of window. Therefore, to the CLIPS programmer, there appears to be a core type of window that allows for buttons, simple text interactions, and color graphics output.

Another feature that was not directly supported under SunView is the dynamic movement of items under user control. By dragging the item with the middle mouse button depressed, the user can reposition the item at will. Afterwards, the new position of the item is reported to CLIPS as a new fact assertion. These last two features make the package much simpler and more useful for developing expert systems.

The animation of an item is also not directly supported under SunView. However, given the growing interest in using CLIPS for the development of simulations (whether or not such simulations are a part of an expert system), the animation feature as given above has also been provided.

Conclusions

The project has now been successfully completed and over thirty-five new functions (mostly graphics-oriented) have been added to CLIPS. Work is now underway to enhance these capabilities even further and study their usefulness within several existing expert systems.

omit

A13 Session:
Aerospace Applications

On A Production System Using Default Reasoning For Pattern Classification

Matthew R. Barry
Carlyle M. Lowe

Rockwell Space Operations Company
NASA/Johnson Space Center DF63
Houston. TX 77058
mbarry@nasamail.nasa.gov

1 May 1990

1 Introduction

This paper addresses an unconventional application of a production system to a problem involving belief specialization. The production system reduces a large quantity of low-level descriptions into just a few higher-level descriptions that encompass the problem space in a more tractable fashion. This *classification* process utilizes a set of descriptions generated by combining the component hierarchy of a physical system with the semantics of the terminology employed in its operation. The paper describes an application of this process in a program. constructed in C and CLIPS. that classifies signatures of electromechanical system configurations. The program compares two independent classifications. describing the actual and expected system configurations, in order to generate a set of contradictions between the two.

1.1 Background

The problem application considered herein involves the operational evaluation of NASA's Space Shuttle hardware configurations by flight controllers in the Mission Control Center (MCC). Specifically, the technique has been applied to one of the tasks involved in monitoring the two Shuttle propulsion systems: the *Orbital Maneuvering System* (OMS) and the *Reaction Control System* (RCS).

Shuttle astronauts operate the propulsion systems by manipulating a collection of switches and valves that control fluid flows throughout the plumbing network. Many of the switches control two propellant line valves simultaneously: an oxidizer valve and the corresponding fuel valve. Position indicators within the valves and switches provide insight into their mechanical position. Flight controllers in the MCC help the astronauts to manage these systems by monitoring the on-board configuration. Valve and switch positions appear to the flight controllers as binary values noting presence of (or lack of) an open indication, closed indication, or both. A set of 16-bit *configuration words* relay all of the available measurements through the orbiter computers to the flight controllers.

The MCC computers help the flight controllers to monitor the on- board valve and switch configuration by executing a program that compares *actual* and *expected* configurations. Since only some of the bits in a given configuration word apply to the propulsion systems, the comparison procedure includes a set of masking words. When the bit patterns that are not subject to the mask do not match, the program indicates a problem by displaying a certain status character next to that word. Since the contents of those words are displayed in hexadecimal, flight controllers are made aware of a discrepancy condition through this status character, but are not informed of the actual discrepancy. Furthermore. several discrepancies may occur in the same word.

1.2 Problem

The process of manually decoding this information is time consuming and prone to error. A decoding program is available that will prompt the user for hexadecimal input values, apply the mask values, then display the descriptions of bits that do not match the expected pattern. It is up to the user to remember the patterns from each individual decoding, and to construct a complete signature from the many hexadecimal words. This process actually must be performed twice, once for the *actual* signature and once for the *expected* signature. Comparison of the two signatures relates the changes that have occurred in the configuration since the last state update.

2 Description

A classifier can perform this decoding task easily through deductive and default reasoning. The decoding program can be extended to isolate each bit in the configuration words and to generate a *proposition*[1] for a database stating the observed position of each valve or switch. The classifier can then attempt to generate a state description for these indications. The state descriptions offer an explanation in high-level, intuitive, terminology. For example, instead of being offered the propositions

p_1 = *The manifold 1 ox open indication is present*
p_2 = *The manifold 1 fu open indication is present*
p_3 = *The manifold 1 ox close indication is not present*
p_4 = *The manifold 1 fu close indication is not present*

the flight controller should be informed

p_5 = *The manifold 1 valves are open*

[1] The term *proposition* is used here instead of the expected *fact* in order to provide consistent terminology with the deductive reasoning systems discussed throughout the paper.

due to the application of a typical rule r_1:

$r_1 =$ if $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ then
assert $p_5 =$*The manifold 1 valves are open*,
and retract p_1, p_2, p_3, and p_4.

Better still, if the following propositions are available,

$p_5 =$*The manifold 1 valves are open*
$p_6 =$*The manifold 2 valves are open*
$p_7 =$*The manifold 3 valves are open*
$p_8 =$*The manifold 4 valves are open*
$p_9 =$*The manifold 5 valves are open*

then the best description is

$p_{10} =$*All manifolds are open*

from the rule r_2:

$r_2 =$ if $p_5 \wedge p_6 \wedge p_7 \wedge p_8 \wedge p_9$ then
assert $p_{10} =$*All manifolds are open*,
and retract p_5, p_6, p_7, p_8 and p_9.

Carrying on to "meta-level" statements regarding a "configuration of configurations," one might make the specialization of the propositions

$p_{10} =$*All manifolds are open*
$p_{11} =$*Both regulators are open*
$p_{12} =$*Both crossfeed valves are closed*
$p_{13} =$*All tank isolation valves are open*
$p_{14} =$*All thruster heaters are off*

resolve to the implicit description

p_{15} =*Prelaunch configuration*

Such descriptions explain implicitly the underlying meaning. In this sense, *the output of the production system is itself the explanation of the reasoning process.*

2.1 Specialization

The sort of classifier described above has been implemented through the use of a production system shell. Statements providing a *specialization of beliefs* are represented conveniently with conventional production rules. The left-hand side of the rule consists of one or more predicate propositions which, when considered together, imply a more specialized statement having equivalent meaning. The right-hand side of the rule asserts the new statement and retracts all of the propositions that were held true in order to activate the rule. This assertion/retraction process decreases the number of propositions in the database, while maintaining equivalent knowledge of the reasoning world. Since the system can retract its own assumptions later in the deduction process, the process is a manifestation of *nonmonotonic reasoning.*

The classifier employs a combination of procedural and declarative programming techniques. NASA's C Language Integrated Production System (CLIPS) provides the rule processing capabilities. The host program, written in C, acquires the necessary data and applies a valuation algorithm to generate database propositions. This algorithm assigns to each positive component position indication a description of the component, a description of the position indication (e.g. *Open*, *Close*, *On*, or *Off*), and a qualifier as to whether that position belongs to the *actual* or *expected* configuration. When all necessary propositions have been generated, the production system evaluates them and builds the state description. The contents of the database after all possible specializations have been applied (i.e. when no more rules fire) represent the state description. The host program expands these remaining propositions into English sentences for display to the users.

2.2 Default Reasoning

Since the independence of valve or switch state indications is not guaranteed by the physical system, the design-intended independence is not considered important by this production system. That is to say, though the valves are intended to reside in either the opened or closed states, the indications may not provide conclusive evidence and perhaps no default assumptions are available. For these situations none of the statements that consider the guilty valve will be applied, thus leaving the lowest level propositions in the database and resulting in a very specific state description. Detection of these situations sometimes leads to further detailed observations of hardware performance in order to obtain alternative cues that support one or more of the indications. Moreover, facts are held based on *observed* states rather than *assumed* states[2].

One important consideration in the solution is that *lack of evidence regarding a position indication is useful information.* That is, missing information may imply a certain position indication. For the OMS and RCS, this happens with the switch positions: lack of an OPEN or CLOSED indication means that the switch is assumed to be in the GPC (General Purpose Computer) position for automatic valve control. Missing information is also important in OMS and RCS valve positions: many valves lack a CLOSED indication, so that if the OPEN indication is not present, then the flight controllers must assume that the valve is closed. For these reasons, the classification process must allow for *default* values for certain propositions.

Recent research efforts attempting to solve default logic problems have centered around extending classical mathematical logics to account for implicit information in the database. This typically is done by making assumptions about missing information by providing default values. In some cases, providing default values is in itself another problem that must be handled in the reasoning system. Etherington [1988] provides a summary of current techniques for handling missing information. Besnard [1989] provides a formal introduction to default logic.

[2] There remains the underlying assumption, however, that the observed state represents the actual state.

In an attempt to restrict the reasoning assumptions to information that is available, the *Closed-World Assumption* (CWA) has been developed [Reiter 1978]. The CWA is the assumption of complete knowledge about which positive facts are true in the world. Under the CWA, it is not necessary to explicity represent negative information. Negative facts may be inferred from the absense of the same positive fact. The CWA corresponds to the knowledge base:

$$\text{if } KB \nvdash P \text{ then infer } \neg P,$$

which states that if the proposition P cannot be derived from the knowledge base KB, then it is reasonable to assume that P is false. Furthermore, one can imagine collecting the set of all false propositions derivable from KB into another knowledge base. Reiter calls this set the *negative extension* of KB, or $\overline{EKB}$.

Traditional logics do not possess means for considering the absence of knowledge. Research has considered two sorts of information types whose implementation can extend the capabilities of traditional logics to cover this shortcoming. In the *positive information* category, one assumes that relevant information is known, therefore anything that is not known must be false. In the *default information* category, one has default values available to fill gaps in the absence of specific evidence. The *default information* category describes the reasoning process embodied by the classifier.

A *default logic* may be constructed from a standard *first-order logic* by permitting addition of new inference rules [Reiter 1980]. These new rules allow *known* and *unknown* premises, making possible conclusions based on missing information. A *default theory*, Δ, is an ordered- pair (D, W) consisting of a set of *defaults*, D, and a set of *first-order formulae*, W. The fundamental statements in Δ are *defaults*, defined by the expression:

$$\frac{\alpha(\overline{x}):\beta_1(\overline{x})\ldots\beta_m(\overline{x})}{\gamma(\overline{x})}.$$

where $\alpha(\overline{x})$, $\beta_i(\overline{x})$, and $\gamma(\overline{x})$ are formulae whose free variables are contained in $\overline{x} = x_1, \ldots, x_n$. This expression states that if certain *prerequisites* α are believed, and it is consistent to belive that certain *justifications* β are true, then it is reasonable to sanction the *consequent* γ. Stated another way, if the prerequisites are known and their justifications are not disbelived, then their consequents can be assumed. Conventionally, if $\beta(\overline{x}) = \gamma(\overline{x})$, then the default is *normal*, and if $\beta(\overline{x}) = \gamma(\overline{x}) \wedge \omega(\overline{x})$, for some $\omega(\overline{x})$, then the default is *semi-normal.* The sets of conclusions sanctioned by Δ are the knowledge base *extensions*.

As a simple demonstration, consider the typical AI example

$$W = \{BLOCK(A) \vee BLOCK(B)\}.$$

If we assume the closed-world defaults

$$D = \{\frac{:\neg BLOCK(A)}{\neg BLOCK(A)}, \frac{:\neg BLOCK(B)}{\neg BLOCK(B)}\},$$

then the theory Δ has the two extensions E_1 and E_2.

$$E_1 = Th(\{\neg BLOCK(A), BLOCK(B)\})$$
$$E_2 = Th(\{BLOCK(A), \neg BLOCK(B)\})$$

This example shows that the system has concluded that either A is a block or B is a block, but not both. The system adds these conclusions to the database as extensions. In elaborate situations it is likely that interactions between defaults may raise conflicts. Semi-normal defaults provide a means for resolving ambiguities between interacting defaults, so long as the interactions are known *a priori* [Reiter and Criscuolo 1981].

Conventional deductive inference involves the *monotonicity* property: as the set of beliefs grows, so does the set of conclusions that can be draw from those beliefs [Ginsberg 1987]. However, if one now adds new information to the set of beliefs, then some of the original conclusions may now be invalidated. The ability to withdraw a previous assumption and reconstruct a new set of conclusions is known as *nonmonotonic reasoning.*

3 Implementation

The pattern classifier presented herein performs default reasoning in a manner analogous to the approach formulated by Reiter. The production system inference engine controls application of the specializations and manages the database. The host program and *deffacts* blocks initialize the database. The host program then calls CLIPS to execute the inference process. After completing the classification, the host program unloads the interesting propositions remaining in the database and displays them to the user.

3.1 Input Processing

Input data can be provided by the user or can be acquired from the telemetry stream via local area network (LAN). If the user provides the data, he is prompted by the host program to enter the configuration word identification tag (or "measurement stimulus identification") and the actual and expected bit patterns (in hexadecimal). When all desired input has been provided, the evalution process begins. The host program unloads the resulting database and parses the remaining propositions into English sentences for display. When the user is satisfied that he understands any configuration descrepancies, he can issue a request to reset the *expected* configuration words to the *actual* configuration words, thus updating the comparison pattern to the known state.

Since there are 90 configuration words recognized by the host program, it is unlikely that the user will provide all possible input. This is of no significance to the classifier, as it will work on whatever propositions are provided, no matter how limited. If very little information can be provided from the configuration words provided, then one should expect low–level results. The more information that is provided, the better the classification. To assist in the data acquisition process, the host program was modified to accept data from a LAN. The network interface requests 24 valve configuration words and 66 switch configuration words from the telemetry stream. These 90 words contain all of the discrete information that pertains directly to OMS and RCS operations[3]. With all of this data, the classifier is able to make the most specific statements possible.

3.2 Providing Defaults

In order to perform reasoning about the default values, a group of special rules were developed. These rules process the *deffacts* statements that are labelled with the *default* token by attempting to match on any overriding fact from the *actual* or *expect* environments. Stated differently, if the default fact is the only one available for a particular valve or switch, then the value provided as the default indication for that component becomes the value of the missing fact. If any evidence other than the default value is available, that evidence is used in the classification process. The rules performing these operations are described in more detail in the following section.

3.3 Production System

The CLIPS inference engine performs all of the deductive reasoning. It is allowed to run through exhaustion, eliminating as many propositions as possible by applying the specialization rules. These rules heavily exploit the pattern matching capabilities provided by CLIPS, due to the symmetric nature of the physical domain. Moreover, the rules work for either of the two configuration states, matching (with restrictions) on the pattern predicate.

The knowledge base construction is rather simple. It consists of *default processing procedures*, *classification schemas*, *configuration comparators*, and *physical system information*. The expertise is explicit in the classification reductions; knowing how to represent a configuration through its operational semantics, and knowing how to manage the associated default assumptions.

The *default processing procedures* are probably the most interesting. These rules fire first so as to build all of the lowest-level indications before starting

[3] Discrete information from other subsystems, such as data processing, indirectly affect OMS and RCS operations, but have not yet been included.

specializations. In order to reason about defaults one must be able to decide when information is missing. This application uses the CLIPS **not** operation for this purpose. This operation returns TRUE if a match is *not* available for the pattern, thus allowing us to determine that default-overriding evidence is not present in the database. Operation of these rules may be described as follows: Given a set of default values in a **deffacts** block,

```
(deffacts default-values
     (default lrcs he-press-a sp-gp)
     (default lrcs he-press-b sp-gp)
     (default lrcs tank-isol-12 sp-gp)
     ...
)
```

we are able to provide a default value for any particular component in the physical system, including those that may be "exceptions."[4] The first entry in the abbreviated table above states that the default position for the Left RCS Helium Pressurization A switch is the GPC position (**sp-gp**). Now, consider the default assertion rule for the expected switch indications,

```
(defrule expect-switch-defaults
     (declare (salience 100))
     (default ?domain ?component ?d&sp-op|sp-cl|sp-gp)
     (not (expect ?domain ?component sp-op))
     (not (expect ?domain ?component sp-cl))
     (not (expect ?domain ?component sp-dm))
     (not (expect ?domain ?component sp-gp))
=>
     (assert (expect ?domain ?componet ?d))
)
```

This rule binds a default indication from the default table (described below), specifying that it handles only switches by restricting the default value

[4] Explicit statement of the default facts is required because the not operator is unable to bind variables for use outside of the not scope.

to one of the three reasonable switch values (the value of *dilemma* (**sp-dm**), though a possible observed state, is not a reasonable default value). It then proceeds to search for an overriding indication by looking for all possible switch values in the **expect** indications. If a match is found, then an **expect** indication is available and the rule fails. If no match is found, then the default value is assumed appropriate, the rule fires, and the default value is asserted as the **expect** value on the right-hand side. Similar rules exist for reasoning about the **actual** indications and for valves.

Most of the production rules represent the *pattern classification schemas*. As described, these rules assemble collections of facts into a more specialized fact implying the same information. The right-hand side of the rule retracts the premises and asserts the conclusion. Each of these rules works for either of the two comparison states. Recalling the manifold example provided above, the classification schema for this specialization appears as the rule:

```
(defrule specialize-group-manifolds
  ?m1 <- (?mode&actual|expect ?domain manifold-1 ?s ?v)
  ?m2 <- (?mode               ?domain manifold-2 ?s ?v)
  ?m3 <- (?mode               ?domain manifold-3 ?s ?v)
  ?m4 <- (?mode               ?domain manifold-4 ?s ?v)
  ?m5 <- (?mode               ?domain manifold-5 ?s ?v)
=>
  (retract ?m1 ?m2 ?m3 ?m4 ?m5)

  (assert (?mode ?domain manifolds ?s ?v))
)
```

This rule collects all five of the named manifolds for an arbitrary domain (Left RCS, Right RCS or Forward RCS) and either environment (**actual** or **expect**). Provided that the switch and valve positions (**?s** and **?v**) for each manifold are the same, the special conclusion **?domain manifolds** is asserted. Prior to the special assertion, however, the antecedants are retracted from the database[5]. If not all of the five manifolds indicate the

[5] The retraction is performed before the assertion in order to reduce the complexity of driving patterns through the network.

same valve and switch positions, this rule will fail for that domain. This will leave the individual (lower-level) facts in the database for the display utility, thus maintaining the highest level of specialization possible without introducing ambiguity.

Two *configuration comparison* procedures perform the comparison between the *actual* and *expected* configurations. These rules fire last, allowing all possible specialization to take place before evaluating the differences between the two configurations. Simply put, if the **actual** and **expect** equivalents for any one component or configuration are not the same, then the configuration is declared a **mismatch**. This simple rule performs those actions:

```
(defrule config-mismatch
     (declare (salience -100))
     ?ce <- (expect ?domain ?set $?des)
     ?ca <- (actual ?domain ?set $?ind)
     (test (neq $?des $?ind))
=>
     (retract ?ce ?ca)

     (assert (mismatch ?domain ?set $?des $?ind))
)
```

The **des** and **ind** variables are multifield variables because they can bind to either one or two fields, depending on the degree of specialization achieved for any one component. Through the test operation, we see that if the multified variables are not the same, then the mismatch is declared. A similar rule, **config-valid**. is used to assert **confirmed** configurations.

There are only a few facts that remain fixed in the application. These are the *physical system information* facts. All of these facts were installed in order to reduce the number of rules required to manage only slightly different configurations. These facts relate the interdependence among various components in the physical system, and enforce some degree of control over variable binding when a model requires information about a component and another "corresponding" or "associated" component. For example, the **deffacts** block:

```
(deffacts relationships
        (corresponding loms roms)
        (corresponding roms loms)
        (corresponding lrcs rrcs)
        (corresponding rrcs lrcs)
)
```

is used to associate the name of the system related to (but not identical to) the system under consideration. Using the first fact, **(corresponding loms roms)**, the token **roms** becomes available when reasoning about the **loms**. This is handy when trying to determine special hardware configurations where one system is connected to another.

3.4 Post-Processing

The existing hexadecimal decoding program was modified slightly so as to accomodate CLIPS fact processing. For each of the bit descriptions, a fact-like sentence was attached to the corresponding data structure. When this bit is given a value and the classifier is subsequently invoked. the associated sentence is *string-asserted* into the fact list. The program was modified to search the fact list for any **mismatch**, **confirmed**, **actual** and **expect** facts upon return from the classifier. Since the first two fields completely define the structure of the English sentence used to describe the fact. the parse tree is rather simple. The fact fields are assembled into a string using **sprintf()**, then sent to the display processor.

The host program "knows" a few things about CLIPS data structures. Since the output is required to be processed on a graphics terminal running under a window manager, display management has been delegated to the host program instead of the production system. Therefore, in order to parse the facts that remain in the database, a simple procedure for processing the facts list was developed. This procedure steps through the linked fact list, searching for facts whose first token identifies an item of interest to the user, i.e. those with a **mismatch** or **confirmed** token. Once it finds a match, the remaining tokens in that fact are assembled into a text string.

with a prespecified format, then passed to the graphics processor for display. A typical output may appear as follows:

```
Configuration Evaluation:

1] Difference in rrcs manifold-1 indication:
   expected open, actual closed.
2] Difference in rrcs manifold-2 indication:
   expected open, actual closed.
3] Difference in rrcs he-press-a indication:
   expected closed, actual open.
4] Difference in rrcs he-press-b indication:
   expected open, actual closed.
```

4 Examples

This section presents a number of examples stressing the various levels of specialization involved in the classifier. Though the real application of the classifier appears in a workstation environment requiring 1440 bit-description inputs. this sequence of cases demonstrates the reasoning capabilities of the system without requiring the normal input or interpreted output. This sequence shows each level of specialization available for full-input classifications.

Default Assumption Given the default fact

```
(default lrcs he-press-a sp-gp)
```

in the `default-values` construct, the `actual switch defaults` rule checks for existence of the facts

```
(actual lrcs he-press-a sp-gp),
(actual lrcs he-press-a sp-op), and
(actual lrcs he-press-a sp-cl).
```

If we say that none of these facts exist, then this rule will fire and assert the fact

```
(actual lrcs he-press-a sp-gp)
```

per the default value.

Discrete Specialization Given the input statements

```
(actual lrcs he-press-a ox-op)
(actual lrcs he-press-a fu-op)
```

the discrete specialization rule matches a combination pattern from the valve discrete summary facts

```
(combine ox-op fu-op vp-op)
```

reducing the two discrete position statements to the one statement

```
(actual lrcs he-press-a vp-op)
```

This process reduces the lowest-level discretes for this valve, *oxidizer valve open* and *fuel valve open*, into the summary statement *valve position open.*

Valve and Switch Assembly Now that the switch and valve positions are available, they can be assembled into one statement that describes the situation about each component. This operation takes two four-field facts, representing *almost* identical information, and creates a five-field fact. Drawing from the examples above, this operation will take the two facts

```
(actual lrcs he-press-a vp-op)
(actual lrcs he-press-a sp-gp)
```

and create the specialized fact

```
(actual lrcs he-press-a sp-gp vp-op).
```

This might seem unusual, but it is actually quite effective. The process of constructing the classification through this point has been one of determining the *appropriate* low-level signatures. By allowing each indication to exist as a single proposition in the early stages, the system has provided a consistent mechanism for managing default values.

Actual/Expected Comparison Each of the steps outlined above is performed for both the actual and expected signatures. The *actual* and *expect* keywords define the environment in which the associated signature applies. In the examples above, the classifier would eventually determine the *expect* fact corresponding to the *actual* fact that was demonstrated:

```
(expect lrcs he-press-a sp-gp vp-op).
```

So far there are no differences between the two modes. But the purpose of the two different signatures is to provide a mechanism for determining the differences between the two. This is performed by the `config mismatch` and `config valid` rules. The `config valid` rule determines whether both states indicate the same values. If they do. then the statement

```
(confirmed lrcs he-press-a sp-gp vp-op)
```

might be asserted, for example. If the two states do not agree. then the `config mismatch` rule takes affect. Suppose the expected state for the `lrcs he press a` valve is something different:

```
(expect lrcs he-press-a sp-cl vp-cl).
```

Then the `config mismatch` rule would fire because the two states for the same component are different, asserting:

```
(mismatch lrcs he-press-a sp-cl vp-cl sp-gp vp-op).
```

This has detected that the valve, expected to be closed. is now open. These two rules possess low salience so that they are not fired until all of the specializations are complete. These rules operate upon components as well as *configurations*, which are described below.

Valve Group Specialization Now that the individual component descriptions have been assembled into the composite facts, collections of these component facts can be specialized into configuration facts. The **valve groups** structure provides the unifying information. For example, assume that the fact

```
(actual frcs tank-isol-12 sp-op vp-op)
(actual frcs tank-isol-345 sp-op vp-op)
```

were generated by the reasoning sequence described above. Given the **valve groups** fact

```
(valve-group tank-isols tank-isol-12 tank-isol-345)
```

then the **specialize group** rule can make the specialization

```
(actual frcs tank-isols sp-op vp-op).
```

Regulator Operation Specialization The most unusual configuration specialization is that of describing the regulator configurations. The propellant tanks have two pairs of regulators each, and can be operated from both, one or none of the individual pairs. Moreover, the switches controlling the plumbing path to these regulators can be in manual or automatic positions. The approach to solving this problem involves the regulator descriptions from **reg desc table**, and steps analogous to those used for other valve components. The rule **reg check** attempts to match associated regulators, A and B, with an entry in this table. If we add the fact

```
(expect lrcs he-press-b sp-cl vp-cl)
```

to the facts considered above, then this fact and the associated one for the A regulator will be matched with the table entry

```
(reg-config sp-cl vp-cl sp-cl vp-cl man regs-0)
```

to create the specialization

```
(expect lrcs reg-config man regs-0)
```

which contains a lot of meaningful intuitive information[6].

Configuration Specialization Now that pieces of each system have been assembled into configurations, the configurations themselves can be collected into even higher–level statements describing each individual system. These specializations are rules only (due to the idiosyncracies of each system), such as **rcs feeding manual**, **active frcs auto**, etc. For example, the **rcs feeding manual** rule states that if the RCS tank isolation and crossfeed valves are all open, then one can conclude that that RCS is providing crossfeed propellant to another system. This terminology is derived from the actual operations lingo, and is quite meaningful to OMS/RCS console operators. The facts generated by this level of configuration specialization contain the keyword config within the fact.

Meta–Configuration Specialization Once the individual system configurations have been determined, it might be possible to assert a more general statement about the "big picture." The *meta-configurations* are essentially *configurations of configurations*. They describe, in one statement, the operational evaluation of all five propellant systems. For input values representing no "problems," the classifier is able to specialize all the way up to this level, deducing a statement such as "Prelaunch configuration." This statement says something about the whole orbiter, and from training flight controllers know that this means the LOMS is feeding crossfeed, the ROMS is active, the RCS systems are in their launch configurations, the OMS regulators are in the auto–closed position, and the RCS regulators are in the manual–open position. Pattern groups representing each of these configurations appear in the rule *prelaunch config*. The effect of this process is to reduce over 100 low–level facts into the one statement

(actual prelaunch config nominal nominal).

Furthermore, the host program interpreter parses this statement to the declaration:

[6] At least to a flight controller.

```
Actual configuration:  PRELAUNCH.
```

5 Enhancements

There are a number of areas for enhancement in the present system. A few of the reasoning extensions are identified below. One obvious quality extension is to change the configuration descriptions to reason about the other orbiter subsystems, such as Data Processing, Life Support, or Electrical Power. Flight controllers responsible for each of these subsystems must monitor telemetry information similar to that monitored for OMS and RCS operations.

5.1 Dynamic Reasoning

Comparing an *actual* signature with an *expected* signature can be interpreted as a matter of temporal persistence. If we can make assumptions about the dynamic behavior of the measured system, then we can draw from knowledge of the *expected* state to help make assumptions about the *actual* state. Often the behavioral assumptions refer to the deduction process, where one might assume *minimum inferential distance* [Touretzky 1986]. Temporal considerations are typically categorized under the *Frame Problem*, as described by Minsky [1975], Hayes [1979], Shoham [1987], Hanks and McDermott [1986], and many others. An interesting enhancement to this system might be found in *predicting the next configuration signature* by incorporating knowledge of procedures and time [Georgeff and Lansky 1987].

5.2 Analog Information

Though the information provided as input to the classifier currently is discrete (binary), there is no reason why analog information may not be added. For instance, some valves on the orbiter do not have discrete position indications, but rather "percentage open" indications. There are published guidelines for interpreting "percentage flow" through these valves that could be implemented as rules with thresholds on their left–hand sides. If a valve is indicating 2 percent open, for example, the interpretation will probably lead to considering this valve closed.

5.3 Instrumentation Failures

A variety of problems may be introduced into the classification process by supplying nonrepresentative signatures as input. There are many orbiter component failures that will cause an invalid signature to be relayed to Mission Control. For example, failure of a computer, demultiplexer, signal conditioner or transducer will cause all of the telemetry measurements associated with those components to be incorrect, without affecting operation of the measured device. These conditions are detectable, however, and can be provided as input to the classifier. When the classifier made aware an instrumentation component failure, and it "knows" the measurements that come from that component, then it can take this invalid information into account when performing the classification. The heuristics for interpreting the actual signature will likely involve *minimum entropy*, *persistence* and *default reasoning.*

6 Evaluation

This classifier performs extremely well for its intended purpose. There is no apparent hindrance to extending the system to incorporate more input or accomodate more configuration models. Adding this configuration evaluator to an existing program shows the capabilities of an add–on expert system. This application derives most of the benefits for developing an expert system outlined by Giarratano and Riley [1989] (the other benefits are not applicable). For example, due to the declarative construction, the system is able to accomodate changes in orbiter procedures without restructuring the inference process. The application performs a complete task, allowing flight·controllers to address their attention to other problems. Most importantly, the expert system is able to perform a mundane task frequently, consistently, and cheaply, and considering the quantity of input, at the level of an expert.

The certified program will be used during all phases of the Shuttle mission to interpret hexadecimal and binary information and to provide a description of the onboard valve and switch configuration. All of the classifications performed thus far in the development process have taken under 6 seconds to complete. This is a highly acceptable amount of time for this activity.

As familiarity with this classifier increases, the users will likely conclude that there are more statements that can be made about spacecraft configurations than have been included in the rule base. There are many subtle descriptions about off-nominal configurations that may prove to be worthwhile in a robust system. The extensibility of the production system will allow such additions to be made without changing the inferencing mechanism or worrying about rule ordering.

References

[Besnard 89] Besnard, *An Introduction to Default Logic*, Springer-Verlag, Berlin, 1989.

[Etherington 88] Etherington, *Reasoning with Incomplete Information*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[Georgeff and Lansky 87] Georgeff and Lansky, "Procedural Knowledge," SRI International Technical Note 411, Menlo Park. CA. 1987.

[Giarratano and Riley 89] Giarratano and Riley. *Expert Systems: Principles and Programming*, PWS-Kent Publishing Company, 1989.

[Ginsberg 87] Ginsberg, *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann Publishers. Inc., Los Altos, CA, 1987.

[Hanks and McDermott 86] Hanks and McDermott, "Default Reasoning, Nonmonotonic Logics, and the Frame Problem," in *Proceedings* *of the Fifth National Conference on Artificial Intelligence*, AAAI, 1986.

[Hayes 79] Hayes, "The Logic of Frames," in *Frame Conceptions and Text Understanding*, Metzing (ed.), McGraw Hill, New York, 1979.

[Minsky 75] Minsky, "A Framework for Representing Knowledge." in *The Psychology of Computer Vision*, Winston (ed.), McGraw Hill, New York, 1975.

[Reiter 78] Reiter, "On Closed-World Data Bases," in *Logic and Data Bases*, Gallaire and Minker (eds.). Plenum Press, New York, 1978.

[Reiter 80] Reiter, "A Logic for Default Reasoning," *Artificial Intelligence 13*, North-Holland, 1980.

[Reiter and Criscuolo 81] Reiter and Criscuolo, "On Interacting Defaults," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.

[Shoham 87] Shoham. "What is the Frame Problem?", in *Reasoning About Actions and Plans: Proceedings of the 1987 Workshop*. Georgeff and Lansky (eds.), 1987.

[Touretzky 86] Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

Embedding CLIPS in a Database-Oriented Diagnostic System

Tim Conway

Senior Engineer, Research & Development
Allied-Signal Aerospace Company
Bendix Test Systems Division, MS 4/8
Teterboro, N.J. 07608

0. Abstract

This paper describes the integration of CLIPS into a powerful portable maintenance aid (PMA) system used for flightline diagnostics. The current diagnostic target of the system is the Garrett GTCP85-180L, a gas turbine engine used as an Auxiliary Power Unit (APU) on some C-130 military transport aircraft. This project is a database oriented approach to a generic diagnostic system. CLIPS is used for "many-to-many" pattern matching within the diagnostics process. Patterns are stored in database format, and CLIPS code is generated by a "compilation" process on the database. Multiple CLIPS rule sets and working memories (in sequence) are supported and communication between the rule sets is achieved via the export and import commands. Work is continuing on using CLIPS in other portions of the diagnostic system and in re-implementing the diagnostic system in the Ada language.

1. Project Overview

The purpose of this project is to develop a generic, database-driven, flightline and/or on-board diagnostic system for electronic and electro-mechanical systems. To re-target the diagnostic system to another device, a new diagnostic database would be generated. The diagnostic system is fully integrated with an on-line hypertext technical manual presentation system.

The initial target for the system is the Garrett GTCP85-180L gas turbine engine. This device is used as an auxiliary power unit (APU) on many military and civilian aircraft. It generates electrical power and pneumatic power used to start the main engines and run systems on the aircraft.

The development environment for this project is a SUN workstation, using "C", CLIPS, and a commercial network database product. The target environment is a portable maintenance aid (PMA) prototype developed by Allied-Signal Aerospace Company, Bendix Test Systems Division. The PMA contains a 68030, 25 Mhz microprocessor with 8 Mb DRAM and 2 Mb PROM, and a 34010 40 Mhz graphics co-processor with 2 Mb DRAM and 1 Mb PROM, a 640 x 480 double super-twist LCD display, a special purpose keypad and a removable 3 Mb SRAM cartridge. The PMA runs a multi-tasking operating system, a Bendix developed windowing environment based on the X windows system, CLIPS, and a commercial network database product. A commercial Analog/Digital data acquisition board is installed, with its own 68000 microprocessor and memory. The PMA weighs 10 pounds, and is 3" x 11" x 16". A removable battery pack, if needed, adds 1" in depth, and 10 pounds.

For this application, a signal conditioning unit, cables, and various sensors connect the PMA directly to the APU for on-aircraft, flight line diagnosis. The signals monitored by the diagnostic system consist of digitized analog signals, such as: Exhaust Gas Temperature (EGT), Oil Pressure (POIL), Compressor Discharge Pressure (PCD), Fuel Pressure (PFUEL), shaft rotations (RPM) and a collection of digital control signals.

2. AI Philosophy

Like any other software approach, AI software techniques have their strengths and weaknesses. A general design guideline for this project is to combine a number of approaches in such a way as to use each for the tasks it does best. In this project, CLIPS (and the RETE algorithm) yields advantages in "many-to-many" pattern matching that procedural programming techniques can not deliver. However, CLIPS is not optimal for our application in implementing the consequences of this pattern matching. "C" code and a network database are more suited to this type of task and are used for this purpose in the system. Such things as window management, the user interface, and the data acquisition subsystem are also implemented in "C". As a result, the CLIPS pattern matching code contains only the minimum data necessary to perform its function. In this way optimum performance is achieved.

3. CLIPS Use: Pattern Recognition Within the Diagnostic Software

In this application, (APU.maintenance) the data acquisition sub-system provides the diagnostic system with a file of data sample records. Each of these records contains a "snapshot" of all of the analog and digital parameters available for the system at that instant in time. The task of the diagnostic system is to extract from this stream of data samples the significant information they contain. This is a two step process, consisting of Event recognition and Pattern recognition.

An Event is any data condition that can be determined from a single data sample record. A history mechanism (the State Vector) allows knowledge of previously recognized events. By definition, an Event occurs at a specific point in time. An example of an Event would be an overheat condition, where an EGT greater than a set amount in any data sample signals an overheat of the APU. Other examples are: transitions of discrete signals, combinations of discrete signals existing simultaneously, and combinations of discrete and analog signals occurring simultaneously.

A Pattern is a higher level concept, encompassing those conditions descriptive of more than one point in time. Patterns exist in hierarchies and can build upon the existence or absence of lower level Patterns and Events. An example of a Pattern would be a failed-lightoff, where an Event detecting sufficient fuel pressure was found, but no Event detecting combustion was seen. Each Event or Pattern can have an effect on the suspicion levels of system components. The suspicion levels of components are adjusted up or down in response to Event and Pattern recognition.

The primary purpose of Event recognition is data reduction. Each data sample "snapshot" is evaluated against each active Event definition once. If the data sample does not trigger any Events, it is discarded. If the sample triggers at least one Event, a copy of it is kept. In this manner, a large proportion of the data samples are discarded and a small number of significant occurrences in the data sample stream are collected for further analysis.

The RETE algorithm performs best when working memory changes slowly. If CLIPS were used for Event recognition, working memory would consist of a single Data Sample record that would change each time a new Data Sample record was acquired. In this case, the benefits of RETE would not be realized. For this reason procedural "C" code is used for Event recognition.

CLIPS is used for Pattern recognition because working memory consists of a set of Event Recognition (ER) and Pattern Recognition (PR) facts which change much more slowly. New facts are added during Pattern recognition based upon successively higher levels of Patterns building on existing facts. In this case, RETE is a very efficient algorithm to use.

The salience feature of CLIPS is useful here as well. Each Pattern is expressed as one or more CLIPS rules. Patterns are expressed in a hierarchy and each Pattern can include conditions that depend on the absence of another lower level Pattern or Event. Rule salience is used here to enforce the hierarchy so that lower level Patterns, that can produce these Pattern Recognition facts are evaluated before Patterns that are conditional on their absence.

4. Database to CLIPS Compilation

The diagnostic system allows multiple sets of Event and Pattern records. For a given target system there is a "root" database, and a number of "child" databases. The root database is the highest level set of Patterns for the target system. An example of a child database would be the Events and Patterns for analyzing the data samples from an APU start-up.

All diagnostic data in the system is stored in database format. This includes root and child Pattern descriptions. To use CLIPS for Pattern recognition, CLIPS rules must be generated for the Patterns. This is done by a "compilation" process on the database. One or more CLIPS rules are generated for each database Pattern. Each rule expresses the Pattern's criteria in logically equivalent CLIPS syntax. A separate file is generated for the root database Pattern rule set and each of the child database Pattern rule sets. This process is accomplished on the host machine and the CLIPS rule set files become a part of the diagnostic database that is downloaded to the target machine.

There are two types of facts that these rule sets work on. These are Event Recognition (ER) facts, and Pattern Recognition (PR) facts. Both types of facts contain fields identifying the database number and identifying number of the Event or Pattern the fact represents. Each type also contains a unique sequence number to distinguish it from all other facts of its type and to allow multiple occurrences of the same ER or PR to exist, each with different sequence numbers. ER facts will also contain a copy of the analog and digital parameters of the data sample which caused their recognition. The digital parameters follow the analog parameters, with each group of 16 discrete parameters compressed into a single 16 bit integer.

Figure 1 shows the organization of a typical Pattern record, and its related records in the database. Each

Pattern can have multiple Event Criteria (EC) records associated with it. Each of these forms a single logical test which in turn is translated into a single CLIPS Left Hand Side (LHS) condition. An EC record can represent either the existence or absence of an ER fact or PR fact. (i.e. the prior recognition or lack thereof of either an Event or Pattern) For ER facts, additional Parameter Condition (PC) records may be specified to further test the analog or digital parameters. These tests can be against constants, other parameters in the same ER fact, or other parameters in another ER fact under another EC for the same Pattern. This allows conditions such as: "the fuel pressure at the 35% RPM Event is at least 10 psi higher than the fuel pressure at the 10% RPM Event". Parameter Conditions also have tolerance ranges which allow for sensor errors and other types of inexact conditions.

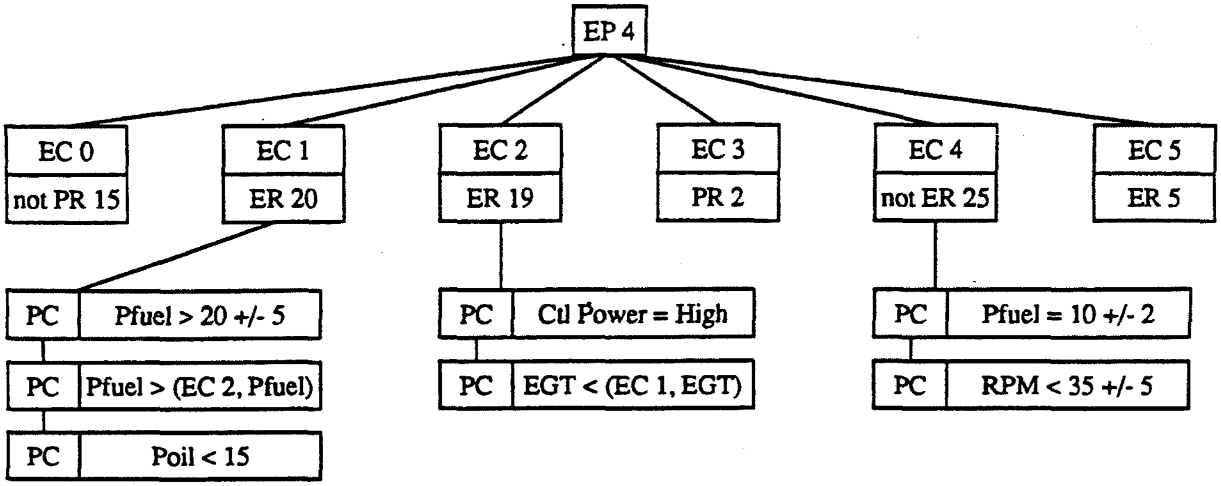


Figure 1 - Typical Pattern instantiation in the database.

Two problems related to CLIPS syntax must be dealt with. First, the first LHS condition of a rule may not be negated. Second, no named fact field can be used in a comparison until after the name has been declared. The first problem can be alleviated by ordering the LHS conditions generated such that non-negated PRs and ERs precede negated ones. This way, if there are any non-negated LHS conditions, they will appear before any negated LHS conditions. While it is still possible for a pathological Pattern to be written with no non-negated ECs, this can be checked for by loading the rule set on the host system after it has been generated. Pattern rules that fail for this reason can then be re-written to add at least one non-negated ER or PR. Figure 2 shows the Pattern from Figure 1 reorganized to move the non-negated ER and PR elements to the start of the rule.

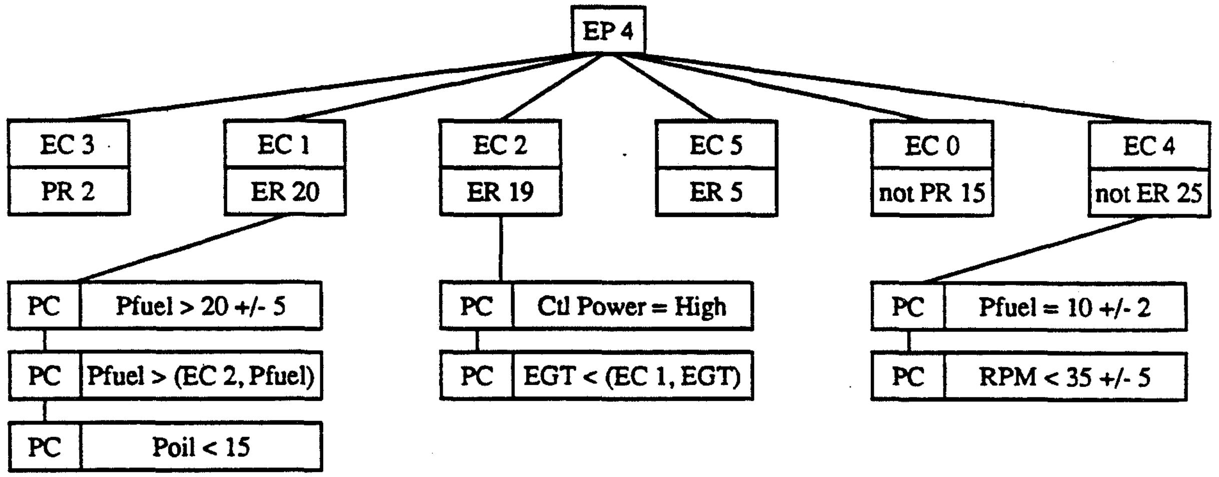


Figure 2 - Pattern from Figure 1 partially reorganized for CLIPS rule compilation.

The solution to the second problem is shown in Figure 3. Since we have now established the order that the EC LHS conditions will be written, we can now go through them looking for "forward references". These can occur when a Parameter Condition (PC) under an ER fact references a parameter which has not been declared by that point in the compilation. When this occurs, we can "reverse" the condition, and attach it to the forward referenced LHS condition.

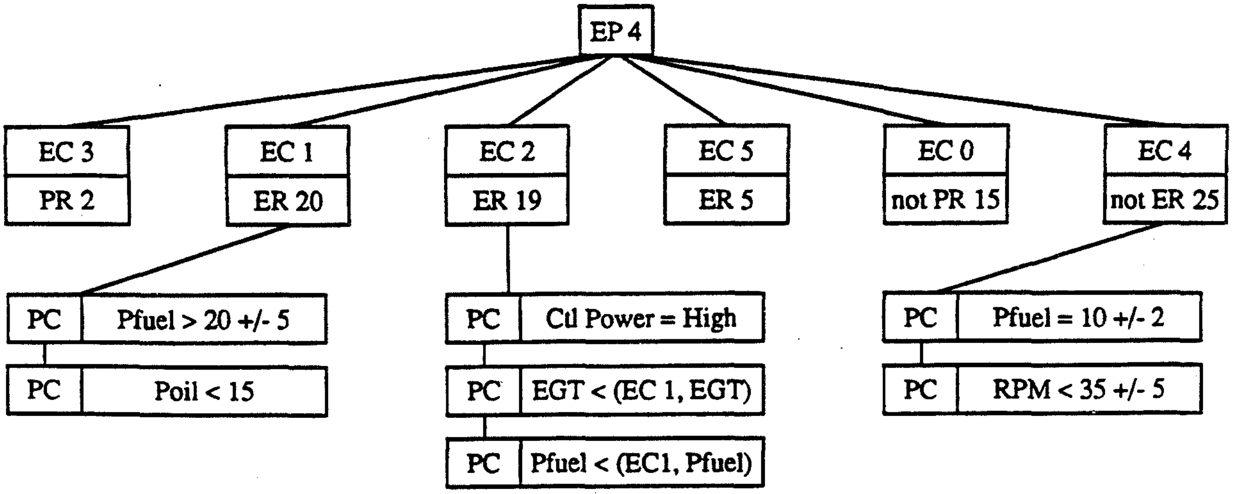


Figure 3 - Pattern from Figure 1 fully reorganized for CLIPS rule compilation.

In Figure 2, a left-to-right compilation of ECs into LHS conditions will fail at the second PC under EC 1. This PC contains a reference to an element (Pfuel under EC 2) that has not been declared yet. To resolve the problem, the condition is reversed, and placed under the forward referenced EC. (EC 2) This is shown in Figure 3. This creates an equivalent logical condition, in a form that CLIPS can digest.

```
; EP # 4 translated

(defrule DB-001-EP-00004 "test rule"
      (declare (salience 290))
      (PR 2 2 $1)
      (ER 2 20 ?ER01 ? ?EC1Ltime ?EC1Time
                  ?EC1Poil&:(< ?EC1Poil 15)
                  ?EC1Pcd
                  ?EC1Pfuel&:(> ?EC1Pfuel 25)
                  ?EC1EGT ?EC1RPM
                  ?EC1S7d )
      (ER 2 19 ?ER02 ? ?EC2Ltime ?EC2Time ?EC2Poil ?EC2Pcd
                  ?EC2Pfuel&:(< EC2Pfuel EC1Pfuel)
                  ?EC2EGT&:(< EC2EGT EC1EGT)
                  ?EC2RPM
                  ?EC2S7d&:(DCc ?EC2S7d 2 1 0))
      (ER 2 5 ?ER05 ? ?EC5Ltime ?EC5Time ?EC5Poil ?EC5Pfuel ?EC5EGT ?EC5RPM
                  ?EC5S7d)
      (not (PR 2 15 $?))
      (not (ER 2 25 ?ER04 ? ?EC4Ltime ?EC4Time ?EC4Poil ?EC4Pcd
                        ?EC4Pfuel&:(&& (>= EC4Pfuel 8) (<= EC4Pfuel 12))
                        ?EC4EGT
                        ?EC4RPM&:(< EC4RPM 30)
                        ?EC4S7d)
=>
      (call (NewPR 2 4)))
```

Figure 4 - CLIPS rule generated from compiling EP 4.

Figure 4 shows the CLIPS rule generated from the reorganized Pattern in Figure 3. The rule name is generated from the database number and identifying number of the Pattern. A more descriptive name is placed as a comment next to the rule name. The salience of the rule is computed to match the Pattern's place in the Pattern hierarchy. Next, the ECs under the pattern are each evaluated and categorized as either negated ERs, non-negated ERs, negated PRs, or non-negated PRs. At this time, the sequence field of each EC's ER or PR fact is given a unique name to allow comparisons of data fields between facts.

Comparisons of analog parameters are done directly in CLIPS code. Comparisons of discrete parameters are done in external routines. "DCc" is an external routine to compare a discrete value with a constant. "DCr" is a similar routine for a relative comparison of a discrete with another discrete. Tolerance ranges are allowed for any comparison. This means that a condition such as "fuel pressure equals 10 psi, +/- 2 psi" would be expressed logically as "((PFUEL <= 12) and (PFUEL >= 8))". Similar condition effects are created for other logical operators.

Lastly, the consequences of each rule are implemented in an external routine, "NewPR". This routine will affect the Ambiguity Group (suspicion) ranking, and will assert a PR fact to indicate recognition of this rule's Pattern.

There are a few additional rules for Pattern construction. For example: relative referenced PCs (i.e. Parameter Conditions under an EC that reference fields in facts other that the fact the parent EC is referencing) are only allowed to reference non-negated ERs. The reason for this is that negated ERs are by definition not present when the rule is evaluated True. Therefore, comparisons of fields of a non-existent fact makes no sense. The compilation process flags these as errors.

5. Database Pattern Sets at Run Time

Figure 5 illustrates the relationship between the root and child databases. The diagnostic system is started with the CLIPS rule set for the "root" database loaded. (i.e. the "rules" file from "root DB" in Figure 5) The root database is the highest level set of Patterns for the target system. When a specific test is to be run, the facts from the CLIPS rule set for the root database are exported to the root database's "facts" file. Then the current CLIPS rules and facts are cleared and the rule set for the selected child database is loaded from its associated "rules" file.

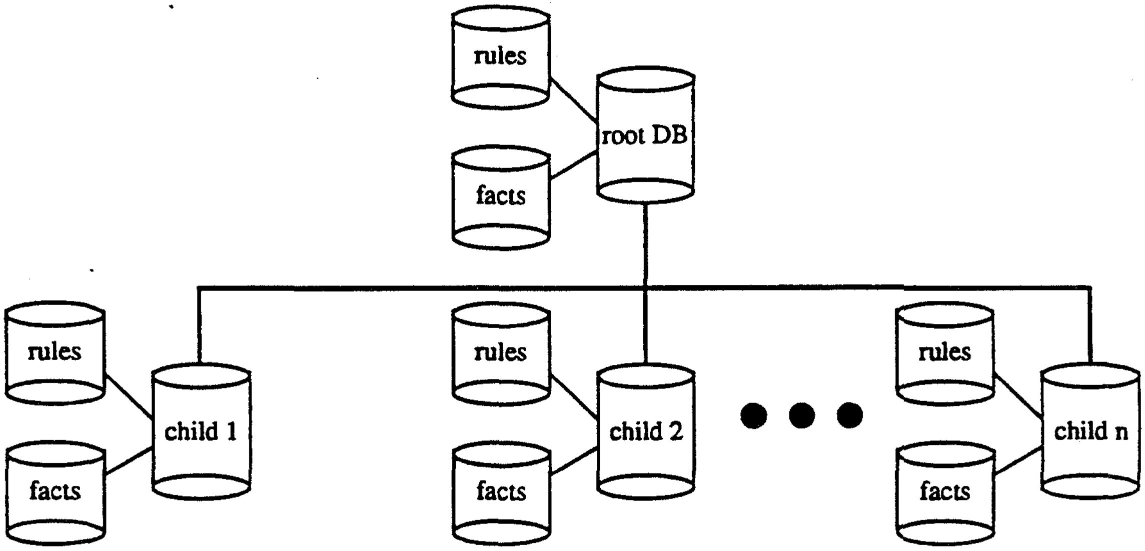


Figure 5 - "Root" and "child" database relationships.

The child Event and Pattern set is then run against the input Data Sample records. Events and Patterns in the child database affect the Ambiguity Group (suspicion levels) in the same manner as root Events and Patterns do. When the conflict set is empty for the run of the child database, Event Recognition (ER) and Pattern Recognition (PR) facts are exported to the "facts" file for the child database. The current CLIPS rules and facts are then cleared and the root database rule set is re-loaded from its "rules" file.

The exported "facts" file from the root database is then re-loaded and a global flag is set to disable the external consequences of rule firings. The root database is then run until the conflict set is empty, causing the root database rule set to return to its last interrupted state without repeating any external effects already accomplished.

Next, the exported "facts" file from the child database is imported and the global flag is reset to allow external consequences of rule firings. The rule set is then run until the conflict set is again empty. In this manner, the root database rules can incorporate additional knowledge from the child database rule firings. This allows still higher level conclusions to be drawn by the root database about conditions that can not be determined in a single test run. Examples of this would be tracking the degradation of the unit over time or evaluating the results of a calibration or adjustment.

When facts are created by a database, one of the fields in the fact is an "export" flag. When facts are exported, only facts with the export flag set are exported. This allows some degree of purely "local" reasoning to take place in child databases, without burdening the root database with every intermediate conclusion in the child database's reasoning process. Child database Patterns are written with the export flag set only for the "highest" level of Patterns in the child database. These are conclusions that the root database can logically do further reasoning on.

6. Future work

This project is part of a continuing research and development effort to improve flightline and on-board diagnostics and monitoring capabilities for complex electro-mechanical systems. Work is continuing on enhancing the basic capabilities of the system to include such things as better explanation capabilities. CLIPS may be used in other portions of the system wherever it will improve performance and/or capabilities. Work is also underway to re-implement the "C" portions of the system in Ada. A new generation of prototype PMA target hardware is under construction. This will make the system more powerful and easier to embed within a target system.

Work is also progressing on the Knowledge Editor. This is a workstation-based set of tools for analyzing target system data sample sets and creating diagnostic databases. A Script Editor is being written to aid in the creation of the on-line manuals needed for diagnostics and maintenance.

UFC Advisor: An AI-Based System for the Automatic Test Environment

David T. Lincoln *
Pamela K. Fink, Ph.D.

Southwest Research Institute
San Antonio, Texas 78228-0510
(512) 522-3368

Abstract

The Air Logistics Command within the Air Force is responsible for maintaining a wide variety of aircraft fleets and weapon systems. To maintain these fleets and systems requires specialized test equipment that provides data concerning the behavior of a particular device. The test equipment is used to "poke and prod" the device to determine its functionality. The data represent voltages, pressures, torques, temperatures, etc. and are called testpoints. These testpoints can be defined numerically as being in or out of limits/tolerance. Some test equipment is termed "automatic" because it is computer-controlled. Due to the fact that effective maintenance in the test arena requires a significant amount of expertise, it is an ideal area for the application of knowledge-based system technology. Such a system would take testpoint data, identify values out-of-limits, and determine potential underlying problems based on what is out-of-limits and how far. This paper discusses the application of this technology to a device called the Unified Fuel Control which is maintained in this manner.

* formerly with SAALC/MAT, Kelly A.F.B., San Antonio, Texas 78241-5000

Introduction

The Air Force maintenance capability is primarily organic in that Air Force personnel perform the diagnosis and repair tasks. Much of the test equipment and the devices they support were developed and fielded in the early- to mid-seventies. Thus, most of the equipment tends to be out-moded and no longer supported by the vendor. Therefore, use of such equipment to diagnose a device requires a certain level of expertise obtained over years of experience. For example, a minimum of ten years of experience is needed to produce an experienced diagnostician for the Unified Fuel Control (UFC).

The UFC is the "carburetor" for the F-100 engine, the engine that flies the F-15 and F-16 fighter jets. It is essentially a large, complex mechanical computer. Nearly 95% of all UFC's in the Air Force's inventory are repaired and tested at the San Antonio Air Logistics Center (SAALC) at Kelly A.F.B. The controls arrive at SAALC for one of two reasons: scheduled overhaul or unscheduled maintenance. A UFC will be scheduled for overhaul when it exceeds the Air Force's recommended maximum operating hours (MOH). Depending on whether the UFC is taken from an F-15, which has two engines, or an F-16, which has only one engine, and the configuration of the UFC, this MOH can vary from 1500 to 4000 hours. UFC's arrive for unscheduled maintenance due to a malfunction that can be caused by a variety of problems. When a UFC arrives from the field it has a processing tag attached to it. This tag contains the problem description as reported by the field, which ranges from very specific (e.g. broken lever arm) to very vague (e.g. does not work).

Determining what could be causing a malfunction can be very difficult. The UFC is composed of over 4500 parts, many of which can cause the control to fail. The test equipment used to maintain the UFC is a customized piece of automatic test equipment and is referred to as a test stand. A test stand is analogous to an electronic diagnostic system one might find at a car repair shop. The UFC is connected to the test stand and run through a series of tests to determine its weaknesses, just as a car's engine might be. An expert in diagnosing the UFC must take into account not only potential problems with the UFC, but the possibility that the test stand may not be within calibration standards. In addition, the UFC is maintained by a set of four different test stands, each with a specific set of test procedures to help diagnose certain parts of the UFC. Thus, the number of possible failures and their underlying symptoms is large, creating a need for very domain-specific expertise.

The UFC Maintenance Process

To standardize the decision making strategy for the maintenance process of the UFC, SAALC uses the concept of On-Condition Maintenance (OCM). This concept is one in which a team of domain experts is chosen to make all decisions concerning the repair of a UFC as it passes through the maintenance process. These decisions are based on the UFC's condition upon receipt at the maintenance facility and at various points during testing. An overview of the entire maintenance process is given in Figure 1. There are six potential areas where knowledge-based system technology could be applied. They include the pre-RAR decision, the post-RAR decision, the Augmentor Body, Gas Generator, and Distribution Body decisions, and the post-M&I decision. Each of these systems would utilize the information available at a given point in the process to form recommendations about what should be done next.

The UFC maintenance process begins with a visual and electrical inspection. The results of these inspections, along with the field reported problem description, give the OCM team personnel a foundation for their first decision: overhaul, demate and repair, or run the Run-As-Received (RAR) test. To overhaul a UFC requires breaking the control down to its lowest levels and replacing defective parts as it is rebuilt. The average length of time required to do this is 650 hours. To demate and repair means to break down the UFC to one of its three major sub-assemblies (Augmentor Body, Gas Generator, and Distribution Body) and perform the prescribed repair actions.

The RAR test is actually a series of automatic tests that are run to give diagnostic information about what might be wrong with the UFC. It is hosted on a Data General computer and is run "hand's off" (i.e. no adjustments made as the test runs). The time required for this test averages seven hours but can go as long as twelve or fourteen. The computer, in turn, drives the test stand that "pokes and prods" the UFC. The RAR generates approximately 450 testpoints and records the UFC's value at each testpoint. The result of the RAR is a one inch thick document with the various testpoints grouped into related paragraphs which represent the three distinct sub-assemblies of the UFC. The RAR is then analyzed by one or more members of the OCM team and, based on this analysis and the team members' experience level, a recommendation is made as to the best repair action. This recommendation may include overhaul, demate and repair, or run the Mating & Indexing test (M&I). The M&I involves the calibration and adjustment of the UFC. If the UFC has been overhauled or demated and repaired, it is then reassembled and run through the M&I. The M&I and the RAR both test the UFC with the same tolerances. Once the M&I has finished, another iteration of decision making is made: overhaul, demate and repair, or run the Service Acceptance Test (SAT). The SAT is essentially the same test as the RAR and M&I with a different set of tolerances. Once the UFC passes the SAT, it is returned to the Air Force inventory.

Although three shifts are required to meet the demand for UFC production, the OCM team is only available during the first shift. During the second and third shift and on weekends, test recommendations are left up to the line or shift supervisors, or the UFC is put on hold until an OCM team member is available. Thus, delays are inevitable in obtaining a diagnosis for a UFC. A crucial task performed by the OCM team that is vital to an accurate diagnosis is visually identifying all testpoints on the RAR that are out of limits. Due to the stress that is placed on the OCM team to produce, there is a good probability that some of the testpoints that are out of limits are not identified. This naturally leads to erroneous and inconsistent decisions.

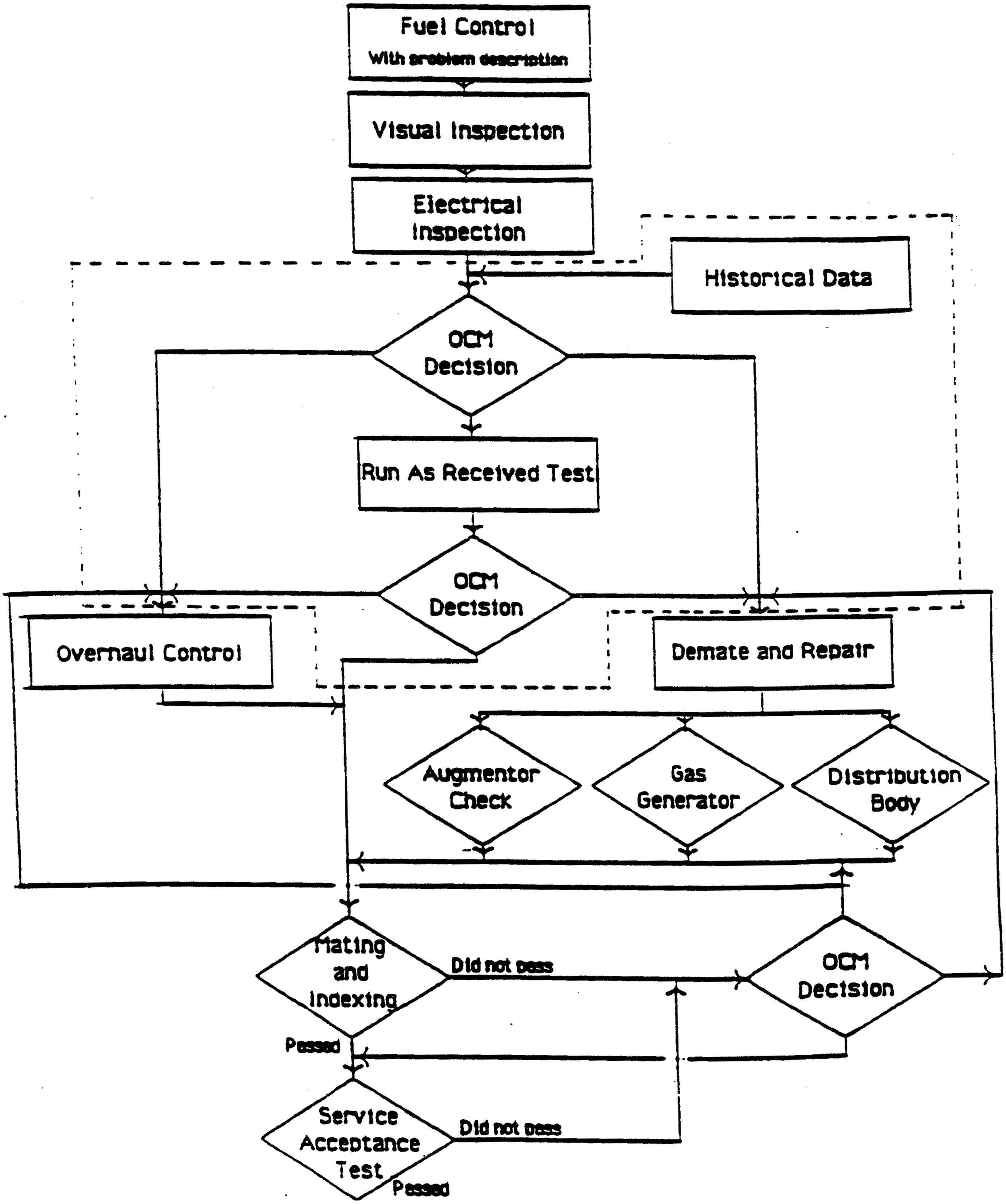


FIGURE 1. Overview of the UFC maintenance process.

Issues Concerning the Development a Knowledge-Based System for the Automatic Test Environment

Of the six potential areas where a knowledge-based system could be implemented, the pre-RAR and post-RAR (hereafter referred to as the UFC Advisor) were selected to start with because they are procedures that most UFC's must undergo and because the problems of integration into an existing test environment were not so severe. These initial phases of the maintenance process are not highly interactive and so did not have to be performed out on the shop floor next to the test stand (a volatile environment). The pre-RAR system is basically a front-end to the historical database shown in Figure 1 that allows the user to enter preliminary data about each UFC as it comes in from the field and to obtain the data on the UFC from previous repair actions.

The UFC Advisor was developed as an effort to streamline the maintenance process and increase the production of UFC's at Kelly A.F.B. Since the experts perform diagnoses from a problem-oriented standpoint, the UFC Advisor is designed to mimic this approach. It makes recommendations based on the RAR test results and furnishes three benefits with respect to the RAR:

- ensures identification of all testpoints out of tolerance
- provides consistent recommendations
- reduces time lost due to the unavailability of the OCM team on second and third shifts

The UFC Advisor was developed as a joint effort between civil service computer scientists and engineers and researchers from Southwest Research Institute. This cooperative effort was one in which the civil service employees acted as apprentices to the more experienced researchers, with the intention that the Air Force would gain an organic capability in artificial intelligence/knowledge-based systems development.

As with any knowledge-based system development, a decision had to be made as to the type of hardware that would host the system and, since many knowledge-based system shells/languages are hardware dependent, which shell or language would best fit the needs for the UFC Advisor. Additionally, data acquisition from the UFC test stands was non-trivial. As stated before, much of the test equipment used in the maintenance process in the Air Force is out-dated. This is true of the UFC test stands. Because these stands are so old, the test data generated is often only accessible at the test stand. This is not a problem when a human is interpreting the test data since he/she can easily read the test stand's screen or the printout to obtain the testpoint out-of-limits data. However, acquisition of such data electronically could be very difficult.

The ideal solution would have been to host the UFC Advisor on the Data General computers that run the test stands, but these computers, which were designed and implemented in the mid-seventies, have only 256k of RAM with memory virtually exhausted and no capacity for expansion. The development team also concluded that the UFC Advisor would be too large to run in a PC environment and so decided that a workstation would be suitable since a workstation has both the memory and speed required to run a system as large as the UFC Advisor. In addition, a workstation is less expensive and more compact than a mainframe. After comparing the Apollo, SUN, and VAX workstations, the SUN was chosen for development. Due to an unexpected hindrance, the development team realized that it would take six months for SUN to deliver the workstations. Thus, an interim decision was made to prototype what would fit of the UFC Advisor on an IBM PC. Then, upon arrival of the workstations, the knowledge could be transferred from the PC to the SUN and expanded to completion.

As to the choice of a software language tool, CLIPS was chosen over many others for a variety of reasons. First, CLIPS was available so it was chosen as the tool to use for development of the initial prototype on the PC. The development team also knew of CLIPS' portability and decided to continue to use it since there was no reason to believe that CLIPS code designed on the PC would not run on the SUN. Second, acquisition of software by the government is slow. In view of the fact that CLIPS is supplied to government agencies at no cost, the normal delay expected to obtain a specialized knowledge-based system development tool such as CLIPS is eliminated. Another advantage CLIPS possesses is its capability of being embedded in an application program written in a conventional language such as C.

Once CLIPS was chosen the next step was to acquire the data from the test stands. As stated before, this acquisition turned out to be very difficult. The initial suggestion was to take the RAR data from the Data General and port it to the SUN, but again the Data General's are virtually out of memory and thus had no capacity to host another software progam which might write the RAR data into a format understandable to the SUN. The next idea was to eavesdrop on each test stand's printer and capture the RAR data with a PC located at each test stand as the data printed out to the printer and then transfer the data by floppy to the SUN. But the Air Force's requirement that any computer equipment located in the test stand area be enclosed in plastic because of the explosive nature of the fuel used to test the UFC, along with the fact that there are over twenty UFC test stands, made it economically unreasonable to use this approach. It was also unrealistic to expect an OCM team member to type in over 450 testpoint values at a terminal. It was still necessary, though, to acquire the data quickly since the RAR data remains memory resident for only thirty minutes. Given shift changes, employee's lunch and scheduled breaks and other unforeseen delays, many of the RAR's could be lost.

The solution decided upon was to monitor each test stand's printer through a series of specialized buffering hardware. The data is shipped over an ethernet that connects each test stand to one of several communications boxes. These boxes then ship the data to a single PC where the data is identified by UFC serial number and undergoes preliminary analysis, storing only what is needed. When it has been determined that all data for an RAR on a given UFC has been obtained, the file is closed and sent to the SUN where the UFC Advisor resides.

The UFC Advisor

The UFC Advisor essentially has no user interface. Under normal operations the system automatically receives over the network a file containing testpoint values from an RAR. When analysis is complete, the system prints out its final report. In case something does go wrong, however, the system does provide a facility for querying about the status of the data on all of the UFC's in the system at that point in time.

The UFC Advisor is a single executible program composed of three parts: a C program to preprocess the data input from the PC, a second C program to read the processed file and test all of the RAR testpoints for in- or out-of-limits condition, and a "diagnostic inference engine". Each of these programs will be discussed in detail. An overview of the total UFC Advisor system architecture in shown in Figure 2.

The preprocessor is essentially a parser and is designed to strip all irrelevant information from the file received from the PC. It also removes duplicate paragraphs, as an RAR may run the same paragraph more than once. If the file contains errors, it is copied into a directory to be corrected by an OCM team member. If there are no errors, the file is read by the second C program.

This second program begins by initializing CLIPS. Then it reads in each testpoint value and determines whether the testpoint is low, high or within limits based on a predefined minimum/maximum file. If the value is out-of-limits, then a string, which contains information such as which subsection (or paragraph) of the UFC contains the testpoint, the testpoint number, its out-of-limits value (i.e. high or low) and its actual value, is written into a "symptoms" files. Also, all testpoints, along with their recorded, minimum, and maximum values are written to an output file, with testpoints that are out-of-limits highlighted by an asterisk. This process is reiterated for every testpoint in the RAR. Upon completion, the diagnostic inference engine assumes control.

The diagnostic inference engine, which was designed and implemented in CLIPS, (ver. 4.2), is a seventy rule knowledge-based system. Each iteration of the system performs a series of steps. It has been designed as a generic diagnostic inference engine to handle association of testpoints out-of-limits with problems and solutions. First, it reads the "symptoms" file and asserts each string (or symptom) as a fact. An example of a fact is

```
P 9003 tp 10 ITEM PFN-PFCB OOL high RCRD 57
```

where 'P 9003' indicates paragraph 9003, 'tp 10' is testpoint 10, 'ITEM PFN-PFCB' is a subcategory of the testpoint, 'OOL high' means out-of-limits high and 'RCRD 57' is the recorded value for the testpoint. The second step involves loading into memory the knowledge that has been acquired from the experts. The knowledge is grouped by paragraph number, where each paragraph is stored in a separate file. It is in the form of CLIPS facts. This set of files comprises the test-specific knowledge base. Thus, to modify the knowledge base simply requires modification of the file which contains the information about the paragraph in question.

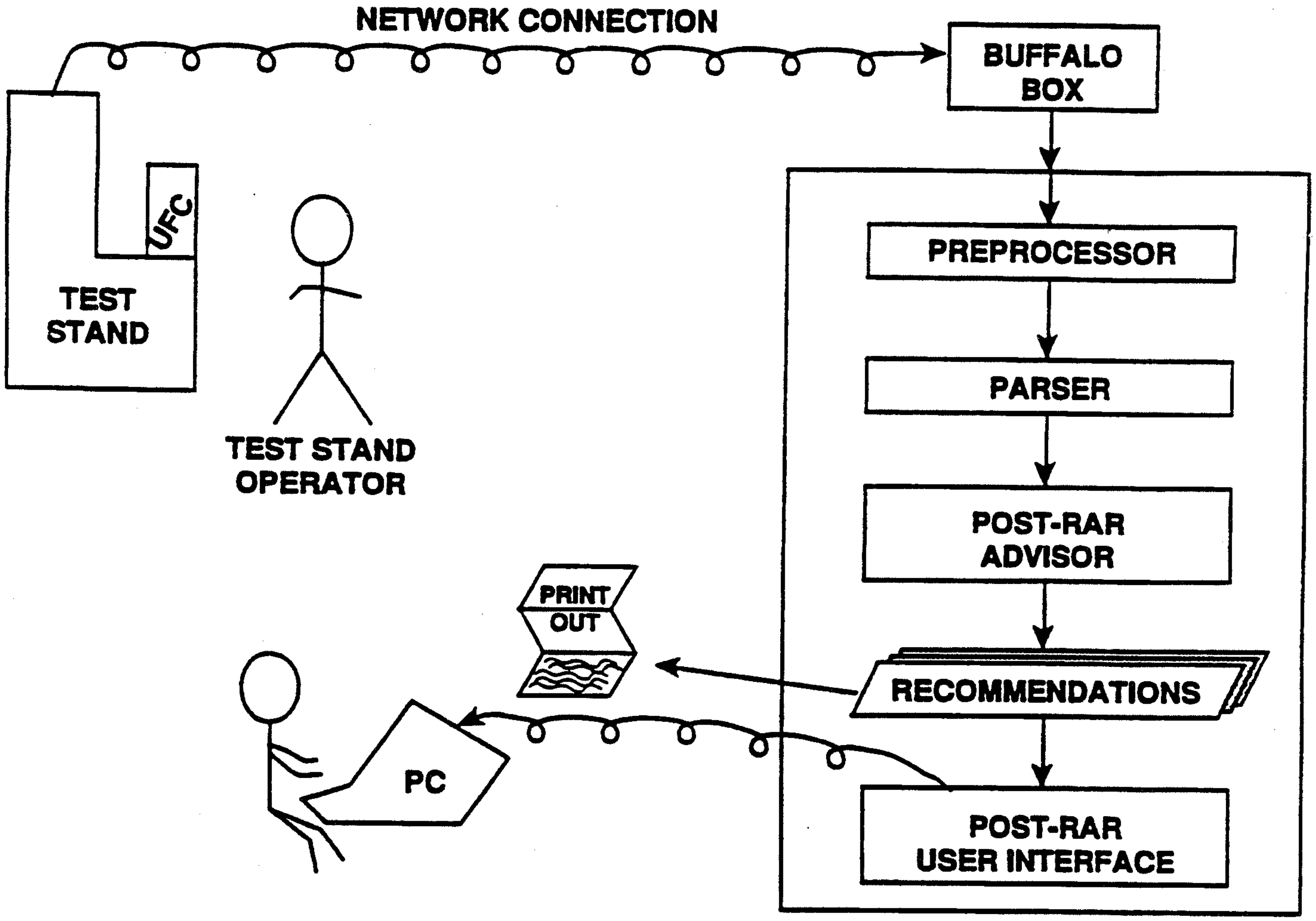


Figure 2. Configuration for the UFC-Advisor System

Since each paragraph is loaded as a fact, changes to the knowledge base do not require a recompilation of the rules. Each fact in the knowledge base has associated with it a symptom, the minimum and maximum value for the symptom's testpoint, a potential problem for that testpoint, evidence for that problem, a possible solution to the problem and the cost to perform that solution. For each symptom, there may be one or more symptom/problem/solution sets associated with it. An example of one of these facts is:

```
SYMPTOM:  P 9003 tp 10 ITEM PFN-PFCB OOL high
MIN 37.5 RCRD dummy MAX 42.0
PROBLEM:  Contamination of speed receiver orifice
EVID:  5
SOLUTON:  Decontaminate speed receiver orifice
COST:  0.5
```

The third step of the diagnostic inference engine utilizes a set of rules that match each symptom from the first step with each symptom/problem/solution set in the second step. Each matching set is then retracted and reasserted with the RCRD field of 'dummy' replaced with the testpoint's actual value. Since many symptom/problem/solution sets have the same symptom associated with them, use of a value like 'dummy' prevents the system from only capturing the first occurrence of a matching set and bypassing the rest. Next, all unused symptom/problem/solution sets (i.e. those with 'RCRD dummy') are retracted to release memory. Many problems may have multiple symptoms and/or solutions and as mentioned before, the UFC Advisor attempts to diagnose from a problem-oriented standpoint.

To further complicate the diagnostic process, discussions with the experts revealed that key testpoints, when out-of-limits, forced repair actions that had to be dealt with immediately. This knowledge is referred to as meta-knowledge. A second set of testpoints, while not requiring immediate action, had priority over all others. Thus, a level of meta-knowledge, plus prioritization of the problems, became necessary. To handle the issues of meta-knowledge and prioritization, a method of evidence maintenance was used.

First, for each unique problem a tally is initialized. Then, all problems that match a tally are combined by combining their evidences. Also, if the paragraph affiliated with a specific symptom/problem/solution set is one with priority over the others, the evidence is multiplied by a "priority factor" before being added. After all sets have been tallied, they are sorted based on total evidence. Next, a set of "meta-rules" execute based on the meta-knowledge obtained from the experts. The purpose of firing these rules now and not initially is two-fold. First, the development team, following the expert's advice, decided to print out all recommendations rather than using a minimum threshold based on evidence. Second, by firing last the meta-rules can write directly to the output file as the first set of recommendations. Figure 3 gives an example of a portion of the UFC Advisor's output. A typical output is around ten to twelve pages.

Finally, the symptom/problem/solution sets associated with the tallies are written to the output file in order of evidence. As one can see from Figure 3, these sets may contain one or more solutions for each problem with one or more symptoms for each solution. Additionally, along with the minimum, recorded, and maximum values, the cost for each solution is written. Thus, the output consists of three parts: a summary of testpoint information, meta-rule recommendations, and all other recommendations, listed by priority.

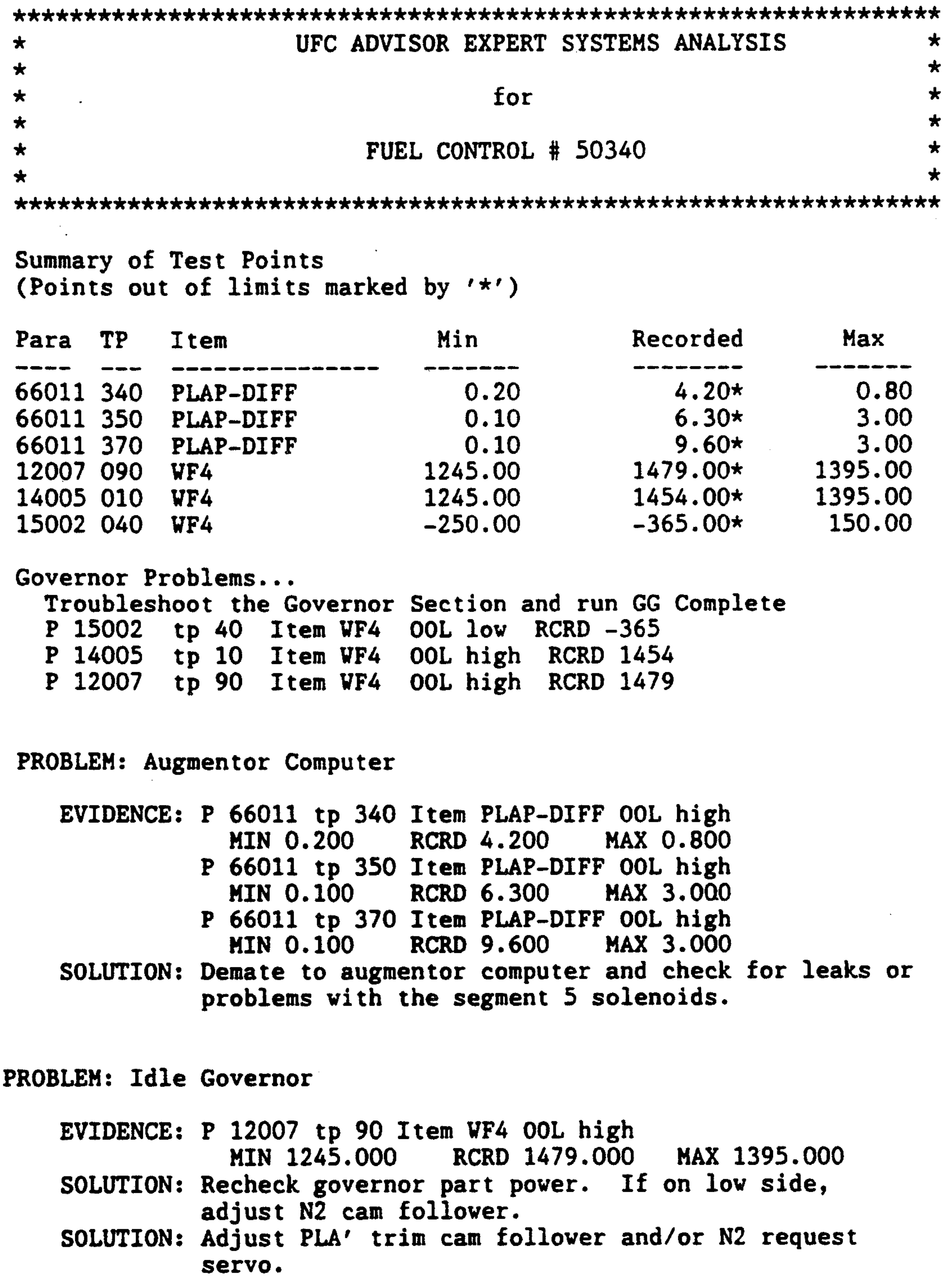
Current Status

At the present time, all record keeping in the UFC maintenance area is paper-oriented. The current method for storing records is to package the RAR, M&I, SAT and all other written documentation into a plastic bag and store the package in a filing cabinet. Thus, to gather any statistical information such as a testpoint that is a recurring problem, occurrences of less frequent but highly critical repairs, or any correlation of testpoints out-of-limits to solutions is almost impossible.

The UFC Advisor as it currently stands, where it is capable of supporting the RAR test has the potential for saving considerable test stand and OCM team time each month. Based on an analysis of the entries in the UFC Test Log for the one month period of August 1989, 25% of the UFC's that came in had an RAR run, with an average run time of 18.2 hours. The average time spent after an RAR was run and waiting for a recommendation from the OCM team was 9.25 hours. The total wait time after an RAR was run was 360 hours, or approximately 15 24 hours days. This equates to half a test stand per month being wasted on just waiting on the decision that has to be made after an RAR is run. In addition, each RAR evaluation requires 30 - 60 minutes of an OCM team member's time. As a result, approximately 36 hours per month of an OCM team member's time could be saved, allowing them more time to spend on the more complex problems and not delay the simpler ones. Thus, the UFC Advisor could save considerable time just where the RAR is concerned.

In addition, because the M&I and RAR tests are so similar, the system is capable of supporting the M&I test. This is because the recommendations that the system makes are often concerned with the adjustments and replacements that could be made to bring testpoints into limits during an M&I. Since the M&I test is operator-intensive, any time savings would increase both test stand and operator availability considerably.

The design of the UFC Advisor centers around the linking of testpoint out-of-limits data with possible problems and then linking possible problems to possible solutions. These linkages are provided as static knowledge in the UFC Advisor. The dynamic knowledge in the UFC Advisor is then essentially a diagnostic inference engine, implemented in CLIPS, than can utilize the linkages to identify potential problems, prioritize the problems and solutions, and write a report containing recommendations on what to do next. This diagnostic inference engine is a very general tool that could be utilized in any knowledge-based system development effort that is to interpret testpoint data and provide recommendations. Only the static knowledge containing the information linking testpoints out-of-limits to problems and solutions would have to be changed to fit the new device being tested.



```
*****************************************************************
*               UFC ADVISOR EXPERT SYSTEMS ANALYSIS             *
*                                                               *
*                              for                              *
*                                                               *
*                       FUEL CONTROL # 50340                    *
*                                                               *
*****************************************************************

Summary of Test Points
(Points out of limits marked by '*')

Para  TP   Item              Min           Recorded        Max
----  ---  --------------  -------         --------      -------
66011 340  PLAP-DIFF          0.20            4.20*         0.80
66011 350  PLAP-DIFF          0.10            6.30*         3.00
66011 370  PLAP-DIFF          0.10            9.60*         3.00
12007 090  WF4             1245.00         1479.00*      1395.00
14005 010  WF4             1245.00         1454.00*      1395.00
15002 040  WF4             -250.00         -365.00*       150.00

Governor Problems...
  Troubleshoot the Governor Section and run GG Complete
  P 15002  tp 40  Item WF4  OOL low  RCRD -365
  P 14005  tp 10  Item WF4  OOL high  RCRD 1454
  P 12007  tp 90  Item WF4  OOL high  RCRD 1479


PROBLEM: Augmentor Computer

   EVIDENCE: P 66011 tp 340 Item PLAP-DIFF OOL high
               MIN 0.200    RCRD 4.200    MAX 0.800
             P 66011 tp 350 Item PLAP-DIFF OOL high
               MIN 0.100    RCRD 6.300    MAX 3.000
             P 66011 tp 370 Item PLAP-DIFF OOL high
               MIN 0.100    RCRD 9.600    MAX 3.000
   SOLUTION: Demate to augmentor computer and check for leaks or
             problems with the segment 5 solenoids.


PROBLEM: Idle Governor

   EVIDENCE: P 12007 tp 90 Item WF4 OOL high
               MIN 1245.000    RCRD 1479.000    MAX 1395.000
   SOLUTION: Recheck governor part power.  If on low side,
             adjust N2 cam follower.
   SOLUTION: Adjust PLA' trim cam follower and/or N2 request
             servo.
```

FIGURE 3. Example of a portion of the UFC Advisor's output

omit

B13 Session:
Advisory Systems I

S37-63

N96-12950

p. 8

358825

EXPERT SYSTEM FOR SCHEDULING SIMULATION LAB SESSIONS

By Chet Lund   *Lockheed Engineering & Sciences Company*
2400 NASA Road One MC/C07
Houston, TX 77058

ABSTRACT. . Implementation and results of an expert system used for scheduling session requests for the Systems Engineering Simulator (SES) laboratory at the NASA Lyndon B. Johnson Space Center (JSC) are discussed. Weekly session requests are received from astronaut crew trainers, procedures developers, engineering assessment personnel, software developers, and various others who wish to access the computers, scene generators, and other simulation equipment available to them in the SES lab. The expert system under discussion is comprised of a data acquisition portion - two Pascal programs run on a personal computer - and a CLIPS program installed on a minicomputer. A brief introduction to the SES lab and its scheduling background is given. A general overview of the system is provided, followed by a detailed description of the constraint-reduction process and of the scheduler itself. Results from a ten-week trial period using this approach are discussed. Finally, a summary of this expert system's strengths and shortcomings are provided.

INTRODUCTION

The Systems Engineering Simulator (SES) lab at the NASA Lyndon B. Johnson Space Center (JSC) provides the real-time engineering simulation capability needed to support various aspects of the Space Shuttle and the Space Station Programs. The SES has been used as a design and analysis tool throughout the Space Shuttle Program.

Early in the Space Shuttle Program the SES was used to conduct conceptual design studies concerned with Orbiter handling qualities, displays and controls, and orbital operations. As the Shuttle Program advanced, the SES provided a testbed in which flight software requirements (mainly guidance, navigation, and control) could be evaluated. The SES was also used extensively in supporting the design of the Remote Manipulator System (RMS). In 1984 the Manned Maneuvering Unit (MMU) was added to the SES. It has provided on-line support during several Space Shuttle missions, most notably the Solar Maximum repair mission.

More recently, the SES developed the Orbiter/Space Station docking simulation. To develop the capability, reasonably sophisticated mathematical models of the Space Station were installed in the simulation. Mass properties, docking port geometry, RMS grapple fixture geometry, aerodynamics, attitude control system, reaction control system (RCS), and visual models are included in the mathematical models. Additionally, a complex Orbiter-to-Space Station Thruster plume impingement model was developed and installed. The plume impingement model produces reasonably accurate forces and moments on the Space Station that would result from any of the Orbiter's 38 primary RCS thruster exhaust plumes impinging on the Space

Station's surfaces during an Orbiter approach.

These are just some of the many functions that the SES has played a role in, and will continue to serve in, throughout the Space Shuttle and Space Station Programs. Interested readers may find a more detailed description of the SES lab and its functions in [1].

SES Lab Equipment

The SES lab is a large complex consisting of dedicated computers, crew stations, computer-generated imagery visual systems, and graphics systems. Minicomputers provide interfaces to the crew stations, host the graphics systems which generate cockpit displays and real-time displays for test evaluators, and also provide the data recording function for the simulations. The mathematical models are also stored here. A large mainframe computer hosts the Space Shuttle entry and landing simulation and is used in conjunction with the Shuttle forward crew station (or forward cockpit).

The SES crew stations include the aforementioned forward cockpit, the Shuttle aft crew station (aft cockpit), a MMU crew station, and a Space Station crew station (cupola). All stations include flight-like displays provided by electronic scene generators so as to make a simulation session as realistic as possible to the participants. The crew stations are arranged in separate enclosures to facilitate parallel simulations.

Approximately 15 lab equipment pieces - i.e., computers (and the math models), crew stations, scene generators, etc. - are available to the lab users.

Where An Expert System Comes In

In earlier times and with a smaller lab, the SES lab manager generated the weekly schedule manually and fairly easily. However, the lab has grown over the years and so has the level of complexity, causing management to consider automating this task.

Some examples of this complexity: Two parallel simulations may proceed during a scheduled session - one on the "A-Side" and one on the "B-Side" - as long as the equipment that each person has requested is mutually exclusive of the other's hardware needs.

Furthermore, an increased workload in SES activities has recently forced the lab to expand its working hours. Altogether, there are 76 schedulable sessions in a week - ([5 days/week* 3 shifts/day * 2 sessions/shift * 2 parallel simulations/session] + [2 days/week * 2 shifts/day * 2 sessions/shift * 2 simulations/session]).

On the average, between 60-75 session requests are submitted each week. Those who need the Aft Cockpit and/or the MMU for their simulations must run on the A-Side. Others who can accomplish their tasks without these equipment pieces can usually run on the B-Side. On infrequent occasions a requestor will ask for both sides simultaneously.

Another factor considered is the relative priority of each project. Certain recurring events such as astronaut crew training are given a high priority. Priorities of other projects such as conceptual design studies or software development work change weekly according to each project's due date. The lab manager must be fully aware of each project's status so as to make the most effective usage of the lab's resources.

Also, the time slots requested are considered whenever possible. There are those who would rather not work third shifts and/or weekends. An attempt is made to accommodate these requests when feasible. Projects also dictate that work must be completed on/before a given date, thereby making some sessions useless to the requestor.

Taking all these factors into consideration when scheduling is a monumental task for the SES lab manager, particularly when scheduling is only one of the many functions that this individual is responsible for. Human errors can and do appear occasionally. The scheduler can inadvertently assign a lab equipment to two people simultaneously, or some hardware that is unavailable or down for repair might get assigned. Some projects cannot run opposite others. Because of the dynamic nature of the job, last-minute changes can cause a completed schedule to be entirely revamped.

In summary, scheduling relies heavily upon human knowledge and experience. But humans are prone to make mistakes as well as subjective judgments. And because the job is very demanding, human scheduling experts are hard to come by and retain. It is for these reasons that an attempt has been made to automate the scheduling process.

OVERVIEW OF THE SYSTEM

The system was developed to mimic the actual process used in generating a weekly schedule. The weekly requests are first reviewed for completeness and accuracy. Requests containing noticeably incorrect or inconsistent data are corrected or resolved by the lab manager. He also assigns a relative priority to each request based upon his knowledge of the various projects' upcoming due dates or the relative importance of the requested session. A data entry specialist then keys the information from the request into a PC-based Pascal program, using both the mouse and the keyboard interfaces. The graphics/mouse interface is vital to this aspect of the system in that, with over 70 data fields associated with each request, the time spent on the data entry phase has been cut in half (versus using a keyboard interface only).

After the requests have been entered and saved to disk, a second Pascal program is called to update the availability statuses of the various equipment found in the lab. For example, any equipment scheduled for preventative maintenance during a session can be marked as being "unavailable" for that session.

From this second program (and assuming that both of the above tasks have been completed, resulting in a request file and an equipment configuration file), one can then initiate that portion of the expert system that looks for "compatible" pairs of session requests - i.e., those pairs of users who can run simulations in parallel because the equipment requested by each is mutually exclusive of the other person's (and they have both specified a given time slot as being "acceptable").

When two compatible requests are found, they are further constrained by checking the Equipment Configuration File for equipment availability during a given time slot. Should at least one equipment requested be found unavailable, this compatible pair is no longer considered as a candidate for that time slot. This process continues exhaustively until all compatible pairs have been considered for the time slots they deemed desirable.

Those pairs having passed this constraining test are written to a file in CLIPS deffacts format. This will serve as an input file to a CLIPS program (the third and final one in the expert system), which does the actual assigning of compatible pairs to sessions, by priority. If a compatible pair cannot be found for a given session, then that time slot will be assigned to just one person who has the highest remaining priority of those tasks being scheduled. Before completing, this CLIPS program writes a schedule to a disk file, which is then printed out and reviewed by the manager. He has the final decision of whether to use any or all portions of it.

DETAILED DESCRIPTION OF SYSTEM

Start of the Scheduling Process

The first constraint check compares a requestor's list of equipment against the Equipment Configuration File for all schedulable sessions. If a person has requested an equipment that is not available for a given session, that requestor is not considered as a candidate for that session. But assuming that his/her requested equipment are all available, this single user is written to the CLIPS file (in the event that no pair can be found for this slot), and the next constraint check is made - comparing that person's equipment requests against the next person's in the linked list data structure.

User 1's list of requested equipment is compared against User 2's list. The check made is that of a Boolean Exclusive-Or function. That is, if User 1 has requested Equipment X and so has User 2, then these two users are no longer considered compatible. This might be referred to as a "hard" constraint. Now, there also exists a case of a "soft" constraint, and it has to do with a user requesting one or more of the three scene generators (referred to as the ESG2, the POLY, and the CT6). Let us briefly look at this issue before continuing on with the scheduling process.

"Soft" Constraints

There are situations where a user needs a specific scene generator, in effect saying: "I've got to have the (ESG2/POLY/CT6) scene generator, or else I can't do my job." One reason for this is that not all scene generators are capable of generating the desired scene for a simulation session. This again would be considered a hard constraint.

But then there are occasions where any one of the three scene generators is acceptable to the requestor. "I don't care which one you assign to me, just as long as I get one." This would be considered a "soft" constraint. Listed below are the different possibilities that must be considered when verifying a soft constraint between two users.

(Requesting the same generator)

	User 1	User 2
Case 1	NEEDS	NEEDS
Case 2	NEEDS	WANTS
Case 3	WANTS	NEEDS
Case 4	WANTS	WANTS

Case 1 is the "hard" constraint example. If both requestors say they "need" it, then these two are considered incompatible. Cases 2, 3, 4, where "wants" is one of the choices specified, are examples of "soft" constraints and require further investigation.

Consider the following example: User 1 and User 2 match up compatibly on all equipment, excepting the scene generators. Assume all three scene generators are available. User 1 "needs" ESG2 and POLY. User 2 "wants" either the ESG2 or the POLY, but just one of the two is sufficient. In this case, User 1 and User 2 would be incompatible because if User 1 needs them, User 2 would be "locked out."

What if User 1 "needs" ESG2 and POLY, and User 2 "wants" POLY or CT6? Now, they would be considered compatible, because User 1 can be assigned his/her equipment, and User 2 can be assigned the CT6 scene generator.

As long as ONE of the scene generators not "needed" by User i is available and deemed as "wanted" by User j, then Users i and j are compatible, and this soft constraint is resolved. Similarly, for the case where both users "want" a scene generator and at least one of the two has requested TWO or more scene generators, then the soft constraint is resolved (our implicit rule is to assign just ONE scene generator if the requestor specifies "wants" and not "needs").

Cases 2, 3, and 4 above can be expressed in Boolean Algebra terminology. Using the following notation for these Boolean variables:

A_1 = ESG2 Requested by User 1	$\sim A_1$ = ESG2 Not Requested by User 1
A_2 = POLY Requested by User 1	$\sim A_2$ = POLY Not Requested by User 1
A_3 = CT6 Requested by User 1	$\sim A_3$ = CT6 Not Requested by User 1
B_1 = ESG2 Requested by User 2	$\sim B_1$ = ESG2 Not Requested by User 2
B_2 = POLY Requested by User 2	$\sim B_2$ = POLY Not Requested by User 2
B_3 = CT6 Requested by User 2	$\sim B_3$ = CT6 Not Requesaed by User 2

Compatible : Boolean;

Case 2: User 1 "needs" and User 2 "wants". Then -

$$\text{Compatible} := (\sim A_1\&B_1) \text{ OR } (\sim A_2\&B_2) \text{ OR } (\sim A_3\&B_3)$$

or, to generalize:

$$\textbf{Compatible} := \textbf{OR(i, i=1,N)} \ \{ \sim A_i\&B_i \}$$

As long as "Compatible" evaluates to TRUE, User 1 and User 2 are compatible on this soft constraint.

Case 3: User 1 "wants" and User 2 "needs". Then -

$$\text{Compatible} := (A_1\&\sim B_1) \text{ OR } (A_2\&\sim B_2) \text{ OR } (A_3\&\sim B_3)$$

or, to generalize:

$$\textbf{Compatible} := \textbf{OR(i, i=1,N)} \ \{ A_i\&\sim B_i \}$$

Case 4: User 1 "wants" and User 2 "wants". Then -

$$\begin{aligned}\text{Compatible} := {} & (\sim A_1\&B_1) \text{ OR } (\sim A_2\&B_2) \text{ OR } (\sim A_3\&B_3) \text{ OR} \\ & (A_1\&\sim B_1) \text{ OR } (A_2\&\sim B_2) \text{ OR } (A_3\&\sim B_3) \text{ OR} \\ & (A_1\&B_2) \text{ OR } (A_1\&B_3) \text{ OR } (A_2\&B_1) \text{ OR} \\ & (A_2\&B_3) \text{ OR } (A_3\&B_1) \text{ OR } (A_3\&B_{2)}\end{aligned}$$

or, to generalize:

$$\begin{aligned}\textbf{Compatible} := {} & [\ \textbf{OR (i i=1,N)} \ \{ \sim A_i\&B_i \} \ ]\ \textbf{OR} \\ & [\ \textbf{OR (i i=1,N)} \ \{ A_i\&\sim B_i \} \ ]\ \textbf{OR} \\ & [\ \textbf{OR (i,j i=1,N j=1,N i.NE.j)} \ \{ A_i\&B_j \ ]\end{aligned}$$

Back to the Scheduling Process

Assuming that User 1 and User 2 have passed the first two constraint checks, the last constraint check made in this program determines that if either User 1 or 2 has requested an equipment, the Equipment Configuration File is checked to see if the equipment is available for this session. If it is, then User 1 and User 2 (with their associated priorities and the session number) are written as a "compatible-pair" entry to a CLIPS-formatted deffacts file. This file will be the input file to the third and final (CLIPS) program in the expert system.

This entire constraint-reduction process is repeated - that is, User 1 is compared with User 3, User 1 with User 4, and so forth - until all combinations have been exhausted.

Schedule Compatible Pairs for Available Sessions

This third and final program is written in CLIPS, as mentioned earlier. The "deffacts" file created by Program 2 is opened/read. Also, the Request File created by Program 1 is read in; it contains the auxiliary request-related information - such as requestor's name, phone number, activity description, etc. - that is used for listing out the people scheduled for the various sessions.

The program schedules sessions in order from the most desirable (first shift Monday through Friday) to the least desirable (third shift). Two deffacts, shown below, are used here. Deffact "next-session" contains the next session number to be scheduled, where 1 = Session 1 on Monday, 2 = Session 1 on Tuesday, 8 = Session 2 on Monday, etc. Deffact "sessions_left" is a list structure showing those remaining sessions to be scheduled, in the order specified. After a session has been scheduled, the "next-session" fact is modified to contain the left-most number from the "sessions_left" fact. Then, "sessions_left" is also changed to remove a session number from its list once it has been "moved" to "next-session."

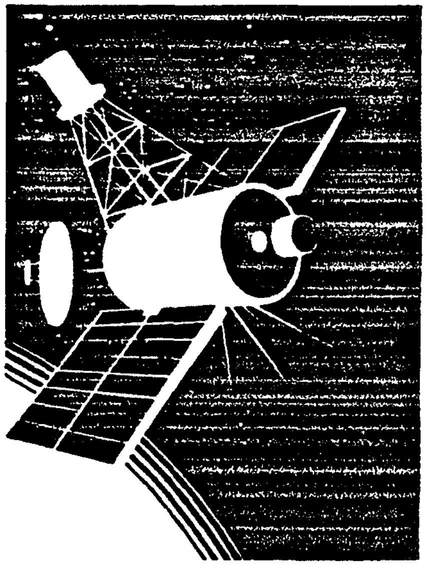
When the final value (0) in "sessions_left" is encountered, the program halts. Note that third shift on weekends (numbers 34, 35, 41, and 42) have been omitted from "sessions_left" because these time slots are currently not used.

```
(next-session 1 Monday)
(sessions_left 2  3  4  5  8  9  10  11  12  15  16  17  18  19
               6  7  13 14 22 23 24  25  26  20  21  27  28  29
              30 31 32 33  36 37 38  39  40   0)
```

The general searching order is to:

- find a compatible pair where both have the current highest priority,
- find a pair where one of the two has the highest priority,
- find just one person (leaving the other slot open for anyone who can use it) having the current highest priority, and
- leave the slot open because no one remaining had specified this session as an acceptable choice.

Also factored into these searching rules is a check to see if either one or both of the current pair being scrutinized were assigned to the last session as well. The reasons behind this are twofold: Those requesting multiple sessions will have a tendency toward wanting to work consistent hours that week (instead of first shift today, third shift tomorrow, etc), and second, this scheme tends to not schedule a multiple session requestor twice on any given day with a gap between sessions (first and third session, for example). A gap would require lab participants to work a non-contiguous eight-hour day.



RESULTS

This system was run for a ten-week trial period. The criteria used for comparison was the number of requests assigned versus the total number requested that week. Shown below are the results.

WEEK	NUMBER ASSIGNED	NUMBER REQUESTED	PERCENTAGE ASSIGNED
Week 1	45	55	81.8
Week 2	52	59	88.1
Week 3	54	59	91.5
Week 4	53	58	91.4
Week 5	47	54	87.0
Week 6	48	54	88.9
Week 7	49	65	75.4
Week 8	59	76	77.6
Week 9	55	66	83.3
Week 10	56	71	78.9

These results are consistent with those obtained by manual scheduling before making forced adjustments. That is, a high percentage of the requests can be satisfied by assigning those with the highest priority to the slots they deemed acceptable. But to fit in the remaining requests, the lab manager must force-assign people to slots they did not specify, or he may assign slots to requestors if they can forgo the use of equipment that is unavailable during that session.

WHAT WAS LEARNED

The approach taken towards the scheduling task had its strong points and its shortcomings. One positive aspect was that the high-priority requests were almost always scheduled, leaving the lower-priority requests to be assigned manually by the lab manager. Another was that a multiple session requestor would often be assigned contiguous sessions as designed. And seldom did a project request get assigned non-contiguous slots within the same day.

A negative point is that a user who requested sessions for two or more DIFFERENT projects that week was often assigned non-contiguous slots within a given day (no check was made to see if the same person was assigned to an earlier session that day). Also, the program found only one schedule. Perhaps better schedules could have been generated to fit in more requests, had some factor of randomness and a looping mechanism been introduced into the program.

Another very influential aspect that became self-evident during the project was the importance of getting requestors to abide by the request submission deadline. Unfortunately, some people at times would not know what their workload for the following week was until the request deadline had passed. Hence, their requests often came in late - typically up until four hours before a completed schedule was to be reviewed by NASA officials. With manual scheduling, one could make certain allowances to accommodate the late entries. However, four hours leaves very little time for the CLIPS program to execute on a minicomputer, particularly with 20 or more interactive users logged in at the time.

SUMMARY

Because of the aforementioned problems, the CLIPS scheduler was eventually replaced by a FORTRAN program on a mainframe to utilize its CPU speed. Most of the problems encountered with the

CLIPS version have been addressed successfully in the new one. The names of users requesting time for different projects are now checked so non-contiguous slots within a day are not assigned to any user. Subject to the above criteria, compatible pairs are randomly selected and assigned to a schedule slot. A completed schedule is then evaluated according to several grading factors, and the 10 schedules with the highest scores are always saved (and later printed at a specified timeout period). The lab manager now has a choice of which schedule to use as a starting base.

One method of circumventing the late submission problem has worked with limited success. "Dummy" requests with the same priority and with the same typical equipment requested by those expected latecomers are entered to serve as place-holders. This allows the scheduler to be started up with more lead time than previously permitted, thus yielding higher-quality schedules.

Because of the constantly changing requirements brought on by new projects, it is felt that it would be difficult, at best, to program in all the constraint checks that are needed. The best that one can expect from the scheduler output is that it is just a starting base that will still require at least some human manipulation to satisfy the constraints associated with that week's requests and to force-fit in any requests that the scheduler cannot handle.

REFERENCES

[1] St. John, R. H., Moorman, G. J., and Brown, B. W., "Real-Time Simulation for Space Stations", PROCEEDINGS OF THE IEEE, Vol. 75, No. 3, March 1987.

□

S38-61

p. 11

N96- 12951

MacDoctor: The Macintosh Diagnoser

David B. Lavery
William D. Brooks

358879

Abstract:

When the Macintosh computer was first released, the primary user was a computer hobbyist who typically had a significant technical background and was highly motivated to understand the internal structure and operational intricacies of the computer. In recent years the Macintosh computer has become a widely-accepted general purpose computer which is being used by an ever-increasing non-technical audience. This has lead to a large base of users which have neither the interest nor the background to understand what is happening "behind the scenes" when the Macintosh is put to use - or what should be happening when something goes wrong.

Additionally, the Macintosh itself has evolved from a simple closed design to a complete family of processor platforms and peripherals with a tremendous number of possible configurations. With the increasing popularity of the Macintosh series, software and hardware developers are producing a product for every user's need. As the complexity of configuration possibilities grows, the need for experienced or even expert knowledge is required to diagnose problems. This presents a problem to uneducated or casual users. This problem indicates a new Macintosh consumer need; that is, a diagnostic tool able to determine the problem for the user. As the volume of Macintosh products has increased, this need has also increased.

The NASA Headquarters Office of Aeronautics and Space Technology (OAST) has become intimately aware of these problems and needs as they installed a Macintosh II computer on the desk of every employee (approximately 180 machines). Early in the installation process, the user support staff received calls to assist with a large number of problems common to multiple users. A desire was expressed for some type of aid to help a user recognize and diagnose the most common of the problems, allowing the user support staff to concentrate their talents on the more uncommon (and typically more difficult) problems. Additionally, such an aid could be used as a training assistant for new or novice user support personnel.

With this idea in mind, the authors began a project to identify and implement the knowledge base required to recognize, diagnose, and provide suggested solutions for, the most common problems associated with typical Macintosh use. This paper will present the process used to develop this implementation, from the initial analysis of user support call logs to identify the problem domain, through the use of CLIPS as the inference engine kernel, to the completion and testing of the system prototype.

MacDoctor: The Macintosh Diagnoser

Executive Summary

MacDoctor is the product of a graduate school project to develop a forward chaining, rule-based diagnostic tool to determine the cause, and thus the remedy, if any, of a Macintosh hardware configuration problem. The problem is identified through the traversal of a discrimination network represented in CLIPS rules. Remedies are directly, if not uniquely, addressed by a given problem determination. Future areas of research include automatic network exploration and mapping, predictive diagnosis, domain expansion and user maintenance.

Introduction

When the Macintosh computer was first released, the primary user was a computer hobbyist who typically had a significant technical background and was highly motivated to understand the internal structure and operational intricacies of the computer. In recent years the Macintosh computer has become a widely-accepted general purpose computer which is being used by an ever-increasing non-technical audience. This has lead to a large base of users which have neither the interest nor the background to understand what is happening "behind the scenes" when the Macintosh is put to use - or what should be happening when something goes wrong.

Additionally, the Macintosh itself has evolved from a simple closed design to a complete family of processor platforms and peripherals with a tremendous number of possible configurations. With the increasing popularity of the Macintosh series, software and hardware developers are producing a product for every user's need. As the complexity of configuration possibilities grows, the need for experienced or even expert knowledge is required to diagnose problems. This presents a problem to uneducated or casual users. This problem indicates a new Macintosh consumer need; that is, a diagnostic tool able to determine the problem for the user. As the volume of Macintosh products has increased, this need has also increased.

The NASA Headquarters Office of Aeronautics, Exploration and Technology (OAET) has become intimately aware of these problems and needs as they installed a Macintosh II computer on the desk of every employee (approximately 180 machines). Early in the installation process, the user support staff received calls to assist with a large number of problems common to multiple users. A desire was expressed for some type of aid to help a user recognize and diagnose the most common of the problems, allowing the user support staff to concentrate their talents on the more uncommon (and typically more difficult) problems. Additionally, such an aid could be used as a training assistant for new or novice user support personnel.

With this idea in mind, the authors have initiated a graduate research project to identify and implement the knowledge base required to recognize, diagnose, and provide suggested solutions for, the most common problems associated with typical Macintosh use. This paper will present the process used to develop this implementation, from the initial analysis of user support call logs to identify the problem domain, through the use of CLIPS as the inference engine kernel, to the completion and testing of the system prototype.

Problem Statement

The objective of this project is to produce an easy-to-use, plain talking diagnostic tool which will be capable of analyzing a user's description of a problem, recognizing the problem condition and suggesting a solution activity. It is noted that Apple and other vendors manufacture products with built-in test and evaluation (BITE) capabilities. However, these are typically designed for board or component-level investigation. The authors intend to address a higher level implementation - a configuration diagnostic rather than a component diagnostic.

The problem is also more complicated than the component BITE testing. Single components are largely fixed in design. Test procedures for such components can be predetermined. At a configuration level, test procedure designs have added complexity in that computer configurations vary greatly depending on the system options and peripherals that the user has chosen for the system.

If an automated tool were made available to help users track down their configuration problems, at least two categories of users of the tool can be identified. The first is the new, non-computer-literate users who will use the tool to identify and correct problem conditions on their local Macintosh systems, and through the use of the tool gain greater degree of computer literacy. The second class of user includes personnel assigned to assist in the diagnosis and correction of problems for a large configuration of Macintosh systems ("help desk" or "user consultant" staffers), who need to quickly become effective and productive in the remote diagnosis of system problems, who would use the tool as both a rapid training aid and a productivity enhancement utility.

Implementation Approach

Early in the definition process for **MacDoctor**, it was realized that a forward chaining diagnosis system would present certain implementation capabilities which would be valuable to the development of the application. Inherent in the design of such systems in the ability to collect an initial set of error conditions from the user, and synthesize a set of possible solutions. As additional information is gathered, invalid solutions are removed, until a final solution set remains. This set can be indexed with confidence factors to indicate the expected precision of the proposed solution. These systems are flexible, both in terms of implementation and operation - as the knowledge base is developed there are few restrictions on the ordering of the knowledge rules, and as the expert system is used, multiple logic paths may be followed by the user to reach the same solution. The logic structure used in the design of the questions to the user can resemble an inverted tree, and yet the user can provide incomplete or inferred information which allows them to move between then logical branches of the tree and traverse the tree without being constrained by the formalism of the tree structure.

The forward chaining expert system was selected as the best solution for developing the Macintosh diagnoser. Based on that decision, the following implementation decisions were made:

- The CLIPS expert system shell was used to create and develop the knowledge base and antecedent-consequent rule definitions. CLIPS is an extensible expert system shell developed by the NASA Johnson Space Center (JSC), with executable versions for Cray, Cyber, CDC, IBM, PC, VAX and Apollo computers, as well as the target Macintosh platform.

- Problem domain information was obtained from the NASA Headquarters User Support Center (USC) service call logs. The USC provides assistance to approximately 180 Macintosh users at NASA Headquarters, by aiding with problem diagnosis, system repair, training, and general user support. During the past two years of operation, the USC has compiled extensive documentation by logging problem calls and documenting the eventual solutions provided to users. The USC made this documentation available, and a set of typical user problems and questions which have been used has been derived as the initial Macintosh diagnoser problem domain.

- The expert knowledge for solution of the problems comes from two sources. First, the system implementers have over two years of experience with diagnosing Macintosh system and configuration problems, gained through a combination of professional experience and participation with Macintosh users groups (which involves training of new users). This learned knowledge is used extensively to develop the knowledge base. Second, for areas where the developers knowledge may be insufficient, the Systems Engineering Group at the Apple Federal Government Operations office in Reston, Virginia, was contacted and agreed to provide documentation and support similar to that normally supplied to the Apple field engineers.

- Development of system components external to the expert system shell (user interface, internal system status queries, etc.) were developed in the C programming language. The CLIPS expert system shell was developed in C, and readily incorporates external C routines.

Problem Domain Definition

The **MacDoctor** domain of expertise was selected based on the availability of raw data and the familiarity of the developers. The domain selected was the interoffice computer network installed in OAET, which consists of over 180

Macintosh II desktop computers connected via Ethernet. NASA has established a computing facilities support staff (help desk) which is responsible for the handling of hardware and software problems encountered by NASA personnel. Typically the users are not extensively trained in computer technology and thus constitute a population of novice users.

To define the problem domain to be addressed by the **MacDoctor** application, copies of the User Support Center calls logs were obtained, and review of the logs was initiated. 1372 call log entries were reviewed, and the following problem breakdown was derived:

Printing problems - networked LaserWriters	192
Printing problems - direct connect LaserWriters	0
Printing problems - networked ImageWriters	3
Printing problems - direct connect ImageWriters	7
Disk problems - SCSI devices	44
Disk problems - Diskette drives	21
Net/comm problems - mail services	66
Net/comm problems - file servers	18
Net/comm problems - modem services	171
System problems	60
Application problems	57
Finder problems	35
I/O problems	61
Total:	735

Note that the problem breakdown displayed above is a summary of the domain definition that we have created. The granularity of detail worked with is considerably greater. For example, the "printing problems- networked LaserWriters" line item above actually contains 28 distinct elements, each of which represents a unique problem state to be recognized by **MacDoctor**. In total, 180 distinct problems which occur within the domain were identified.

637 calls from the log entries were rejected, as they were determined to be outside the domain of the defined problem. These include items such as: requests for software, requests for specific training, problems pertaining to non-Macintosh systems, etc.

Determining the problem space was the first step. The more significant task was to build the discrimination network which would select the correct problem identification from the problem space. Again the help desks supplied much of the information. Each entry in the help desk log included the staff member's name, the problem as reported to the help desk, the procedure undertaken to identify the problem, the problem as determined by the staff member, and the steps taken to remedy the problem. Examination of the collection of the help desk log entries for each distinct problem showed a similar pattern of diagnosis and remedy. For each problem, the diagnosis and remedy were reviewed by domain experts to insure their validity. This process resulted in classes of problems with each problem represented by a description of the problem, a unique set of symptoms which the problem will exhibit, and the remedy to the problem. By matching the symptom set, the problem can be identified and the remedy proscribed.

The symptom sets for the various problems were found to intersect to a high degree. A particular symptom could often be exhibited by several different problems. The problems were thus combined into a discriminate network or tree. The root node of the structure represents the most discriminating symptom, that symptom which reduces the problem space the most. For any node to be higher in the tree, this property must be maintained. If this is maintained, traversal of the tree will rapidly converge on the correct diagnosis.

Scope of Solution

It would be impractical to attempt to implement **MacDoctor** with the ability to recognize every problem identified in the problem domain. Instead, it was the developer's intent to sort the problems identified in the domain by frequency of occurrence and then provide an implementation which will address the top 80% of this list. The remaining 20% of the problem space includes items which tend to be either specific to a unique system configuration, or problems which occur with very low frequency.

Field testing of the **MacDoctor** application was arranged with the NASA User Support Center (source of original domain information) once the application knowledge base was established and implemented. The User Support Center agreed to utilize the system as a training aid for new members of the USC staff to increase productivity while the staff members are becoming familiar with the Macintosh installation, and to distribute the application to selected end users for evaluation and knowledge base validation. This field testing is still underway, and feedback from the testing is being used to implement a second iteration of **MacDoctor**.

Application Design

The design of **MacDoctor** separates the overall system into the following parts: user interface, inference engine, expert knowledge representation, and maintenance front-end.

As each of the segments was implemented, the developers were confronted with the issue of how the contents of knowledge base would be divided between the interface driver and the inference engine. These are the options considered:

- Have all the possible queries which may be asked of the user predefined in the interface portion of the application, installed in dedicated dialog boxes. The results of each query are interpreted by the interface portion of the application and either passed to the inference engine for incorporation within rules and further processing, or the interface portion may act directly upon the results and process additional queries. The advantage of this approach is that the number of communications between the portions of the applications are minimized, and all the queries are precompiled, which will result in minimal execution times. The disadvantage is that any future extensions of the application will require considerable source-level reprogramming and recompiling of the application, and overall modularity of the application is minimized. Additionally, any change in the logic used in the knowledge base will require modification of both the interface and the inference portions of the application.

- Have all the possible queries which may be asked of the user predefined in the interface portions of the application, installed in dedicated dialog boxes. The results of each query are passed back to the inference engine for incorporation within rules and further processing. The advantage of this is faster processing of queries by minimizing the communication required for the inference portion to request a query, resulting in improved execution times. The disadvantage is that future extensions to the application will require source-level reprogramming and recompiling of the application.

- Have all the queries defined within the inference portion of the application, and queries are passed forward to the interface portion as they are needed. The interface portion is basically a small set of dialog box "shells", which accept and display the query strings from the inference portion, and return the query results. The advantage of this is that full modularity of the application is maintained, and that extensions to the knowledge base and modifications of the rule logic will not require recompilation of the application (it should be noted that input to the inference portion of the application will be done via a single text file containing the rule definitions for the knowledge base; therefore, modifications to the rule base will require only the use of a text editor, and not a compiler or development environment). This will significantly ease maintainability of the application. The disadvantage is that query requests from the inference potion of the application to the interface portion will require more communication between the portions, resulting in slightly decreased application performance.

The interface implementation method selected was to develop a general-case query interface driver which will allow the inference engine to pose query text to the inference driver for display. This will allow all of the logic, rule definitions, query text, and suggested solutions to be located in one modular file (permitting easier maintenance and extension), and allow the user interface to automatically handle extensions to the knowledge base without requiring recompilation of the application. This is done at a slight cost of system performance, but the impact to the user is negligible.

Knowledge Representation

Experience so far indicates that through the use of CLIPS we are able to adequately represent the knowledge base required to address the known problems, and a small subset of the knowledge base has been implemented to verify this. Initial efforts concentrated on the implementation of the rules required to recognize and suggest solutions to file server access problems. This problem class was selected as it included most of the major elements common to the problem space (i.e. network connectivity, supplied power, access control, network definition, device selection, etc.). The definitions required to represent the knowledge for this section of the application was stated with 39 rules in about 420 lines of code. As yet undetermined is the best way to encapsulate the knowledge data separately from the knowledge base framework, to allow extension of the rule set without full knowledge of the CLIPS syntax and structure (to allow maintenance of the knowledge base).

By combining the query format with properly structured rules in the knowledge base, the search paths used to move from the initial state to the complete problem space have been structured to emulate a recursive binary tree, where each node is either a query to the user or a fact inferred by the inference engine, each branch is based on the response to the query, and each terminal leaf is a problem state. For example, a node may consist of "is the printer is plugged in?". The set of possible answers determines the number of exits from the node; with this example they might be "yes" or "no". This is analogous to collecting a set of facts, "the printer is plugged in" or "the printer is not plugged in." The answering of the question corresponds to the consequent of a rule. Determining whether or not to fire a rule and test the premise corresponds to testing for the presence of the effects of a parent node's corresponding consequent. Continuing with the example, the parent node is "is the printer turned on?" with possible answers "yes" and "no". The current node, "is the printer plugged in?" is a child connected to the "yes" exit from the parent. In order to visit the child, the parent node must have been visited and exited via the "yes" arc. Mapping this to the rule representation, in order to fire the second rule (child node) the first rule must have asserted facts which allow the premise of the second rule to fire. So, in the example, the premise of the second rule would be "if (printer turned off)".
So translation maps answering the node's question (choosing an exit arc) to a rule consequence and the parent's exit arc to a rule premise.

Currently, all queries to the user concerning states of the configuration require responses which can be answered if the user makes some direct observation from the workstation (i.e. "is your network interface turned 'ON' or 'OFF'?"). Some problem conditions exist which cannot be uniquely isolated by direct observation responses. For example, if too many users are logged on to a file server to allow an additional user to log on, the user may not be able to tell if he is not being allowed access due to server overbooking or an invalid user account. Without some external information from the server administrator, the user does not have a mechanism to identify which of these problem conditions is true while sitting at the workstation. Under these conditions, the current system halts and displays a list of all the possible problem conditions which fit the known information and suggests sources for the external information which can further isolate the exact problem. Future expansion of the system could provide an option to wait for the user to retrieve the information and the proceed.

With regard to the formalization of the rule schemas, the rules have been classified into these categories: phase control, queries, configuration inference, and solution suggestion. These categories are defined as follows:

Phase control rules:

```
IF current-phase-completed
     THEN assert-begin-next-phase
```

These rules act as flow control "traffic cops" during the execution of the inference engine. The real purpose of including phase control within **MacDoctor** is to force all queries to the user to take place before any suggested solutions are displayed. This is an issue in those cases where the system is diagnosing multiple problems and identifies a solution to one problem before posing all the queries required to isolate the remaining problems.

Query rules:

```
IF query-phase AND device-state-needed
     THEN request-state-from-user
     AND assert-device-state
```

These rules are fired during the query phase to pose questions to the user when information about the state of a device or configuration component is needed. The queries are specifically designed to constrain the user to a "yes/no" or "on/off" response. The "request-state-from-user" attribute is used to define the query string that is displayed to the user and to receive the user response. The "assert-device-state" attribute is used to assert a fact into the fact list which defines the state of the device, based on the response from the user. This fact, when added to the fact list, typically fires either another query rule, a configuration rule, or defines a terminal problem condition.

Configuration inference rules:

```
IF device-state-known
     THEN assert-derived-facts
```

These rules are fired by facts asserted by the query rules, and are used to define facts inferred from known device states. For example, if a user provides a response which determines that a file server is visible, a configuration inference rule would fire which would infer that the network interface is on, the network is active, and the server is up.

Solution suggestion rules:

```
IF problem-condition-known
```

```
        THEN assert-problem-solution

    IF problem-solutiön-known
        THEN display-problem-solution
```

These rules are fired by facts asserted from either query rules or configuration inference rules, and are intended to define solutions to isolated problems and then display the solutions to the user. The "problem-condition-known" attribute is either a problem definition or device state which defines a problem. "Assert-problem-solution" defines the solution text and then "display-problem-solution" displays the text to the user.

Implementation

The implementation of **MacDoctor** was written in Think C 3.2 on a Macintosh II. Macintosh was chosen for its user interface and Think C for its software development environment. The software was based on the general intention to embed the CLIPS rule engine within a C application. CLIPS-to-application communication was accomplished through the creation of a user function which interacted with the user through Macintosh user interface. The user function was defined in the CLIPS environment as a parameter returning function. The function was passed the node's question ("Is the printer plugged in?") and returned the user's response to the question ("yes", "no" etc). Within the CLIPS language, the function call was embedded within an assertion. The assertion statement were of the form:

'(assert (printer-state = (user-dialog "Is printer plugged in?" "yes" "no")))'

The **user-dialog** function was written in C and designed to present to first parameter in a user dialog window with the remaining parameters as answer buttons. The answer buttons are mouse selectable fields on the window. The user-dialog function creates a CLIPS symbol representing the user's selection, such as "yes" or "no".[1] This symbol is returned to the CLIPS environment and is used in the assertion.

Future Development Directions

At this point, the future plans for the development of **MacDoctor** include completion of the knowledge base to allow the application to recognize the aforementioned 80% of the problem domain, and to fully implement the Macintosh interface to the knowledge base and inference engine. Following that several directions are being considered, including:

- Implementation of a "machine learning" capability, whereby **MacDoctor** will be able to record and analyze patterns of user responses which lead to "dead ends" in the knowledge base (i.e. the user describes a problem

[1] Note that redundant attempts to create a CLIPS symbol simply returns the pre-existing symbol.

which **MacDoctor** does not recognize). The application could be given the ability to analyze the response patterns and alert the knowledge base maintainers of the occurrence of an unrecognized problem class. The maintainers can then use this information to extend the knowledge base of the application.

- Augmentation of the information-gathering capabilities of the application which would allow **MacDoctor** to determine several system configuration statistics and conditions instead of requesting all status information from the user. For example, enable the application with the capability to query the Chooser directly to determine the currently selected printer, rather than posing a query to the user requesting the name of the printer.

- Add a solution feedback mechanism which would allow the system to track the solution suggestions presented to the user and verify that the solution corrected the described problem. In those cases where the solution and the actual problem do not match, enable the system with an analysis capability which could determine if an alternative solution in the knowledge base would provide a "more correct" answer, or if an extension to the knowledge base is needed to handle the actual problem.

- Augment the user interface for the solution suggestions to expand the text description of the solution to display drawings and/or animation to better describe the corrective action required by the user. For example, if the suggested solution is to have the user check that the LocalTalk cable is connected to the printer port on the Mac, include an option which would display a short animation sequence illustrating the back of the Macintosh with a LocalTalk cable being connected.

The Authors

Dave Lavery is the Deputy Manager of the Artificial Intelligence and Robotics Research Program for the National Aeronautics and Space Administration (NASA). He is currently a part-time graduate student pursuing a Masters Degree in Computer Science at George Mason University. Contact: 202-453-2720, DLAVERY@NASAMAIL.AMES.NASA.GOV

Bill Brooks is a project manager with Advanced Decision Systems in Rosslyn, Virginia. He is currently a part time graduate student at George Mason University, working on a Masters Degree in Systems Engineering. Contact: 703-243-1611, WBROOKS@POTOMAC.ADS.COM

Development of an Instructional Expert System for Hole Drilling Processes

Souhaila Al-Mutawa, Vijay Srinivas, and Young Bai Moon

Mechanical and Aerospace Engineering
Syracuse University
Syracuse, NY 13244

ABSTRACT

An expert system which captures the expertise of workshop technicians in the drilling domain was developed. The expert system is aimed at novice technicians who know how to operate the machines but have not acquired the decision-making skills that are gained with experience. This paper describes the domain background and the stages of development of the expert system.

1 INTRODUCTION

Human expertise is essential for process planning in the manufacturing environment. In a workshop, process planning is concerned with determining the sequence of individual machining operations needed to produce a given part. The decision process is guided by a multitude of variables which include the process requirements and equipment capability. The process plan involves a set of machining operations. Each of these operations demands skill and knowledge derived from experience on the part of the technician. The goal of this project is to capture the expertise of the technicians in an expert system. The domain of this project will be restricted to the hole drilling operations performed in a workshop on manually controlled machines.

Several expert systems have been developed for generative process planning [1]. GARI was developed in 1981, its domain is restricted to the metal cutting industry. In 1984, EXCAP was developed to generate process plans for machining of rotational components, and CUTTECH was developed to select cutting tools, speeds and feeds.

Our expert system is aimed at the novice, or apprentice, in the workshop who has been formally taught to operate the machines but has no experience. A novice will usually be trained by observing the experienced technicians propose a process plan, and then execute each machining operation in the plan. When a novice asks the technicians to justify a certain plan of action, they will usually attribute their decisions to "experience." In order for the novice to learn from their experience he needs to follow the reasoning process involved in such decisions. With the aid of an expert system a novice will be able to follow the decision-making process. Eventually the novice should acquire the experience required for the job, and he will be able to expand the expert system by adding his own judgments.

2 DESCRIPTION OF DOMAIN

There is hardly a product that does not contain one or more holes. Holes are produced in a variety of ways; for example, they may be drilled, punched, or sawed. Drilling accounts for more than 80% of the metal-cutting operations in a workshop [2]. Drilling is generally not a precision operation. In order to produce holes within a specified tolerance and with a good surface finish, the drilling operation is followed by precision sizing operations. The most common one being reaming.

In this section the process of drilling a hole will be discussed. This process begins with the engineer designing a part to be manufactured by the technician. The technician will receive a blue-print of the part, and then it is up to him to generate the process plan. The process plan is the sequence of individual machining operations needed to produce a given part, keeping within the specifications on the blue-print and any special instructions it may contain.

2.1 The Blue-Print

In order to produce a process plan, the technician is supplied with a blue-print of the part to be machined. The blue-print is an engineering drawing of the part. It provides two or three views (front, top, side) of what the final product should look like. The material and dimensions of the part are specified. The hardness of the material may be specified on the blue-print. It is usually given as a Brinell Hardness Number (BHN).

For parts with holes, the position of the hole, on the part, and its diameter are given. If the hole needs to be machined within a certain tolerance then its value is also given. The tolerance value is specified as an upper and a lower allowable limit for the hole diameter. For example, a hole with a diameter D and a tolerance of +/- t can have a diameter size anywhere between (D+t) and (D-t).

2.2 The Drilling Process

The position of the hole must first be located, in accordance with the specifications on the blue-print. Once the position of the hole has been marked then the drilling process can begin. The machine and drill tool to use for machining a particular hole are selected. The choice is based on factors such as the depth of the hole, the accessibility of the hole, and the material hardness.

The drill tool is selected by specifying its type, diameter, tool material, the shape of the shank and the flutes. The shank of a drill tool is the part by which it is held and driven, it may be straight or tapered. The flutes are the helical grooves on the drill body which permit the flow of coolant and the removal of chips. These are illustrated in figure 1.

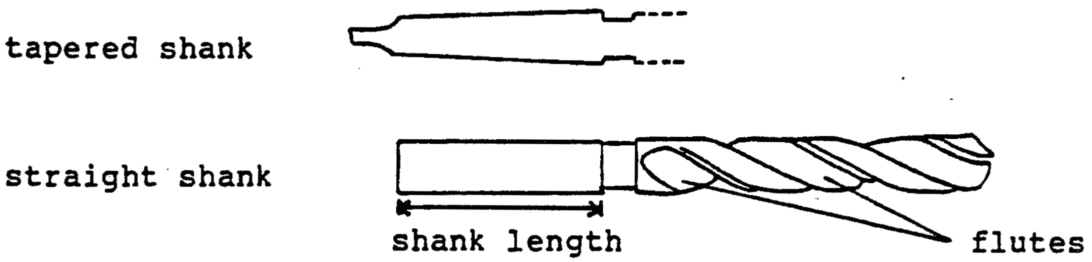


Figure 1 Shank and Flutes on a Drill Tool

2.3 The Reaming Process

When the size of the drilled hole must be kept within a tolerance of at least +/- 0.005 inch or a good surface finish is needed then the hole needs to be reamed. After drilling, the hole diameter is measured and then an appropriately sized reamer is selected to remove whatever material is left to bring the hole size within the specified tolerance.

The reamer is selected by specifying its type, material, and diameter. These depend on the hole diameter, amount of material left by the drill for reaming, the number of holes to be reamed, and the required surface finish.

2.4 The Machines

The three manual machines which can perform the drilling and reaming operations in the workshop are the lathe, the drill-press, and the milling machine. The part to be manufactured is referred to as the workpiece.

On the lathe, the cutting tool (i.e. drill tool or reamer) is held in the tailstock and the workpiece is held in the chuck. The tailstock is advanced manually into the rotating workpiece. The speed of the drilling operation is the speed of rotation of the workpiece, specified as the number of revolutions per minute (rpm). The feed is the number of inches moved by the drill-tool into the workpiece per revolution of the workpiece (ipr).

On the drill-press, the workpiece is placed on the stationary horizontal table and the cutting tool is moved towards it manually. The speed is specified as that of the cutting tool rotation measured in revolutions per minute (rpm).

In the process of drilling a hole using the milling machine, the workpiece is placed stationary on the horizontal table and the cutting tool approaches it. The speed of the tool is measured in revolutions per minute (rpm).

3 FEASIBILITY STUDY

This expert system is aimed at the apprentice. An apprentice is someone who has been formally trained to use the machines but has no experience. He is usually asked to follow the instructions given to him by a more experienced technician. If the output does not match his expectations then he may have difficulties in producing an alternate plan.

The system developed is an instructional system, which contains an explanation facility. When confronted with the task of drilling a hole, the apprentice can consult the expert system and can expect to receive advice on the decisions that need to be made in order to carry out the task. At any stage of the questioning, the apprentice can ask the system to clarify the question.

The experience of the technicians is accumulated in the form of rules of thumb. In the domain of this expert system, there are tables which match the diameter of the hole with the required speeds and feeds for a particular material. Most of these tables do not take into account the practical aspects of the problem, such as the production rate. However, the technicians will tend to rely on their experience when setting these variables by balancing the number of pieces that need to be produced and the time allocated for the production. Also the technicians tend to think more in terms of a range of speeds, rather than absolute values as given on some tables, and in terms of the production rate required.

Since the nature of the knowledge is in the form of rules of thumb and their combinations, then this domain is well-suited to be implemented with a rule-based expert system shell. The goals of the expert system are the selections of machines, drill-tools, reamers, speeds, and coolants. These are all of the specifications that a technician needs to determine before starting to drill. The expert system will ask for information which is given on the blueprint of the part to be drilled.

4 KNOWLEDGE ACQUISITION

The experts in the workshop are the technicians. They are usually asked to make an object from its description on the blue-print. Hence it is up to their ingenuity to decide on the most feasible machine to use for drilling and all of the other decisions that are involved in the operation. There are many variables which control this decision-making process. The experience of the technician is gained by the amount of variety in the jobs encountered, and not necessarily in the number of years spent working in a workshop.

The experts consulted for this domain will be referred to as A, B, and C. Expert A has 12 years of experience and he is a tool and die-maker which is the highest training for a technician. Expert B has 10 years of experience, and expert C has 20 years of experience.

The knowledge acquisition phase of the project was the most time-consuming. This phase was divided into three stages:

1. Initial consultation - the experts were consulted to determine the feasibility of the proposed problem.

2. Knowledge solicitation - the experts were consulted when building the knowledge base.

3. Feedback during implementation - the experts were consulted when an inconsistency appeared or when more clarification was needed during the implementation.

4.1 Initial Consultation

The original intention of the project was to produce a process plan for any part which could be manufactured on the manually operated machines in the workshop. The process plan was to list the sequence of operations, the tools, machines, and their settings in order to manufacture the part.

Expert A was the first to be consulted. He explained the overall decision-making that one would undergo when confronted with a blue-print and asked to manufacture the part. He emphasized that the sheer amount of variables that need to be taken into account in order to produce a complete process plan of a simple job was too many to be handled simultaneously. So at his suggestions, the problem was confined to one operation in the process plan. The drilling operation was chosen because most manufacturing products have at least one drilled hole, thus making it the most common operation in the workshop.

Even though the number of variables have been reduced considerably, there are aspects of the drilling domain which have been eliminated in order to produce the expert system during the allotted time. These aspects have been singled out by expert B. After consulting expert B over a period of four days, the decision to exclude the methods for positioning a hole on a workpiece and the drilling of threaded holes was made.

This initial consultation with the experts was essential in formally defining the domain of the expert system. Due to their expertise in the field, the domain was confined to a functional subset of a larger problem.

4.2 Knowledge Solicitation

The drilling and reaming operations are well documented in textbooks and handbooks relevant to the workshop operations. So the basic goals of the expert system were initially defined based on the literature [2,3]. All of the experts used these two books as their major sources of information.

Experts A and B were interviewed independently. During B's interviews, a series of open-ended questions were posed because the project was at the design stage and the problem domain was being refined. An example of a question posed to expert B is: "Under what circumstances would you choose the milling machine for drilling or reaming, and why?" Expert B was interviewed for four days, and each interview lasted approximately 2 hours.

Expert A was interviewed one week after expert B's interviews. By then the questions became more specific as the problem was better defined. An example of the questions that expert A was asked is: "If the Brinell hardness number was not specified on the blueprint how would you classify the material hardness, and when would you need to use this classification?"

Expert C was not consulted during the knowledge acquisition phase. The main reason being that he was not available during that time, and the interview format did not suit him. His collaboration was essential in the validation phase of the development of the expert system.

4.3 Feedback During Implementation

During the implementation of the expert system, expert A was consulted several times to clarify some of the points made during the interviews and to verify the rules extracted from the literature. Most rules which were extracted from the literature were revised to reflect what the experienced technician would use and do. For example, in [2] several types of reamers are suggested, whereas according to expert A the most commonly used reamer in the workshop is the chucking reamer because it is available in all sizes.

The knowledge acquisition continued into the validation phase, when the experts were presented with the output of the system for hypothetical problems. If the results from the expert system were not acceptable by the experts and a justification was given, then they were altered.

5 CONCEPTUAL DESIGN

The conceptual design phase established the necessary and optional inputs to the expert system. The minimum specifications required before drilling were also established. The relationships between the variables and the constraints imposed upon them were determined from consultations with the experts.

The level at which the knowledge is described is based on the level that the experts use to reason. The basic components of knowledge are naming, describing, organizing, relating, and constraining [4]. These components will be described as related to the project domain.

The naming process consisted of assigning names to the parameters involved in the domain. It was observed that even though both the drilling and reaming operations made use of the same input knowledge from the user, the experts tend to think of them as two separate processes. So all of the parameters related to the drilling process were superseded with DRILL, and all of the ones related to the reaming operation were superseded with REAMER. For example, DRILL-TYPE and REAMER-MATERIAL.

In order to describe the important properties of a parameter it is necessary to decide what the system has to know about them in order to be able to carry out its reasoning tasks. This is best illustrated by an example: the experts choose to apply the reaming operation when the hole needs to be made with precision and a high quality surface finish is required. However, there are instances when this information is not specified explicitly on the blue-print, but the technician may know that a good surface finish is needed for the specific part he wants to manufacture. So when deciding whether to ream or not the expert system needs to know all of the cases when reaming is necessary even if it is not stated in the blue-print.

The information that the experts gain from knowing the material is basically knowing whether they will be required to drill into a relatively hard or soft material. So the Brinell hardness number is used as an indicator of classifying the material as either hard or soft. For commonly used materials, the technicians know from experience which of them are hard and which are soft.

Constraints control the properties of the parameters. Values such as the size of the hole and the Brinell hardness number were given a range. Thus the diameter of the hole needs to be a positive number which will not exceed 4 inches, since this is the maximum size considered in the domain. The Brinell hardness number was constrained to be input as a positive number, because it cannot be negative. So if the user ignores these constraints, the system will reject his answers, thus preserving the integrity of the expert system.

6 IMPLEMENTATION

The expert system was implemented using CLIPS. Forward chaining was used. The expert system requires some essential facts about the drilling problem before it can make any decisions. All of the input facts are derived from the blue-print and the required production rate. They are listed as follows:

1. The material of the part to be machined.

2. The size of the hole to be drilled.

3. The type of hole.

4. The time limit imposed on the operation, if any.

5. The number of pieces that require drilling.

Additional information such as the material hardness and the tolerance may or may not be available from the blue-print. Nevertheless, they are inferred by asking additional questions to the user.

The output parameters from the expert system have been chosen after consulting with the experts. They are determined by the information necessary before a drilling or reaming operation can be undertaken.

The expert system will produce recommendations which involve the specifications for choosing the cutting tools (type, material, and size), in addition to choosing the machines and their starting speed. When more than one machine is chosen, then the choice between them is not critical for that particular problem. The system will also make recommendations on whether the hole needs to be reamed and if a coolant is required.

7 TESTING AND VALIDATION

The expert system was evaluated for program accuracy and utility. The rules were checked for conflicts and redundancies. Rules were in conflict if for the same condition statement, two or more rules asserted conflicting facts. The conflicts were resolved by reviewing the accuracy of the knowledge. Rules were redundant when other rules assert the same facts by inferring with the same knowledge. The redundant rules were eliminated from the rule-base by either removing them or combining them together.

The utility of the results was confirmed by the experts during the knowledge acquisition phase. The specifications for the drill tool and the reamer in the conclusion of the consultation were unambiguous and the correct tool can easily be identified. Care was taken in mentioning that the recommended speeds were starting speeds, because as the hole is being drilled the technician may alter the speed depending on how rigidly the part was held.

The overall validity of the expert system was tested by posing several hypothetical problems. The technician, C, was consulted with the problems, and his recommendations for the choice of tools, machines, and speeds were recorded. Another expert, B, was shown the conclusions that the expert system produced and was asked if he would consider these as reasonable recommendations. The experts' comments are given with two problems below. The expert system's recommendations are given in Appendix A.

7.1 Problem 1

The top and side views of the part is shown in figure 2.

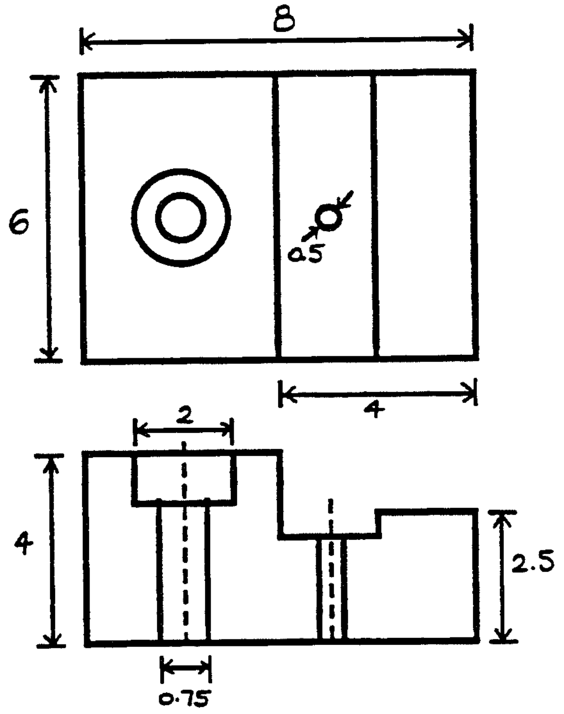


Figure 2 Part with Two Counter-Bore Holes

The specifications for the part are as follows:

material: mild steel
number of pieces to manufacture: 50
good surface finish required

The experts thought that the conclusions from the system were reasonable. The drill-tool recommended was for the smaller hole diameter, and the expert system suggests using a piloted-boring tool for the larger diameter. The experts expected the system to specify the piloted-boring tool specifications as it did for the drill-tool. This was not specified because the boring operation was not within the scope of the domain, and only a qualitative recommendation was given.

7.2 Problem 2

The top and side views of the part are shown in figure 3.

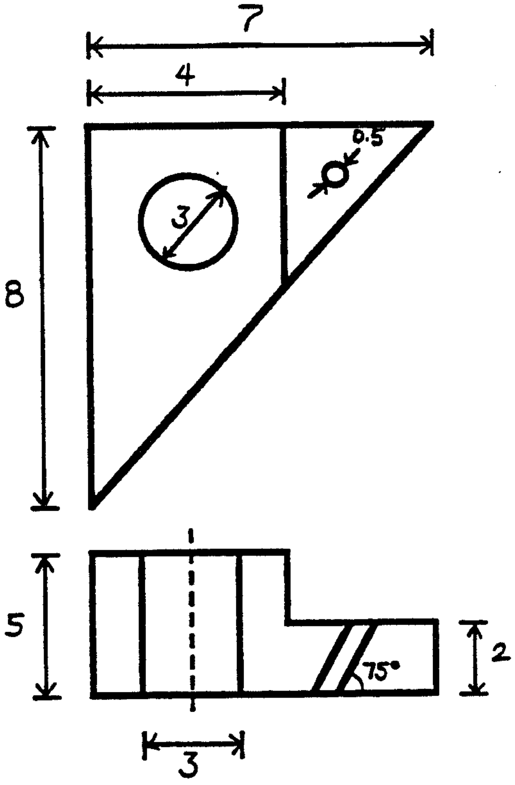


Figure 3 Part with Through Hole and Oblique Hole

The specifications for the part are as follows:

material: cast iron
tolerance of through hole : +/- 0.01 inch
number of pieces to manufacture: 50

The suggested coolant was compressed air, but the experts said that it is very messy for an apprentice to use because it will blow the metal chips all over the place. They suggested using a water-soluble coolant with rust inhibitor, or the aromatic coolant Cool-Tool for low production volumes.

8 FURTHER EXTENSIONS

The present expert system encompasses the major decisions that need to be made for the machine operation of drilling a hole, using manually operated machines. As mentioned earlier, the drilling operation is one of a series of operations that make up a process plan. Since a process plan consists of a collection of operations then the same expert system has the potential of being used with other systems which make the decisions for other operations. A set of meta-rules can be used to determine the order in which these operations are to be performed.

The expert system as it stands has a limited domain. The limits being set by the choice of materials and the types of holes. These may be extended without affecting the system, by including their relevant rules. Also trouble-shooting advice may be added, to help the user solve the common problems encountered when drilling or reaming.

The interview format was used for the knowledge acquisition phase. This knowledge acquisition method evolved from earlier expert systems such as MYCIN, whose experts are people in the medical field. They are usually more articulate than people in the engineering field [5]. Even though the experts consulted for this project articulated the knowledge to our satisfaction, but as the domain becomes more involved and the number of variables increases, the use of automated knowledge acquisition systems would be advantageous.

Appendix A
A Sample Consultation

Problem 1

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

 This expert system helps you select the variables
 to be set in drilling a hole. The input to the
 system is the information supplied on the blue-print
 of the part to be drilled.

 If you need help in answering the questions
 then type - help - instead of answering the question
```

```
What is the type of hole to be drilled?
     a. through hole
     b. oblique hole
     c. counter bore
     d. counter sink
c
What is the depth of the hole to be drilled?
help

The depth of the hole is the thickness of the part
to be drilled in inches. The minimum value allowed is
0.15 inch, below this limit the part is considered
to be sheet metal and should be punched rather than
drilled.
For counter-bore and counter sink holes give the
depth of the smaller hole.

What is the depth of the hole to be drilled?
3
Enter the size of the hole to be drilled?
help

Enter the diameter of the hole in inches,
as specified on the blue-print.
For counter-bore and counter sink holes give the
diameter of the smaller hole.

Enter the size of the hole to be drilled?
0.75
Is the Brinell hardness specified on the blue-print?
no
What is the material of the work-piece?
     a. aluminum
     b. copper
     c. cast iron
     d. brass
     e. mild steel
     f. carbon steel
     g. alloy steel
e
Is the production volume large?
help

If more than 100 parts with one or two holes, or
one part with many holes need to be drilled then
answer - yes -

Is the production volume large?
no
Is the tolerance of the hole specified on the blue print?
no
```

```
Is a good surface finish needed?
yes
Is the hole concentric with respect to the part?
help

If the hole lies along the center axis of the part
then answer - yes -

Is the hole concentric with respect to the part?
no
Is the production time limited?
no


Recommended machine is mill
Drill tool type: drill and piloted counter bore
Material of drill tool: HSS
Diameter of drill tool: 0.735 inch
Use tool with standard helix flutes and straight shank
Start drilling with a speed range between 500 and 600 rpm

Reamer type: chucking reamer
Material of reamer: HSS
Diameter of reamer: 0.75 inch
Reaming speed: 165 rpm

46 rules fired
```

Problem 2

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

 This expert system helps you select the variables
 to be set in drilling a hole. The input to the
 system is the information supplied on the blue-print
 of the part to be drilled.

 If you need help in answering the questions
 then type - help - instead of answering the question

What is the type of hole to be drilled?
     a. through hole
     b. oblique hole
     c. counter bore
     d. counter sink
b
What is the depth of the hole to be drilled?
2
Enter the size of the hole to be drilled?
0.5
```

```
Is the Brinell hardness specified on the blue-print?
no
What is the material of the work-piece?
     a. aluminum
     b. copper
     c. cast iron
     d. brass
     e. mild steel
     f. carbon steel
     g. alloy steel
c
Is the production volume large?
no
Is the tolerance of the hole specified on the blue print?
yes
What is the tolerance of the hole in inches?
help

Enter the absolute value of the tolerance in inches.

What is the tolerance of the hole in inches?
0.01
Is the hole concentric with respect to the part?
no
Is the production time limited?
help

If the time allocated for machining the part is
limited then answer - yes -

Is the production time limited?
no


Recommended machine is mill with appropriate fixturing
Drill tool type: jobber drill
Material of drill tool: HSS
Diameter of drill tool: 0.485 inch
Use tool with standard helix flutes and straight shank
Start machining with an average speed of 300 rpm

Reamer type: chucking reamer
Material of reamer: carbide
Diameter of reamer: 0.5 inch
Reaming speed: 90 rpm

46 rules fired
```

Appendix B
Partial Code Listing

```
(defrule question-tolerance
   ?rem <- (ask-question)
   (tolerance-available yes)
   (not (tolerance ?))
   =>
   (retract ?rem)
   (printout t "What is the tolerance of the hole in inches? "
         crlf)
   (bind ?x (read))
   (if (eq ?x help)
   then
     (printout t crlf)
     (printout t "Enter the absolute value of the tolerance in
         inches. " crlf)
   else
     (assert (tolerance ?x))))

(defrule question-surface
   ?rem <- (ask-question)
   (tolerance-available no)
   (not (reaming ?))
   (not (surface-finish ?))
   =>
   (retract ?rem)
   (printout t "Is a good surface finish needed? " crlf)
   (bind ?x (read))
   (if (eq ?x help)
   then
     (printout t crlf)
     (printout t "When a good surface finish is needed then answer
         - yes -  this will determine whether the part needs to be
         reamed or not." crlf)
   else
     (assert (surface-finish ?x))))
(defrule reamer-speed1
   "reamer speed is one-third of drilling speed"
   (reaming yes)
   (speed ?var1)
   =>
   (bind ?var2 (* 0.3 ?var1))
   (assert (reamer-speed ?var2)))

(defrule no-reaming1
   "if no tolerance available and rough surface finish, then don't
ream"
   (tolerance-available no)
   (surface-finish no)
   =>
   (assert (reaming no)))
```

```
(defrule no-reaming2
   "if no tolerance available but a good surface finish needed,
then ream"
   (tolerance-available no)
   (surface-finish yes)
   =>
   (assert (reaming yes)))

(defrule ream-based-on-tolerance
   "if the specified tolerance <= 0.005 inch then ream"
   (tolerance-available yes)
   (tolerance ?var)
   (test (<= ?var 0.005))
   =>
   (assert (reaming yes)))

(defrule default-reamer-material
   "default reamer material"
   (declare (salience -10))
   (reaming yes)
   (not (reamer-material ?))
   =>
   (assert (reamer-material HSS)))

(defrule print-ream-recommendations
   " if reaming is required then print its recommendations"
   (print-drill)
   (reaming yes)
   (reamer-type $?type)
   (reamer-material ?mat)
   (reamer-diameter ?dia)
   (reamer-speed ?speed)
   =>
   (printout t "Reamer type: " $?type crlf crlf)
   (printout t "Material of reamer: " ?mat crlf crlf)
   (format   t "Diameter of reamer: %g inch" ?dia)
   (printout t crlf crlf)
   (printout t "Reaming speed: " ?speed " rpm" crlf crlf))
```

References

[1] Kumara, S., S. Joshi, R. Kahyap, C. Moodie, and T. Chang, "Expert Systems in Industrial Engineering," Int. J. Prod. Res. vol. 24, no. 5, pp. 1107-1125, 1986.

[2] Donaldson, C., G. LeCain, and V. Goold, Tool Design. McGraw-Hill Book Company, 1973.

[3] Oberg, Erik, Franklin Jone, and Holbrook Horton, Machinery's Handbook: A Reference Book for the Mechanical Engineer, Draftsman, Toolmaker, and Machinist. Twenty-first Edition, 3rd. Printing, 1980.

[4] Parsaye, Kamran, and Mark Chignell, Expert Systems for Experts. John Wiley & Sons, Inc., 1988.

[5] Lu, S. C-Y., "Knowledge-Based Expert Systems: A New Horizon of Manufacturing Automation," Knowledge-Based Engineering Systems Research Laboratory, Dept. of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois.

omit

A14 Session:
Simulation and Defense

S40-63
64165

N96-12953

9

Knowledge/Geometry-based Mobile Autonomous Robot Simulator KMARS

Dr. Linfu Cheng, John D. McKendrick and Jefferey Liu
Elcee Computek, Inc., Boca Raton, FL 33431

ABSTRACT

Ongoing applied research is focused on developing guidance systems for robot vehicles. Problems facing the basic research needed to support this development (e.g., scene understanding, real-time vision processing, etc.) are major impediments to progress. Due to the complexity and the unpredictable nature of a vehicle's area of operation, more advanced vehicle control systems must be able to learn about obstacles within the range of its sensor(s). A better understanding of the basic exploration process is needed to provide critical support to developers of both sensor systems and intelligent control systems which can be used in a wide spectrum of autonomous vehicles.

Elcee Computek, Inc. has been working under contract to the Flight Dynamics Laboratory, Wright Research and Development Center, Wright-Patterson AFB, Ohio, to develop a Knowledge/Geometry-based Mobile Autonomous Robot Simulator (KMARS). KMARS has two parts: a geometry base and a knowledge base. The knowledge base part of the system employs the expert-system shell CLIPS ('C' Language Integrated Production System) and necessary rules that control both the vehicle's use of an obstacle detecting sensor and the overall exploration process. This initial phase project has focused on the simulation of a point robot vehicle operating in a 2D environment. Obstacles were depicted as complex (non-convex) polygons and the vehicle movement was constrained to the x-y plane. Rules controlling the vehicle's motion in free-space activated, when necessary, a sensor that derived obstacle information put into CLIPS working memory. The vehicle must use its sensor to learn about obstacles blocking its path toward the goal and what obstacle vertices can be seen from a given vehicle location. Factory supplied sensor technical performance specifications (e.g., range and bearing) can be selected under the "Sensor" menu option. The user can also select a number of "Display" options that show various aspects of the vehicle's environment (e.g., vehicle track, vehicle location, portions of obstacles discovered, etc.). With the use of an "Obstacles" option, a user can create new obstacles, delete and/or move old ones to new positions. Control of the CLIPS knowledge base activities is accomplished through various "Explore" menu options. A plan view of the environment on the screen, allows the user to monitor the progress of exploration and information being accumulated in working memory.

It is anticipated that this research will progress to develop operational capabilities for 3D environments.

I. INTRODUCTION

A. Need For Autonomous Systems

Today, autonomous systems are required for tasks in hazardous environments (ie., toxic, radioactive, etc.) that are extremely injurious to human health. Additionally, capabilities for autonomous operations are needed in those environments that are characteristically unstructured and, as such, are unpredictable. These environments may be the result of a catastrophe or the characteristics of the environment may have been unpredictably altered since last being visited. An autonomous system must be able to use its sensor(s) to detect the presence of and the locations of objects in an unknown environment. The system must also be able to incorporate updated spatial information into its task-reasoning capability.

B. Background Research Efforts

An autonomous vehicle's efficient utilization of available information for the purpose of exploration and navigation is a key problem in robotic research. The simplest expression of the problem of motion amongst obstacles is that of a point automaton which can move in the 2D plane, avoiding obstacles [1,3]. Research into robot motion planning has been approached from two different vantage points, each based on different assumptions about information or knowledge that the automaton has about its surrounding environment. In the first approach [4,10] the automaton is assumed to possess complete, a priori, knowledge of all aspects of each obstacle. Under this assumption, the vehicle's movement problem is that of "path planning with complete information," and the planning of an optimized path can be a one-time computation. Because all spatial information about the environment is known at the onset of vehicle operations, there is no need to use a sensor to acquire new information about the location of obstacles.

In the other approach, the automaton is assumed to have no knowledge or only limited knowledge of its surroundings [2,6,7,9]. The vehicle must rely on some sort of sensing capability to gather information about the environment. There is no opportunity for optimized transits to all parts of the environment until all aspects of the environment have been fully learned. However, once complete spatial knowledge has been accumulated for a certain region of the environment (ie., a complete regional map is available), regionally optimized transits to goals within this region can be undertaken. In this situation, regional path planning can be a purely computational process and no further sensor operations are required in that region. There may still be other unknown regions of the environment in which sensor operations will be required when transits into or through those regions are required.

Prior research has concentrated on robots operating in known environments and on algorithms for finding globally optimized paths. Research into algorithms for exploring and navigating in unknown environments is less able to address the problem of path optimization to a goal.

There is a need for the capability to simulate exploration and navigation activities so that the efficiencies of various autonomous systems techniques both for vehicle movement control and for sensing operations can be more fully assessed.

II. Exploration and Navigation

Navigation conveys the sense of directing the course of a mobile system based on an a priori knowledge of where impassable areas are located which have to be avioded.

Exploration concerns the initial acquisition of knowledge of where an object is located. Usually the discovery as to the existence of an object is made through a "sighting" of the object and a recording of its location is made. The format of the record of object locations can be either textural or spatial (i.e., map) such that the information can be readily used for subsequent navigation.

The acquisition of spatial knowledge involves three activities: (1) the use of a sensor (e.g., vision, sound, touch, etc.), (2) recording of spatial detail for possible future use, and (3) movement to a new vantage point for the reapplication of (1).

A. Expert System for Unknown Exploration and Navigation

It has been found that the use of expert systems combined with modular procedures provides a convenient and powerful method for controlling a robot vehicle's behavior [5, 11, 12]. It is possible to use an expert-system shell to make high-level decisions concerning exploration and navigation via the shell's internally implemented inferencing procedure. Within the shell, learning can be emulated through the updating of information into working memory.

Knowing nothing about what lies between it and a desired goal position, the first need of an autonomous system equipped with a vision/ranging system, is for its sensor to be activated to "see" if the goal can be detected. If the goal is visible, the implication is that there are no obstacles in the path of the vehicle [infinite width vehicle] and it can move directly to the goal. By treating the vehicle as a point, there are no passages too small for the vehicle to pass through.

A state-space representation of the exploration and navigation process is shown in figure 1. In an unknown environment a vehicle would be operating in the states in the upper-right portion of the graph. As more information is acquired, the vehicle might be operating in the states in the lower-left portion of the graph.

B. Sensor Operations

Long range sensor operations (vision/ranging) are not essential for a system to find a goal in an unknown environment. It has been shown that a goal can be found with a sense of touch and continuous knowledge of the direction to the goal [5]. Although some research has addressed the exploration and navigation process utilizing unlimited range sensors, little research has focused on how sensor range limitations affect the process.

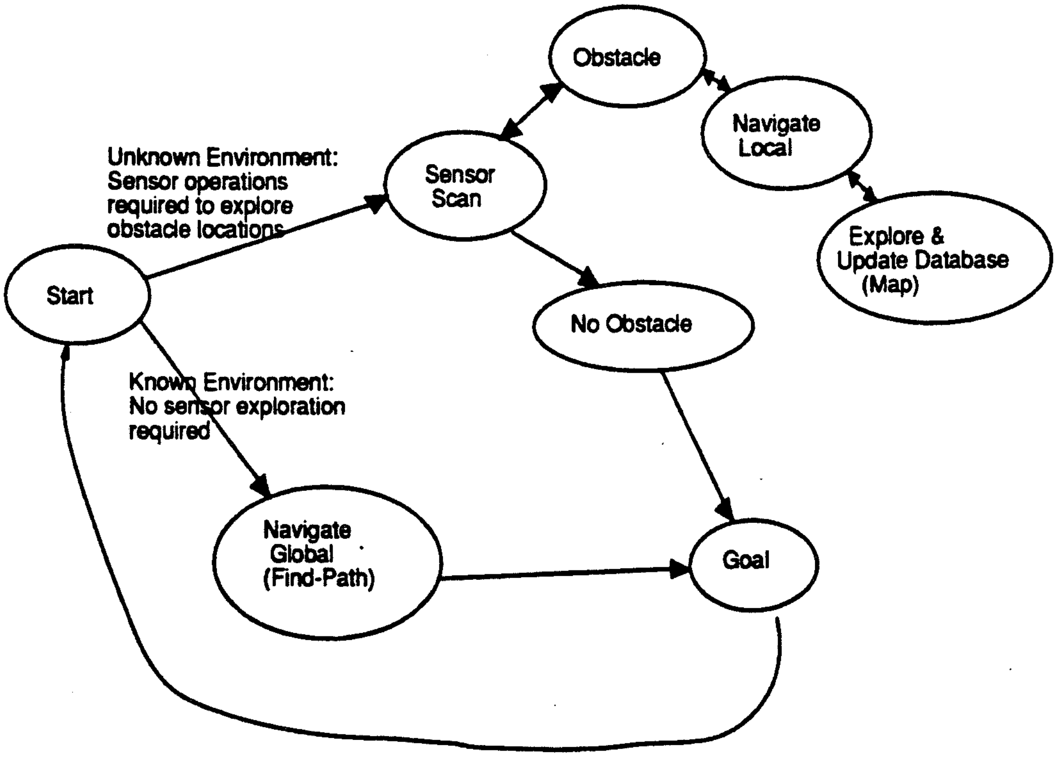

Obstacle
Navigate Local
Unknown Environment: Sensor operations required to explore obstacle locations
Sensor Scan
Explore & Update Database (Map)
Start
No Obstacle
Known Environment: No sensor exploration required
Navigate Global (Find-Path)
Goal

Figure 1. - State-Space Representation of Exploration Problem vs. Find-Path Problem

III. KMARS System

A KMARS user can specify the shape of and the placement of polygonal obstacles in a 2-dimensional environment, select characteristics for a sensor used by the robot vehicle, and compose rules that control the vehicle's activities in exploring the unknown environment. The firing of a rule might activate 'C' functions that perform necessary vehicle tasks. Figure 2 shows the relation between CLIPS and the activation of 'C' functions. The function may return updated information to CLIPS working memory.

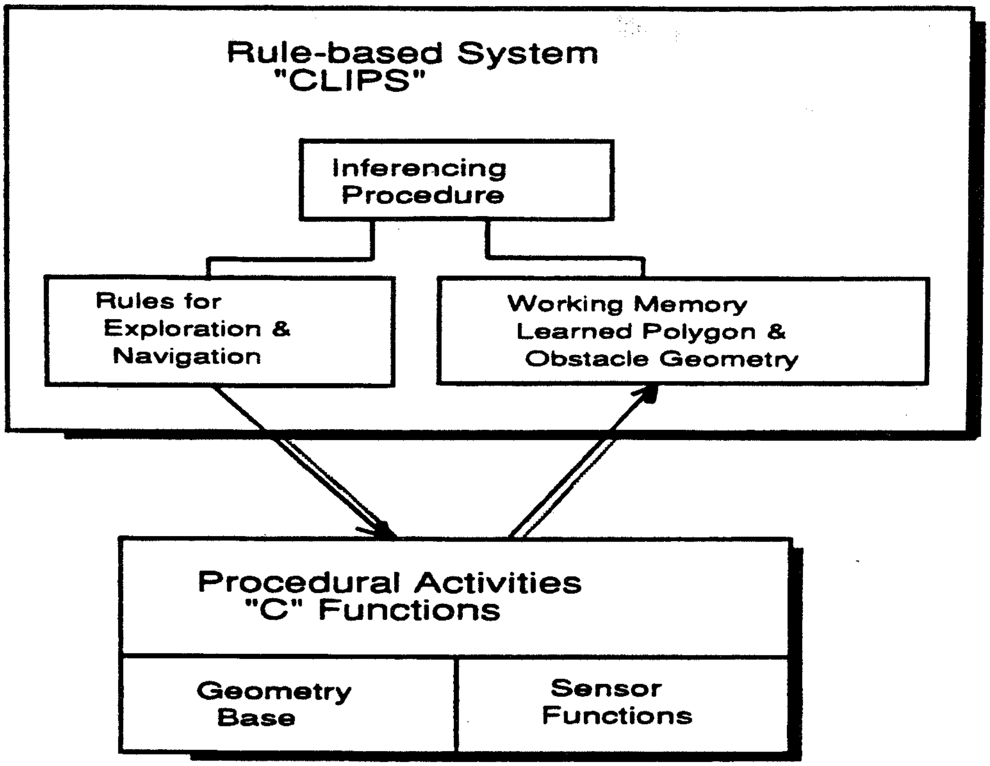


Figure 2 - The KMARS Control System Architecture

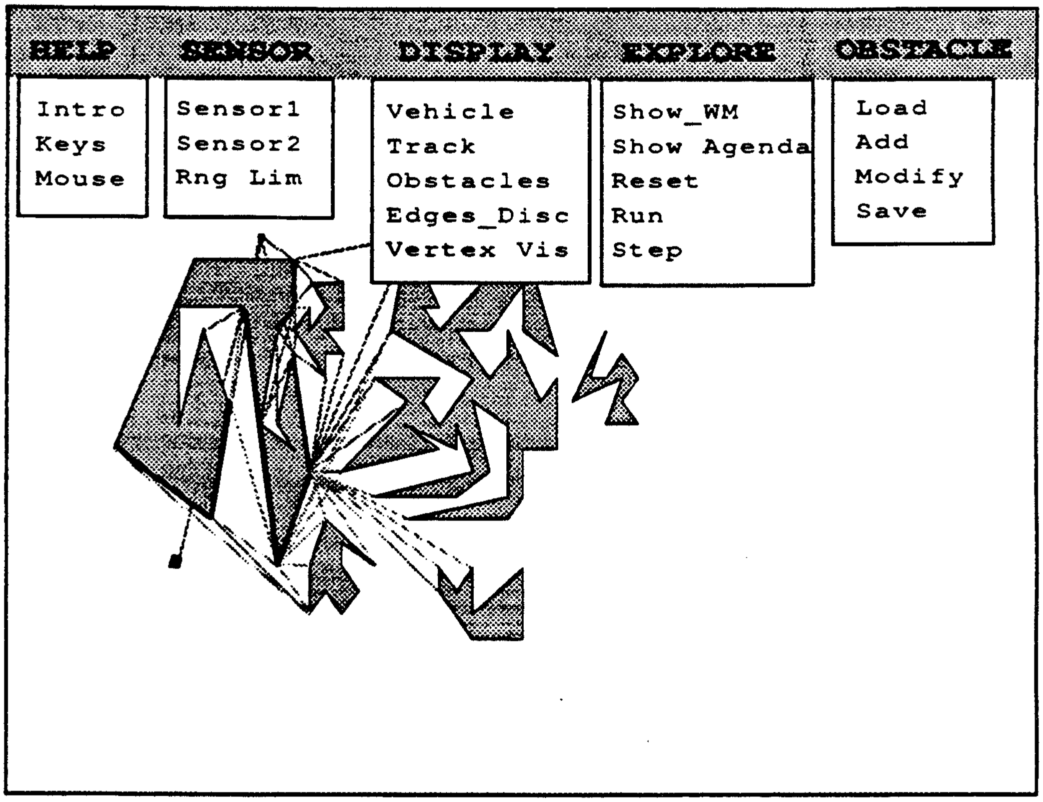


Figure 3 - KMARS Menu Bar and User Selection Options

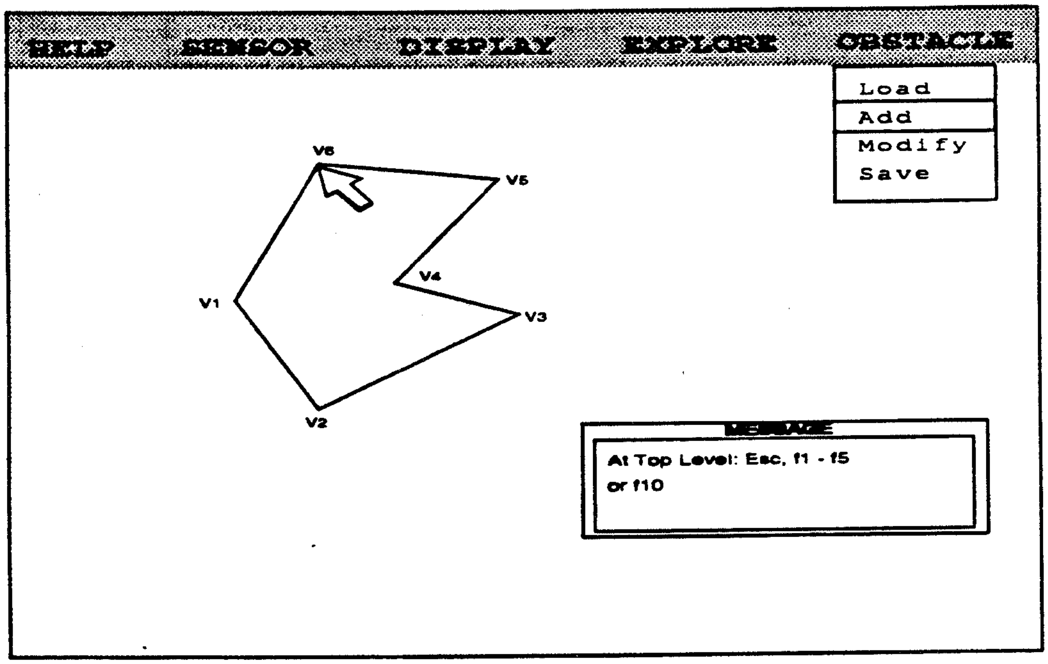
The user's computer screen shows the menu selections and the vehicle's operating area. By using Menus, the user can select various KMARS operating options. The Menu Bar and individual Menu Selections are shown in Figure 3.

A. Geometric Model

The operating environment of KMARS can be constrained by the presence of 2-dimensional polygons. They are, generally, non-convex. The geometry-base verifies that a user-specified obstacle is a valid polygon and checks to insure that a newly defined polygon does not overlap one that has previously been defined. The geometry model also generates an edge-vertex matrix which relates visibilities amongst all polygon vertices. The matrix stores the polygon vertex visibility information.

A goal is obscured if the line-of-sight to the goal intersects a polygon edge. The question of visibility involves a computation that is handled by the [analytical] geometric model. The simulation and manipulation of 2D obstacles in KMARS is maintained by the geometry base that models the 2-dimensional polygon objects and the model provides information about the properties of the polygonal objects.

Polygon obstacles can be created, moved, deleted from the operating environment by the KMARS user. Figure 4 shows a polygon obstacle being created by point and click of the mouse at the position a polygon vertex is desired.



At Top Level: Esc, f1 - f5 or f10

Figure 4 - Creation of Polygon Obstacle

B. Sensor information into working-memory

Figure 5 shows an environment with several 2D obstacles and a

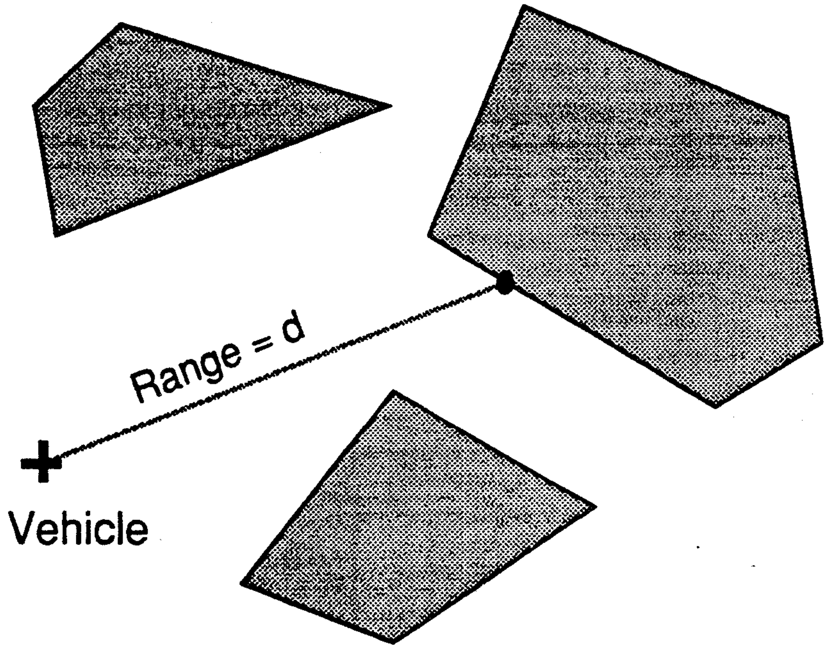


Figure 5 - Laser-Type Narrow-Beam Sensor Detection of an Obstacle

narrow-beam (e.g., laser) sensor pointing toward the goal. An object is detected at some range (d), less than the sensor's limiting range. The detection of an obstacle implies that the goal can not be seen from the vehicle's present position.

A vehicle operating in the KMARS environment can encounter three typical situations which a sensor attempting to check the visibility of a goal might experience:

- the goal is visible
- the goal is blocked by some distant edge
- the goal is blocked by the adjacent polygon

Once a rule has activated the sensor, a 'C' function is called which computes the visibility condition based on information (e.g., current vehicle position, goal position, etc.) passed to the function. The function also calculates spatial information about goal visibility and inputs it into working memory.

C. KMARS Rule-Base

The exploration and navigation activities of the vehicle in KMARS is controlled via a rule-based system. This rule-based system, using data received from its sensors, maps out the "visible" portion of the environment as the vehicle traverses toward a defined goal. The mapping is handled by additions and deletions of spatial facts to working memory.

KMARS has a basic exploration and goal finding strategy and some added rule refinements. The basic strategy is to move to the position beside the polygon that blocks the view toward the goal. From there, if the polygons left-most vertex has not been explored, the vehicle moves to that vertex. The vehicle then calls for a sensor activation and all of the other polygon vertices visible from that vertex are noted. Following vertex exploration, a sensor scan toward the goal is made. If the goal is visible, the vehicle moves to the goal. Otherwise, the vehicle moves to the vertex, right-most from the present vehicle location. If that vertex has not been explored, a sensor activation is made and all of the other vertices visible from that vertex location are noted. Next, another sensor activation determines if the goal is visible. If the goal is obscured by another polygon edge, the above process is repeated. Figure 6 shows the interaction within the KMARS rule-set.

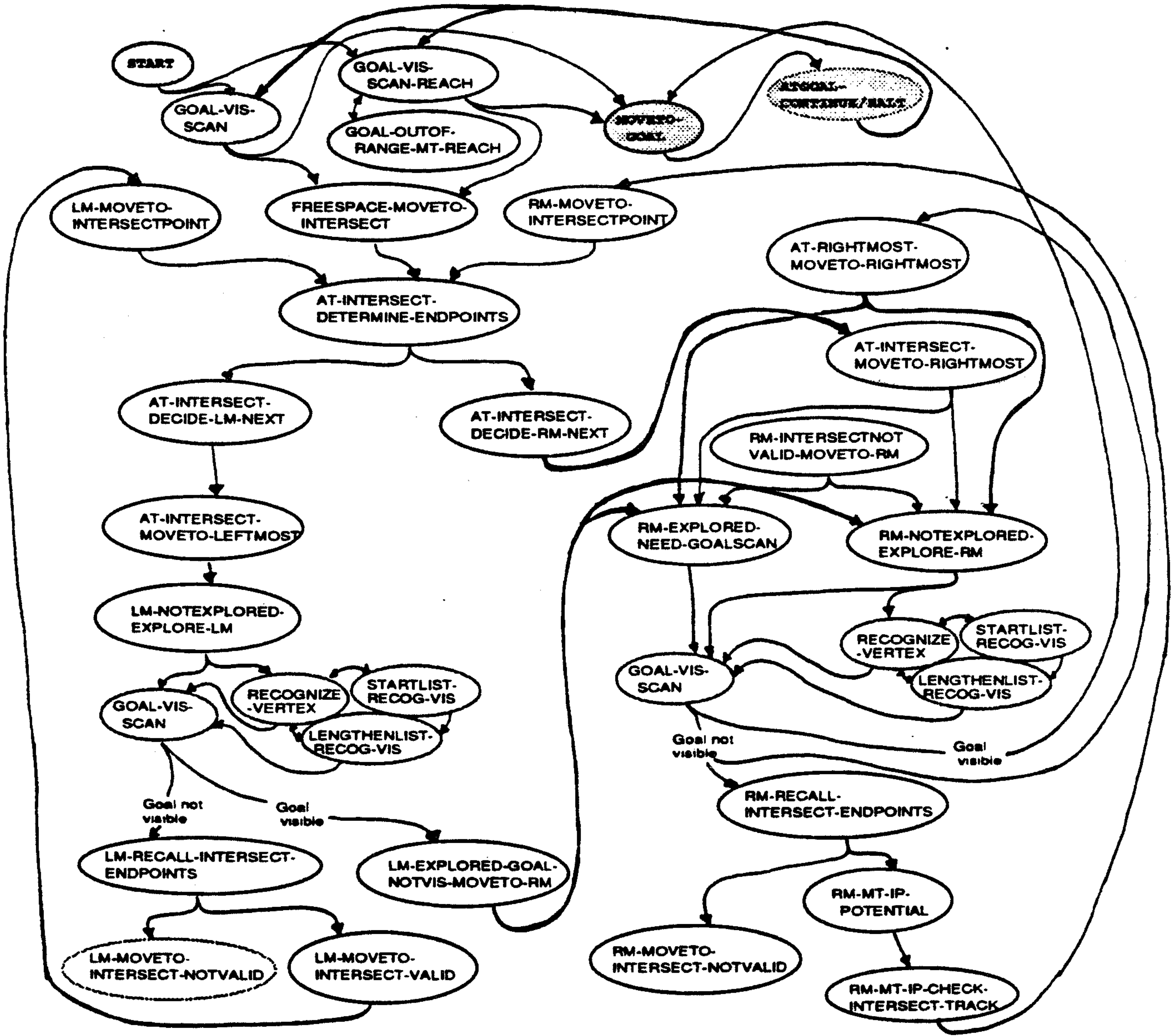


Figure 6 - KMARS Rule-Set Interactions

D. Example of KMARS Rule-Base 'C' Function Interaction

Once the preconditions for the sensor activation rule GOAL-VIS-SCAN have been met, a 'C' sensor function "clips_goal_vis" is called and the necessary arguments that specify both the present vehicle position and the goal position are passed to it.

Using those argument values, the function checks to find the polygon boundary [closest] that blocks the line-of-sight between the vehicle and the goal. If the line-of-sight is not blocked and the goal is within sensor range, the goal_vis function updates working memory with a fact noting that the goal is visible. When the goal is visible, a rule moving the vehicle to the position of the goal is fired. If the goal is blocked by a polygon edge, the sensor function updates working memory with a fact noting that the goal is not visible and a fact noting the coordinates of the point of blockage (i.e., the point-of-intersection of the line-to-goal and the closest blocking polygon boundary).

On other occasions, the vehicle may be at the vertex of a given polygon when its scanner is activated. If the goal is within the vertex-angle of this polygon, the sensor updates working memory with a fact giving the coordinates of the next right-most vertex.

```
(defrule START
   (initial-fact)
 =>
   (retract 0)
   (bind ?veh (clips_get_mouse_position Vehicle free-space))
   (bind ?goal_id (gensym))
   (bind ?veh (clips_get_mouse_position Goal ?goal_id ))
   (assert (Edges-Explored 0))
   (assert (Vertices-Explored 0))
   (assert (Goal-Count 0))
   (assert (Explore-Status Goals 0 Vertices 0 Edges 0))
   (assert (Agenda goal-scan)) )
```

Figure 7 - KMARS Rule START

Actions taken as a result of the START rule firing include:

The function "clips_get_mouse_position" is evaluated. The parameters passed from CLIPS to the function are the word 'Vehicle' and the word 'free-space'. The function in turn sends a prompt to the screen instructing the user to click the mouse at the position where the vehicle is to start from. The value returned by this function is bound to the dummy variable ?veh.

A symbol, needed to identify the next goal position, is generated and is bound to the rule-variable ?goal_id.

The function "clips_get_mouse_position" is again evaluated. The parameters passed from CLIPS to the function are the word 'Goal' and the word assigned to '?goal_id'. The function in turn sends a prompt to the screen instructing the user to click the mouse at the goal position which the vehicle is to find. The value returned by this function is bound to the dummy variable ?veh.

Input into WM are facts that will be used to keep count of Edges-Explored, Vertices-Explored and Goal-Count. Explore-Status will be used to keep track of and to update the exploration status each time a new goal is achieved. Finally, a fact is put into WM that keeps track of future actions to be undertaken.

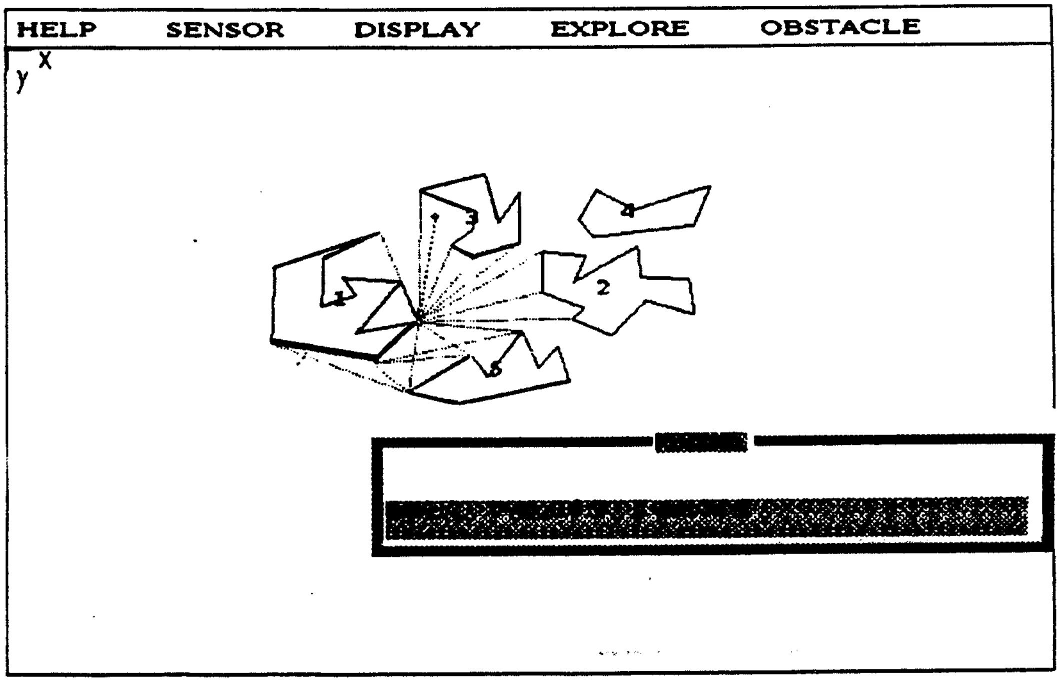


Figure 8 - KMARS Exploration and Attainment of User Defined Goal

E. Results of KMARS Exploration

As the number of traversals to new goals within the environment increases, the exploration of new vertices increases the number of known, obstacle-free paths. These are the vertex to vertex paths that are in working memory. The impact of exploring a vertex is that it does not have to be visited again solely to learn what other vertices can be seen from it. In addition to the saving of time from not having to travel to a known vertex, there is a saving in the time required for sensor exploration scanning. This economy can be monitored as the KMARS vehicle progresses. The next addition to the rule-base should, however, be that of performing a heuristic search of known free paths to find a regionally optimized path to a goal if it is within a totally known region. Although an algorithm has previously been developed to drive exploration and to determine when a complete knowledge of the environment has been acquired, KMARS rules to implement that capability have not yet been developed.

IV. CONCLUSIONS

There is much insight into spatial problem solving that can be derived from using KMARS. In particular this approach allows research into the exploration of unknown environments of an autonomous system. KMARS provides a capability for simulating exploration and navigation activities. The system allows the user to build complex 2D obstacle environments in which to test the efficiencies of various autonomous system vehicle control heuristics. It also allows the user to employ varying sensor characteristics that might be used by a real-world vehicle. Although the strategy implemented in the current rule-set is only one of many that can explore complex 2D environments and achieve goals hidden within the confines of obstacles, the methods implemented in KMARS can be extended to the operations of autonomous systems in real-world 3D environments.

BIBLIOGRAPHY

1. H. Abelson and A. diSessa, *Turtle Geometry*, MIT Press, 1980, pp. 179-199.
2. S.S. Iyengar et al., "Robot Navigation Algorithms Using Learned Spatial Graphs," *Robotica*, Vol. 4, 1986, pp. 93-100.
3. V.J. Lumelsky, A.A. Stepanov, "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment," *IEEE Transactions on Automatic Control*, Vol. AC-31, No. 11, November, 1986, pp. 1058-1063.
4. T. Lozano-Perez and M. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Comm. ACM*, Vol. 22, 1979, pp. 560-570.
5. J.D. McKendrick, *Simulation of Autonomous Knowledge-Based Navigation in Unknown Two-Dimensional Environment with Polygon Obstacles*, MS Thesis, Florida Atlantic Univ., Dept. of Computer Engineering, 1989.
6. J.B. Oommen et al., "Robot Navigation in Unknown Terrains Using Learned Visibility Graphs, Part I: the disjoint convex obstacle case," IEEE *Jour. Robotics and Automation*, Vol. RA-12, 1987, pp. 672-681.
7. N.S.V. Rao,et al, "On Terrain Acquisition by a Point Robot Amidst Polyhedral Obstacles," IEEE *Jour. Robotics and Automation*, Vol. 4, No. 4, 1988, pp. 450-455.
8. N.S.V. Rao, "Algorithmic Framework for Learned Robot Navigation in Unknown Terrains," IEEE *Computer*, Vol. 22, No. 6., June, 1989, pp. 37-43.
9. M. Sharir and A. Schorr, "On the Shortest Path in Polyhedral Space," *Proc. 16th Symposium on Theory of Computation*," pp. 144-153, 1984.
10. P.F. Spelt, G. de Saussure, E. Lyness, F.G. Pin, and C.R. Weisbin, "Learning by an Autonomous Robot at a Process Control Panel," IEEE *Expert*, Winter 1989, pp. 8-16.
11. C.R. Weisbin, G. de Saussure, and D.W. Krammer, "Self-Controlled: A Real-Time Expert System for Autonomous Mobile Robot," *Computers in Mechanical Engineering*, Vol. 5, No. 2, Sept, 1986, pp. 12-19.

EMBEDDED CLIPS FOR SDI BM/C3 SIMULATION AND ANALYSIS

Brett Gossage and Van Nanney
Nichols Research Corporation
4040 South Memorial Parkway
Huntsville, AL 35802

ABSTRACT

Nichols Research Corporation is developing the BM/C^3 Requirements Analysis Tool (BRAT) for the U.S. Army Strategic Defense Command. BRAT uses embedded CLIPS/Ada to model the decision making processes used by the human commander of a defense system. Embedding CLIPS/Ada in BRAT allows the user to explore the role of the human in Command and Control (C^2) and the use of expert systems for automated C^2. BRAT models assert facts about the current state of the system, the simulated scenario, and threat information into CLIPS/Ada. A user-defined rule set describes the decision criteria for the commander. We have extended CLIPS/Ada with user-defined functions that allow the firing of a rule to invoke a system action such as weapons release or a change in strategy. The use of embedded CLIPS/Ada will provide a powerful modeling tool for our customer at minimal cost.

THE PROBLEM

Battle Management, Command, Control and Communication (BM/C^3) systems accomplish the automated control of tactical and strategic military systems. Large-scale BM/C^3 systems such as for the Strategic Defense System (SDS) present several difficult problems. Decision timelines are too short and amounts of information too vast for a human Man-in-the-Loop (MITL) to effectively control or interact with the system without automated decision making or decision support. It is unlikely (and undesirable) that any experience will be gained in actual combat for building a set of rules for an automated SDS decision system. It is also unlikely that the builders of the system will accept full automation of all decision functions. That is, the system designer will require "positive control" of the system by some human commander. Computer simulations of the system are the only currently available method to study these problems. These studies are done in two fundamentally different ways. One is to create simulated command centers with human participants and the other is to use detailed simulations with embedded rules of engagement.

Simulated,"mock-up," command centers with human participants drive real-time displays with discrete simulations or scripts. Separate simulations may generate the scripts independently in non-realtime mode. These scripts have to be generated separately since the run time for full-scale SDS simulations is generally to long for real-time displays. Automated decision software may also be used for decision aids. The main drawback of such studies is that the decisions of the commander cannot affect those parts of the simulation that are run off-line. Thus, the decision loops can only be closed for the more simple parts of the simulation. Closing this loop becomes a Hobson's choice between lowering model fidelity to close the decision loops and leaving some loops open to gain higher model fidelity. However, such simulations provide a means to study the appropriate decision aids and decision criteria for the human commander and provide training for command center personnel.

Another method for studying BM/C^3 decision making is to embed an expert system tool in a simulation of the system of interest. This tool may consist of an inference engine and a rule base.[1] This method allows the closure of all decision loops since running in real-time is not an issue. The main drawback of this method for SDS studies is that no experts exist with the knowledge necessary to derive the rule base. Some the rules can be generated from existing rules of engagement, from experienced SDS simulation engineers, or from personnel who have participated in mock-up SDS command centers. But other rules will have to be generated through experimentation. Rules deemed appropriate in embedded expert system experiments could provide guidance to commanders in mock-up simulators, thus the two methods may complement each other.

THE APPROACH

The requirements for BRAT presented us with several challenging problems. BRAT is required to simulate all phases of SDS operation including peacetime to wartime transition and reaction to failures. The BRAT simulation cannot assume any architecture for the system under study and hence must be able to assemble a simulation from a collection of predefined models. Since the MITL controls the peacetime to wartime transition of the SDS, a BRAT model must be constructed that models the decision processes of the commander. BRAT simulates the system with a large collection of models of varying levels of detail. The BRAT simulation framework integrates these models together employing object-oriented techniques and event graphs.[2] The models capture the physical characteristics of the system, the performance of the automated BM/C^3 functions and the control of the system exercised by the commander. While most of the models can be implemented in procedural code, a model of the commander requires the greater flexibility provided by declarative languages. In BRAT, one model, designated as Command_Defense, accomplishes the simulation of the role of the commander in an SDS system. We have chosen to embed an expert system in the Command Defense model. Command_Defense and its integration with this expert system (CLIPS) are the subjects of the rest of this paper.

To meet the BRAT requirements for modeling the role of the commander and the rules a commander would use to operate the system, we chose to imbed an inference engine in the Command_Defense model. It was further decided that a forward chaining engine would be appropriate since the BRAT simulation is an "event driven" environment.[3] Command_Defense is one of many models that are required for BRAT, so it was not feasible within cost or time constraints to implement an inference engine of our own. CLIPS provided the ideal solution since cost was zero. Also, CLIPS is designed to be embedded in other software which lowered the risk associated with interfacing to stand-alone expert system tools. The major work that remained then was to design and build the interfaces for asserting facts about the system state to CLIPS and to extend CLIPS with user-defined functions that allow rule firings to cause changes in the simulated system state and the current engagement strategy.

IMPLEMENTATION

The BRAT simulation executive and its models are implemented in Ada. As a proof-of-concept we implemented a prototype Command_Defense model using the C-Pragma interfaces provided with the C version of CLIPS. (The Ada version was not available at that time). While this was successful, it caused problems when ported from one environment to another since different Ada compilers implement the C-pragmas differently. A second solution which solved the portability problems was to build a fact file from the Ada code and then execute CLIPS through the operating system. The CLIPS rules wrote all commands generated as a result of rule firings to a file read in by the model when CLIPS terminated. This solution was also unsatisfactory since the process was much slower than a fully embedded design. When CLIPS/Ada became available, the model was redesigned to accommodate it. This resulted in the loss of some CLIPS features such as **bsave** and **bload** which are not now available in the Ada version. The added portability and ease of integration made the switch worthwhile, however.

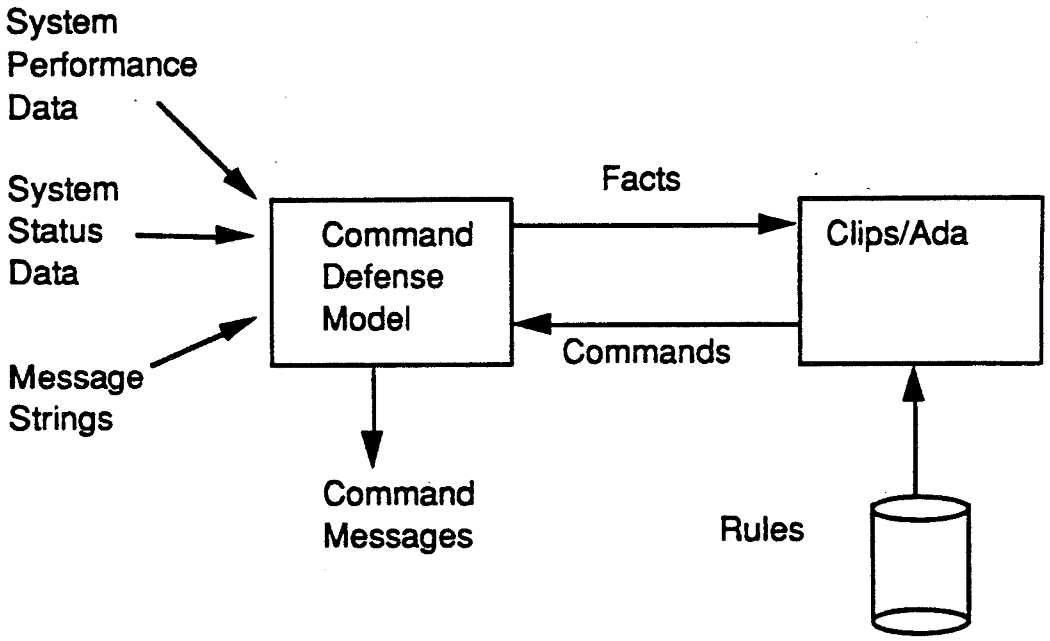


Figure 1. Model Interfaces.

The interfaces to the Command_Defense model occur through three routes. (See Figure 1) The first is the definition of the rules by the user. This is accomplished in the BRAT user interface in a text editor or in the CLIPS stand-alone program. The latter is probably preferable since the user can take advantage the CLIPS system to test the rules before their use by the model. The second interface is the assertion of facts about the current state of the system into CLIPS. This accomplished by converting system information into fact strings and asserting them into CLIPS. The rules bind quantitative system information to variables by pattern matching these facts. The third interface is through the extension of CLIPS with Right Hand Side (RHS) functions. These RHS functions pull information from the CLIPS buffers and insert it into global package data structures. The model reads these global data structures when new commands are to be sent to other models through the simulated communications system.

Rules for the Command_Defense model are divided into three basic types: time-based, relational, and free-form. Time-based rules fire on or after a given simulation time has been reached (see Listing 1). The time fact is bound and then reasserted to allow other time-based rules to fire. A lower salience rule eventually binds the time fact and retracts it. This allows the user to cause system actions to occur such as releasing weapons 300 seconds after simulation start. Frame-based rules use information generated as Ada records by other BM/C^3 models and sent to Command_Defense in messages. Free-form rules can follow any syntax desired and allow the user to define external string messages (such as those generated by external simulations) as Left Hand Side (LHS) patterns for firing rules.

```
   (defrule release "A rule to release weapons at time t"
      (current time ?simtime) ;bind ?simtime to current time
      (release-time ?rtime)   ;bind ?rtime to release time
      (test (>= ?simtime ?rtime)) ; if time >= release t
      ( not(time1-past))          ; and rule not fired yet
   =>                             ; then
      (assert (command RELEASE_WEAPONS))  ;release weapons
      (assert (time1-past))
   )

   (defrule retract-time " so other time-based rules fire"
      (declare(salience -1)) ; lower salience so that all
      ?timefact <- (current time $?); rules for current
                                    ; time fire first
   =>
      (retract ?timefact)           ; retract time fact
   )
```

Listing 1. Example of time-fired rule.

The user is responsible for creating and maintaining the rule-base for the Command_Defense model. Without detailed knowledge of the available fact patterns and RHS function syntax, this task could overwhelm the user and cause errors in rule execution. To ease this burden and assure proper use of the model, "defexternal" and "defrelation" statements provided to the Cross Reference Style Verification (CRSV) utility to assure rule validity [4]. The CRSV tool uses defexternals to assure that RHS function names and arguments are correct. An example defexternal is given is Listing 2. This definition assures that only the available weapon target assignment optimization modes are selected. Defrelations assure that LHS patterns for rules are consistent with the facts asserted by the model. An example defrelation is given in Listing 3. This definition assures that the user does not define a rule for which no valid fact pattern will exist. It also helps to assure that the proper variable bindings will occur.

```
   (defexternal SET_OPT_MODE
      (true-function-name SET_OPT_MODE)
      (min-number-of-args 1)
      (max-number-of-args 1)
      (assert ?NONE)
      (retract ?NONE)
      (return-type NUMBER)
      (argument 1
         (type WORD)
         (allowed-words
            PREFERENTIAL_ASSET_BASED
            PREFERENTIAL_TARGET_BASED
            SUBTRACTIVE))
   )
```

Listing 2. Example defexternal for CRSV.

```
(defrelation threat-data
   (min-number-of-fields 3)
   (max-number-of-fields 5)
   (field 1
      (type WORD)
      (allowed-words threat_class asset_class))
   (field 2
      (type NUMBER)
      (range 1 ?VARIABLE))
   (field 3
      (type WORD)
      (allowed-words count))
   (field 4
      (type NUMBER)
      (range 1 ?VARIABLE))
)
```

Listing 3. Example defrelation for CRSV.

All information about the state of the simulated system is input to CLIPS through facts. Current time is always asserted on each execution of the model. Simulated messages are sent to Command_Defense by other models and are received as Ada records or as strings. The Threat_Assessment and System_Performance models summarize available system information in data records and transmit them in simulated messages to the Command_Defense model. These records contain summary information for system element operational status, weapon system performance, assets threatened, and missile launch fields. Record fields are converted to strings and concatenated with appropriate description strings. For example, the fact (data threat_class 1 count 20) provides the type and number of threat objects of a given class currently detected. The rule base uses this threat information to make inferences about the objective and intent of an attack. The model asserts string messages directly so the user is responsible for assuring that the rules are consistent with them. String message facts allow the user to define arbitrary scenarios for a simulation run. These message facts are defined in an input exogenous event file along with a message arrival time for input to the simulation through event generators.

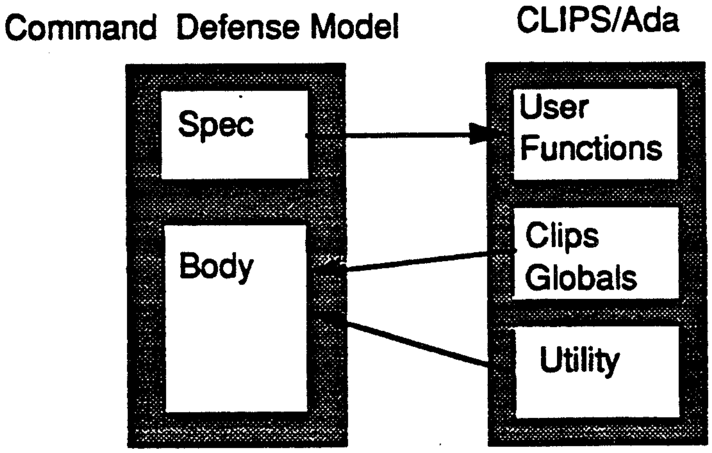


Figure 2. Software Architecture

The RHS functions added to CLIPS which change defense strategies, select types of assets to defend, specify weapon withhold, release weapons, or send strategy change messages. The software architecture for exporting these functions to CLIPS is shown in Figure 2. These functions are exported to CLIPS throughout the model Ada specification file while the code for the functions is kept in the model code body file. Each function pulls the function parameters from the CLIPS buffers and places them in a global strategy variable. When the SEND_STRATEGY_MESSAGE RHS function is invoked, the current strategy is sent to weapon control models. W e expect to continue to expand the number of RHS functions as the BRAT simulation grows. An example of a RHS function which sets the weapon withhold percentage is shown in Listing 4.

```
function SET_WITHHOLD
   (The_Problem : in CLIPS_GLOBALS.Test)
    return CLIPS_GLOBALS.Real is

----------  constants SET_WITHHOLD  -------------------

Check_Value : FLOAT_TYPE_PKG.Float_Type := 0.0;

----------- exceptions SET_WITHHOLD  -----------------

Probability_Out_Of_Bounds_Error : exception;

use FLOAT_TYPE_PKG;
begin
Check_Value := UTILITY.GET_FLOAT_ARGUMENT(The_Problem,1);
if (Check_Value > 1.0) OR (Check_Value < 0.0) then
   raise Probability_Out_Of_Bounds_Error;
end if;
Percent_Withhold := Check_Value;
return 0.0;
------------Exception ---------------------------------
   when Probability_Out_Of_Bounds_Error =>
      BRAT_ERROR_PKG.Log_Error
        ("Invalid probability retrieved from
        CLIPS buffer");
      Raise BRAT_ERROR_PKG.Cc_Function_Error;
end SET_WITHHOLD;
```

Listing 4. Example RHS function.

STATUS AND FUTURE PLANS

As of this writing the Command_Defense model is undergoing integration testing with the BRAT Simulation Executive. Time-based rule firings have been tested in a prototype simulation. The use of defrelation and defexternal statements in the User Interface for rule verification has been defined. All interfaces have been successfully tested and verified.

CONCLUSIONS AND RECOMMENDATIONS

Embedding CLIPS in Command_Defense has proven to be straight-forward, so long as both the model and the CLIPS version are written in the same language. The loss of the **bload** and **bsave** features in the Ada version restricted our ability to build simulations with multiple instances of Command_Defense models. Simulated systems with multiple commanders require multiple model instances for studying devolution of control when primary C2 nodes are lost. An added feature that would be useful in this regard is for **bsave** and **bload** to include the fact list along with the rules. This would allow saving the models perception of the system at a given time to a binary file for fast reloading later. We expect the rule base for the model to expand over time as more users take advantage of its capabilities. We will be defining a baseline set of rules to be delivered with the BRAT product that the user can modify as needed. This may also involve the addition of more RHS functions to CLIPS. In sum, embedding CLIPS in the Command_Defense model has proven to be a powerful, easy-to-use, and cost effective choice for the BRAT project.

REFERENCES

[1] Mitchel, Robert R. "Expert Systems and Air Combat Simulation." AI Expert 4(9). (September 1989): 38-43.

[2] Daniel, Robert S., Gossage, Brett N., Barnett, Gene A. "The Battle Management Requirements Analysis Tool Simulation Environment." Presented at the 1989 Summer Computer Simulation Conference, Austin TX. Nichols Research Corporation.

[3] Baker, Louis. Artificial Intelligence With Ada. McGraw-Hill, New York. 1989.

[4] CLIPS Reference Manual. Version 4.3. Houston, Texas; NASA Johnson Space Center. June 1989.

** The opinions expressed in this paper are those of the authors and do not reflect any official position of U.S. Army Strategic Defense Command (USASDC) or Nichols Research Corporation. The work is supported under USASDC contract DASG60-88-C-0069.

Embedding CLIPS-based Expert Systems in a Real-time Object-oriented Simulation

Patrick McConagha and Joseph Reynolds
Tracor Applied Sciences, Inc.
6500 Tracor Lane
Austin, Texas 78725

1.0 INTRODUCTION

This paper describes our continuing work embedding CLIPS-based expert systems into the System Test Environment (STE)[1]. We are embedding simple, compact rule engines in STE to simulate the actions of Naval platform commanders and equipment operators. Our eventual goal is to implement expert system modules that will replace all human participants and some of the equipment present in the simulation.

This paper will briefly describe STE and then discuss its structure and implementation in more detail. Next, we will consider how expert systems could enhance STE's current capabilities. This will be followed by the examination of a specific CLIPS-based expert system model to be embedded in STE. Finally, a summary of our experience and a discussion of anticipated work on this project will close this paper.

2.0 AN OVERVIEW OF STE

So that the reader will understand the environment into which the CLIPS-based expert systems are to be embedded, we will now briefly describe STE. This discussion will be rather short and high-level. A more complete description of STE can be found in [1], from which the following description has been condensed.

[1] Our work on STE was sponsored by Mr. Steve McBurnett of the Integrated Warfare Branch, Code 5570 of the Naval Research Laboratory (NRL) under Contract # N00014-88-C-2175.

STE is not a simulation in itself but rather a simulator. The purpose of STE is to supply data describing the kinematics, equipment, and operation of Naval assets thereby simulating the "real world". This data provides an environment in which to develop and test operational equipment for the Navy. STE can be considered a test bed on which a large range of simulation experiments will be run.

The initial application of STE was to provide data to stimulate a prototype Anti-Submarine Warfare (ASW) decision aid, called TABS, under development at NRL. A typical configuration of STE for testing TABS is shown in Figure 1. Although STE can and will support testing of a range of experimental equipment, work to this point has been directed toward the requirements of TABS. This paper will address applications of expert systems and issues present in this first application of STE.

2.1 STE Structure

The functional requirements imposed on STE were similar to those for any large-scale simulation test bed. These requirements included the following.

- Modularity - STE must readily accept any extensions needed to provide an acceptable environment to the equipment under test. This means STE must be able to generate all data needed to stimulate a piece of equipment and must deliver that data to that equipment as it would receive it in its operational environment.

- Flexibility - Simulation operators must be able to substitute models with various levels of fidelity as required by the equipment under test.

- Speed - STE must run in real time and take advantage of hardware resources available at NRL.

There were other requirements levied on STE, but the three outlined above are all we need to consider. These requirements resulted in an object-oriented design for STE.

STE objects were designed based on the low-level objects in the Object-Oriented Support Library (OOPS) [2]. The following OOPS objects provided the bases from which all STE objects are derived:

- Movable objects - This category includes platforms such as ships, aircraft, torpedoes, etc. as well as other "movable" objects like minefields, storms, convoy perimeters, and land masses. These objects can move and can have equipment objects (see below) attached to them. Land masses do not move, but they are useful as navigation hazards and where land-based forces, such as aircraft, must be considered.

- Equipment objects - This category includes sensors (sonar, radar, etc.), weapons, communications gear, and ship and equipment commanders. Equipment objects are attached to movable objects by the scenario.

- Environment objects - These objects model the operational environments for sonar, radar, etc. as those environments affect the various pieces of equipment.

- Launcher objects - These objects can create new instances of objects as the simulation progresses. For example, a helicopter launcher creates a new helicopter object and attaches to it any radars, sonars, radios, or other equipment objects specified by the scenario.

- Operator objects - These objects serve as translators between STE and entities in the outside world. These entities can be humans sitting at a console or equipment under test.

- Internal Communication objects - This category includes objects used internally by STE to control data exchange and communication between other simulation objects.

- Miscellaneous objects - This category includes low-level objects such as random number generators used by STE to control the simulation.

One of the obvious benefits of an object-oriented design is that although objects share a common structure, they are very much independent. As long as their interfaces conform to what is expected from specific objects, ships for example, implementation of the ship model is wholly contained in the ship object. In fact, two ships in the same scenario could be modeled quite differently. A ship that controls local air traffic could be modeled at a high level of fidelity while another ship that launches helicopters is simply modeled as a movable platform with a helicopter launcher object attached to it. With this in mind, one or more expert systems can be introduced into this structure in place of algorithmic models or in place of models that require human response. We have done this by replacing the specified models with simple embedded CLIPS-based expert systems. Specific applications of expert system models will be discussed in section 3.

2.2 STE Implementation

STE was written in C++, an object-oriented programming language based on C. It runs on a 128 node Butterfly parallel processor with human interfaces implemented on Sun workstations networked with the Butterfly[2]. The current version of STE provides the simulated environment for the initial TABS prototype. It has been able to satisfy the real time speed requirements of TABS, providing data faster than TABS can process it.

[2] Sun is a trademark of Sun Microsystems, Inc., Butterfly is a trademark of BBN Advanced Computers, Inc.

3.0 USING CLIPS IN STE

CLIPS-based expert systems will be used to automate decision making in STE. These embedded expert systems will replace models that currently require a response from an operator sitting at a console. In some cases, an embedded expert system could replace an algorithmic model or a table look-up model. Any object in STE whose function can be described by a set of rules, however fuzzy, is a candidate for an embedded expert system.

The benefits gained from this effort include the ability to rapidly develop prototype "experts" for specific STE objects in the CLIPS standalone environment. Enhancements to initial implementations of these experts will likewise be a relatively straightforward task. Similarly, "tweaking" the system by reprogramming experts provides a valuable means of studying various effects of different actions taken under similar situations. These trade-off studies are a major part of STE's functionality. Finally, considering a specific function from a rule-based perspective may lead to insights that help us build better algorithmic models.

Objects in STE that are candidates for an expert system model include the following:

- Platform Commander - A human in command of a ship, airplane, or other platform. A platform commander receives data from equipment on his platform and operational orders from his superiors in the chain of command. He must then determine how to best use his platform and the equipment attached to it to carry out his orders.

- Asset Commander - Examples include a Battle Group, Task Force, or ASW commander. This object differs from a platform commander in that an asset commander issues orders and receives feedback from other commanders. An ASW commander, for example, might have frigates, destroyers, and several ASW aircraft at his disposal. In carrying out his orders, he controls these assets by issuing commands to each of the platforms' commanders.

- Equipment Operators - These commander objects operate specific equipment. For example, a sonar operator receives data from his sonar equipment and reports sonar contacts up the chain of command.
- Specific Functions of Equipment - This is where an embedded expert system replaces a traditionally algorithmic function. The track correlator example in section 4 is an example of this application.

To illustrate the application of embedded expert systems in STE consider the following scenario. A task force is leaving port and steaming to its assigned patrol area. The ASW Commander for the task force is ordered to protect the task force from hostile submarines en route to the patrol area. Assets at his disposal include frigates, destroyers, aircraft, and a variety of equipment on each of these platforms. Figure 2 shows the relationships between some of the STE objects that exist in this scenario. Objects that could possibly be replaced by expert system models are so marked. This example is simplistic but it serves to illustrate the breadth of possible applications of expert systems in STE.

4.0 AN EXPERT SYSTEM MODEL FOR A TRACK CORRELATOR

As our first investigation into expert system applications in STE, we implemented a rudimentary track correlator model. This particular object was chosen mainly because its functionality in STE was well understood. Secondly, the track correlator model in place in STE was a very simple one; almost any new model would have been an improvement.

A typical track correlator is a sequential algorithm that does the following. Given a list of established tracks and a set of new sensor reports, the correlator tries to match each new report to an existing track. A new track is created if a new report doesn't correlate with any of the existing tracks. Finally, existing tracks that do not match new reports are dropped. This process is repeated each time a new set of reports is received.

This is a simplified explanation of a track correlator. Specific issues such as how "closely" a new report must match an existing track, what to do when a new track matches more than one existing track, under what circumstances a new track is created, and how old a track must be before it is dropped vary between applications. Nevertheless, the basic functionality of a track correlator is straightforward.

4.1 The Track Correlator Model

Our initial implementation of an expert system track correlator is shown in Figure 3. This program defines four templates that are used by the expert system. The **sim-time**[3] template defines the fact that maintains the current simulated time and time step. Since STE is an event-driven simulation, the time step is not necessarily a constant value but represents the simulated time that has elapsed since the CLIPS rule engine was last called. The **new-report** template defines the format of facts that contain new sensor reports. A sensor report consists of current information about the sensor itself (e.g.position) and information about the detected target such as bearing. A sensor report can contain much more information about the target, but this information varies between types of sensors (active sonar, passive sonar, radar, etc.). Sensor position is useful when trying to localize the target's position; it was not considered in this example. The **current-track** template defines the facts that identify established tracks. A **current-track** fact contains a contact number and a list of times at which a report was received on this target. The **contact** template defines facts that contain the actual data from each specific sighting of a target. A **contact** fact contains the same information as a **new-report** fact with the exception of sensor position. If sensor position were considered in this model, a **contact** fact would contain an estimate of the target's position derived from the sensor's position and its report on the target.

[3] Boldface words name fact templates, facts, or rules. Fixed-width font words denote function or constant names.

This model contains three rules; one to perform each basic function of a track correlator. The first rule, **extend-track**, tries to correlate a new sensor report with an existing track. This rule compares target information in the new report to information contained in the most recent **contact** fact for a given track. An external function, `same_target`, is called to make the comparison. For this simple model only relative bearing of the target is considered. A higher fidelity test could easily be implemented in `same_target` which would then require more arguments to be passed from CLIPS (report times and target characteristics), but the structure of this rule would be essentially the same.

When this rule fires, the **new-report** fact is removed from the fact list and replaced by a **contact** fact. The outside world is notified of the continuing track via another external function call `same_track`. Finally, the **current-track** fact is modified to incorporate the newest contact with the target.

The second rule, **make-new-track**, creates a new track when a sensor report does not match an existing track. It fires when there does not exist a **contact** fact in the fact list that correlates with the new report. The `same_target` test is used as a predicate function inside a negated pattern to perform this test. As in the **extend-track** rule, the **new-report** fact is replaced by a **contact** fact in the fact list when this rule fires. The outside world is notified of the track creation via a call to the external function `new_track`. Finally, a **current-track** fact is created with a unique track number and asserted. The track number is derived from a track counter fact that is initialized in a `deffact` statement.

The last rule in this model, **lost-track**, fires when no new report is received for an existing track. After **extend-track** and **make-new-track** have fired for each of the extended and new tracks, respectively, **lost-track** simply checks if the most recent contact in an existing track was received before the start of the current CLIPS execution cycle. The **sim-time** fact used in this rule is updated before each execution cycle by the calling program. When this rule fires, it simply reports the loss of contact by calling the external function `no_contact`. Discontinued tracks are not removed from the fact list in this model.

4.2 Running the Track Correlator Model

The 'C' program shown in Figure 4 was used to demonstrate the execution of the expert system track correlator model. The program first opens a data file that contains time and bearing information. Next, it initializes CLIPS, loads the rule base, and resets CLIPS. It then works through the data file building and asserting the **sim-time** fact containing the current simulated time and time step, building and asserting **new-report** facts for each bearing given at the current time (a negative bearing in the data files represents an execution cycle where no new reports are received), runs CLIPS, and retracts the **sim-time** fact. The **sim-time** fact is asserted using the `assert` command so that it may be retracted later. The **new-report** facts are asserted via the more efficient `add_fact` mechanism.

The program listing in Figure 4 also contains the declaration for the external functions called by the track correlator (in `usrfuncs`) and the functions themselves. The `same_target` function simply compares the two parameters and returns `TRUE` if they are within a specified tolerance. Otherwise it returns `FALSE`. The `same_track`, `new_track`, and `no_contact` functions simply print informative messages to the screen.

A sample data file and execution output is shown in Figure 5. Several test data sets were executed to examine the performance of this track correlator model under a wide variety of operating environments. These tests were run on a 20 mHz, 80286-based personal computer. Sample execution times are shown in Tables 1 through 5. Each table shows the time, in seconds required to complete a single iteration of the main loop of the 'C' driver program (see Figure 4). The different number of tracks represent the number of targets being tracked by the system. This value increases as more targets enter the scenario. The maximum number of contacts represents the maximum number of times the system has detected a specific target. This value generally increases as the length of the simulation increases. The number of new reports represents the number of sensor reports received in the current execution cycle. It increases with the number of targets present at the current simulated time.

Not surprisingly, execution time increases with an increase in the number of tracks, contacts, and new reports. While this seems reasonable, the amount of increase was unexpected. Further analysis of the model revealed several improvements which might improve performance.

The **extend-track** rule was relatively straightforward. Maintenance of track information in the fact list was costly. A better implementation might have the `same_track` function update an external database where track histories are stored. The `same_target` test could then access the database to determine track continuity. This would be useful as the need for a more sophisticated correlation test is realized.

The **make-new-track** rule was a little more confusing. The use of a predicate function within a negated pattern circumvented the CLIPS rule that `and` constraints were not allowed inside a negated pattern. This implementation, however, resulted in numerous calls to the `same_target` function. In fact, since the **make-new-track** rule did not limit its correlation attempts to just the most recent **contact** fact for each target, the assertion of a **new-report** fact resulted in a call to `same_target` for each **contact** fact in the fact list. This means that `same_target` was called once for each **current-track** fact and once for each **contact** fact in the fact list each time a **new-report** fact was asserted. With three current tracks consisting of four contacts each and only two new reports, `same_target` would get called seven times when the first report is processed and nine times when the second report is processed (the first report either lengthened an existing track or established a new one).

The initial implementation of the **lost-track** rule was poor. It was activated for every track maintained in the fact list at the beginning of each execution cycle. Because of the salience declaration, activations of **extend-track** fired and removed activations of **lost-track** for those tracks that were extended in the current execution cycle. **lost-track** was modified and the salience declaration was replaced with a (not (new-report)) constraint. Along with minor changes to **extend-track** (retraction of the **new-report** fact was delayed until the track was updated) and the test program (assertion of the new **sim-time** fact was delayed until after all **new-report** facts were asserted), this change ensured that **lost-track** would not be activated unnecessarily. However, this "improvement" actually resulted in slightly LONGER execution times. A seemingly obvious improvement to the model resulted in a degradation of performance.

5.0 CONCLUSIONS

We have successfully implemented a low-fidelity model of a track correlator using CLIPS. This model takes advantages of many of the features CLIPS offers for embedded expert systems. More importantly, the experience gained while working on this model will allow us to design and implement better models for a wide range of functions within STE. We plan to continue our work developing and improving these models. The track correlator we examined in this paper may not ever be used in an STE simulation, but it has demonstrated that simple rule-based models will have a place in the real-time, object-oriented environment of STE.

We have ported CLIPS to a Sun workstation and to the Butterfly computer at NRL. The track correlator model has been run successfully on both. The next major task ahead of us is to modify CLIPS so that multiple expert systems can run concurrently on the Butterfly. From there we can integrate working expert system models into STE.

TABLE 1
Execution times with zero tracks

0	1	2	5	number of new reports
.02	.05	.06	.17	execution time

TABLE 2
Execution times with 1 track

		number of new reports	
		0	1
maximum number of contacts	1	-	.05
	2	-	.05
	3-5	-	.05
	6-10	.03	.06

TABLE 3
Execution times with 2 tracks

		number of new reports		
		0	1	2
maximum number of contacts	1	-	-	.08
	2	-	-	.06
	3-5	-	-	.11
	6-10	-	.09	.13
	15	.04	-	-

TABLE 4
Execution times with 3-5 tracks

		number of new reports				
		0	1	2	3	5
	1	-	-	-	-	.33
	2	-	-	-	.11	.38
maximum number of contacts	3-5	.06	.11	.16	.16	.59
	6-10	-	-	.28	-	.97
	15	-	.18	-	-	-
	20	.05	.24	.39	-	-

TABLE 5
Execution times with up to 49 tracks

		number of new reports								
		2	3	4	5	6	7	8	9	10
maximum number of contacts	35	1.92	2.93	3.73	5.61	7.47	9.50	12.30	15.90	20.80
	49	-	-	-	-	13.95	-	-	-	-

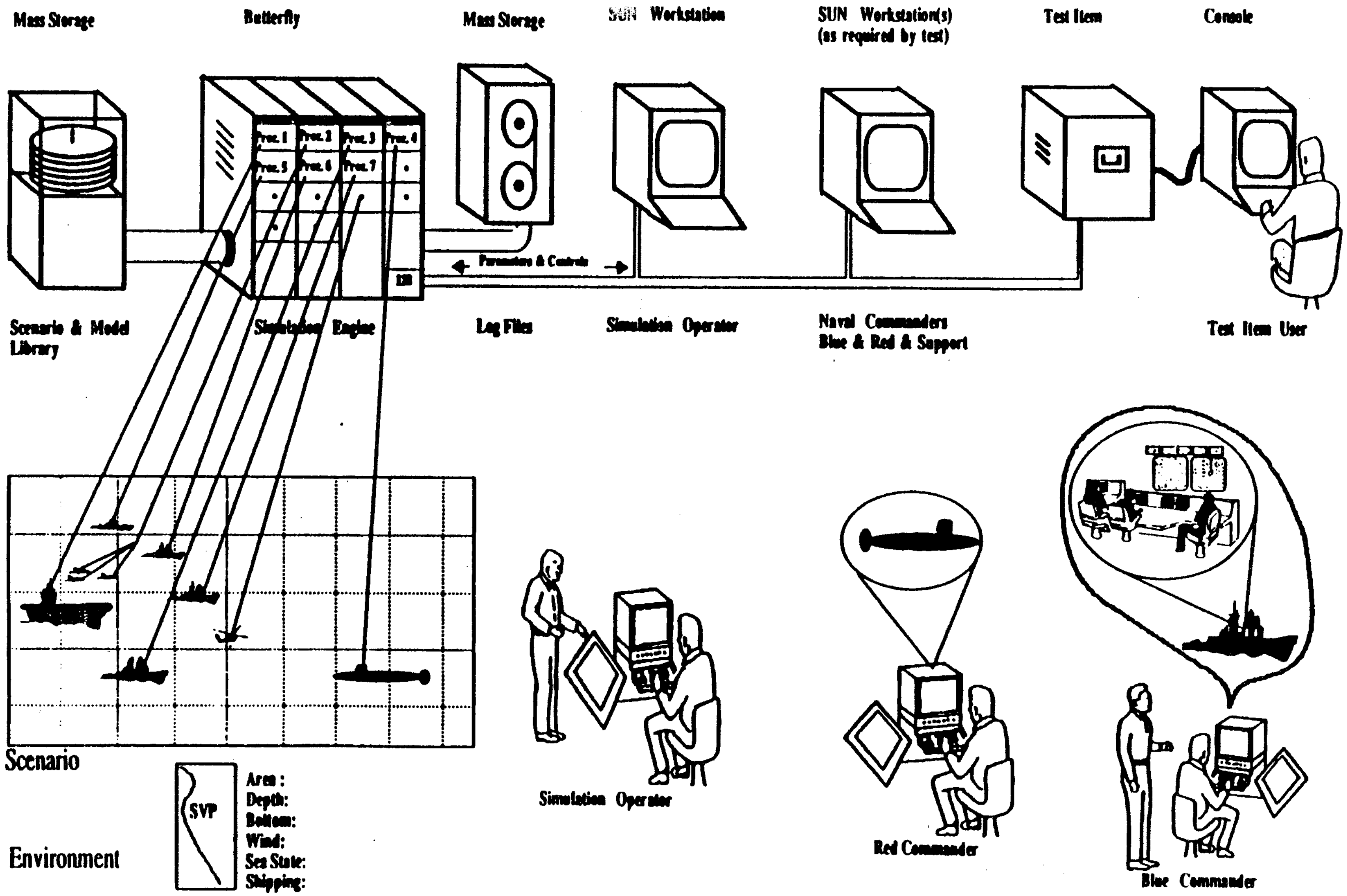
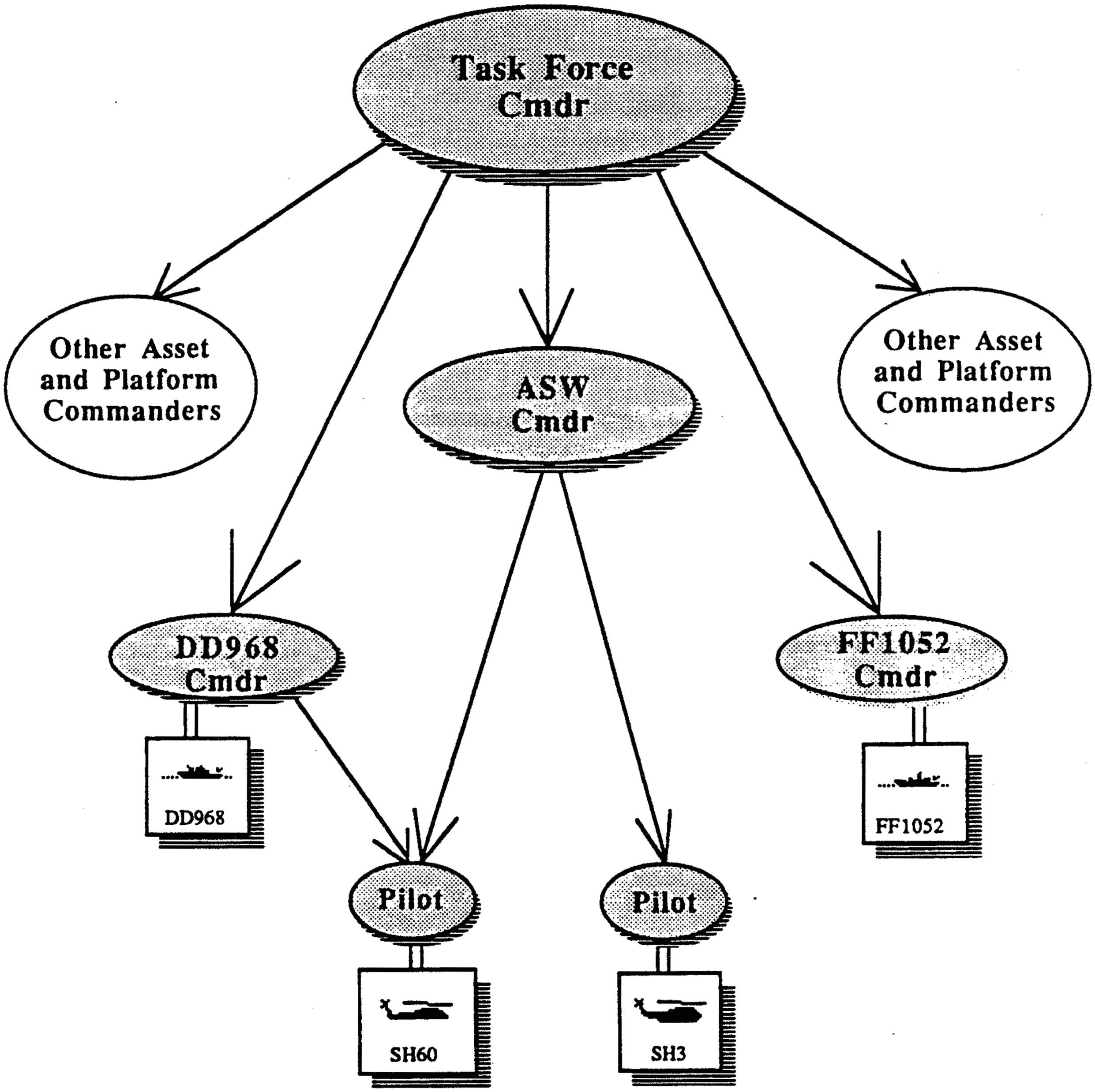
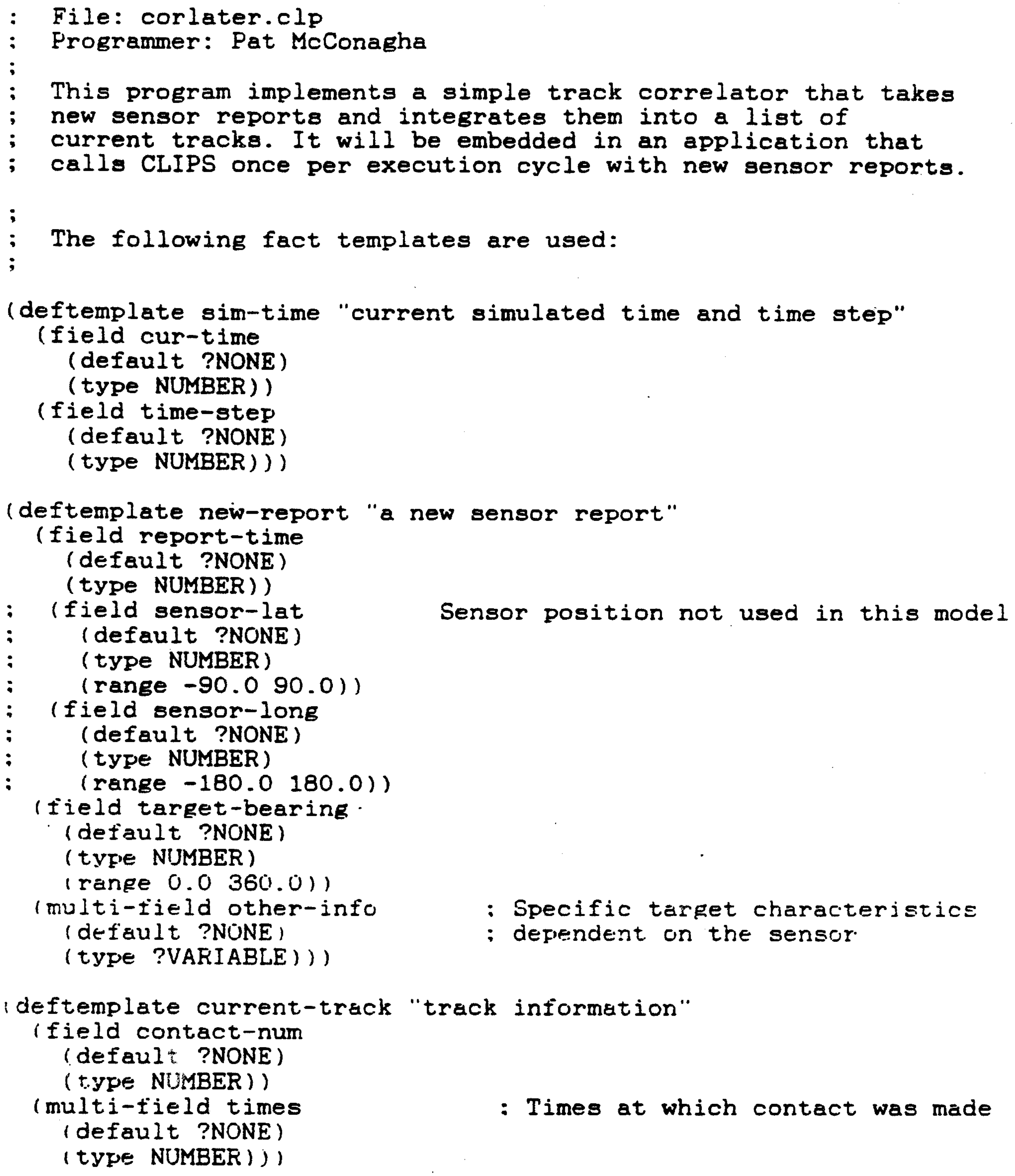


Figure 1. STE Configuration



Shaded objects could be modeled with an Expert System

Figure 2 - A Sample STE Scenario



```
:  File: corlater.clp
;  Programmer: Pat McConagha
;
;  This program implements a simple track correlator that takes
;  new sensor reports and integrates them into a list of
;  current tracks. It will be embedded in an application that
;  calls CLIPS once per execution cycle with new sensor reports.

;
;  The following fact templates are used:
;

(deftemplate sim-time "current simulated time and time step"
  (field cur-time
    (default ?NONE)
    (type NUMBER))
  (field time-step
    (default ?NONE)
    (type NUMBER)))

(deftemplate new-report "a new sensor report"
  (field report-time
    (default ?NONE)
    (type NUMBER))
;  (field sensor-lat          Sensor position not used in this model
;    (default ?NONE)
;    (type NUMBER)
;    (range -90.0 90.0))
;  (field sensor-long
;    (default ?NONE)
;    (type NUMBER)
;    (range -180.0 180.0))
  (field target-bearing
    (default ?NONE)
    (type NUMBER)
    (range 0.0 360.0))
  (multi-field other-info        : Specific target characteristics
    (default ?NONE)              : dependent on the sensor
    (type ?VARIABLE)))

(deftemplate current-track "track information"
  (field contact-num
    (default ?NONE)
    (type NUMBER))
  (multi-field times              : Times at which contact was made
    (default ?NONE)
    (type NUMBER)))
```

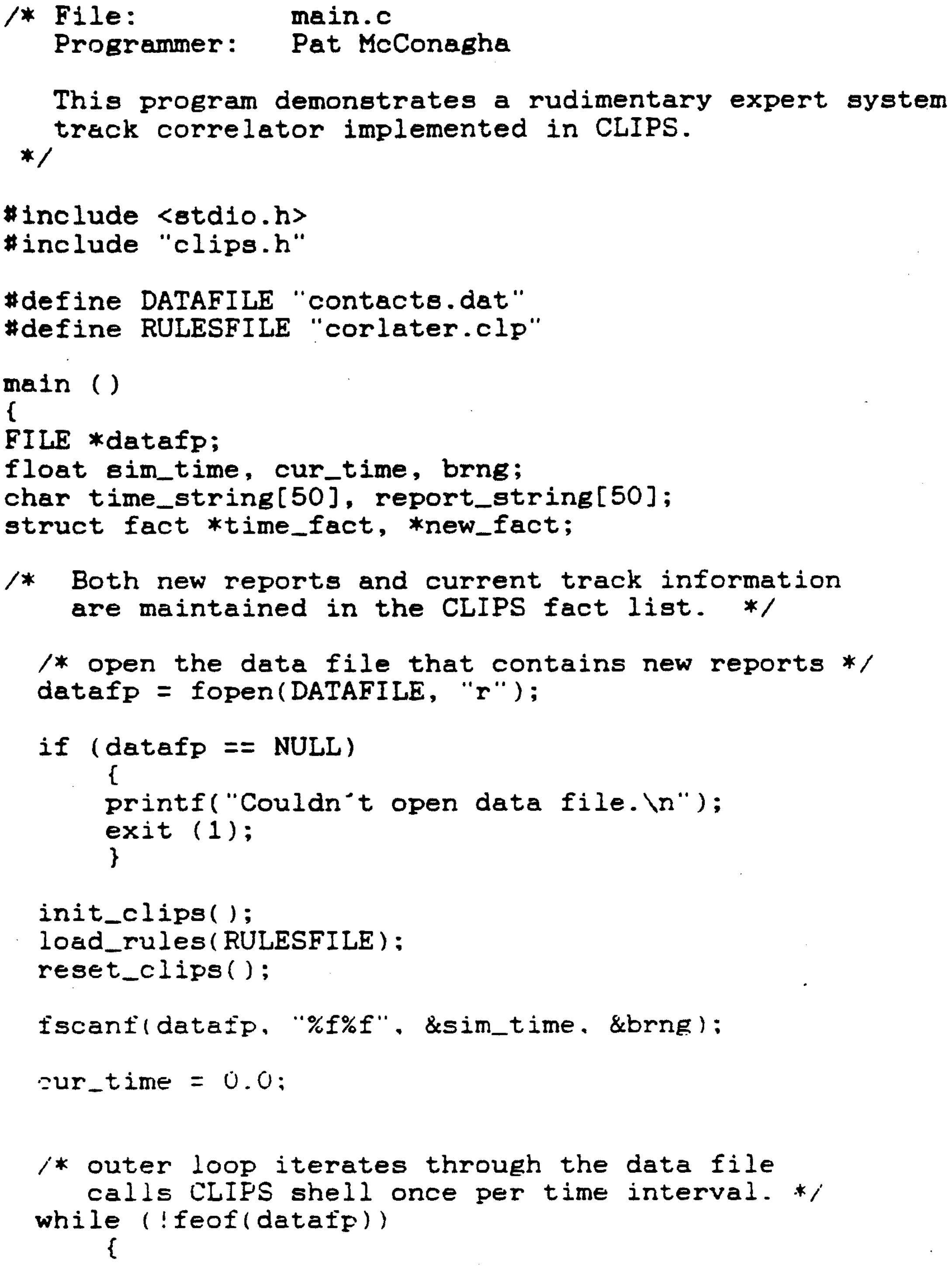
Figure 3 - An Expert System Track Correlator

```
(deftemplate contact "specific information from each contact"
  (field contact-num
    (default ?NONE)
    (type NUMBER))
  (field time
    (default ?NONE)
    (type NUMBER))
  (field target-bearing
    (default ?NONE)
    (type NUMBER)
    (range 0.0 360.0))
  (multi-field other-info          ; Specific target characteristics
    (default ?NONE)                ; dependent on the sensor
    (type ?VARIABLE)))

;
;  Initial facts
;
(deffacts initial-conditions
  (last-track-number 0))

;
; Define the rule for extending an existing track.
; A track is extended if bearings match between a new
; report and an established contact
;

(defrule extend-track
  ?report <- (new-report (report-time ?time)
                         (target-bearing ?bearing)
                         (other-info $?other))
  ?track <- (current-track (contact-num ?num)
                           (times ?last-time $?times))
  (contact (contact-num ?num)
           (time ?last-time)
           (target-bearing ?last-bearing))
  (test (same_target ?bearing ?last-bearing))    ; Simple test
=>                                               ; to match bearings
  (retract ?report)
  (same_track ?num ?bearing ?time)
  (modify ?track (times ?time ?last-time $?times))
  (assert (contact (contact-num ?num)
                   (time ?time)
                   (target-bearing ?bearing)
                   (other-info $?other))))
```

Figure 3 (Cont'd)

```
;
; Define rule for creating a new track
; A new track is created if a new report does not match the
; bearing of a known track
;

(defrule make-new-track
  ?report <- (new-report (report-time ?time)
                         (target-bearing ?bearing)
                         (other-info $?other))
  (not (contact (target-bearing ?old-bearing&:   ; No known contact
       (same_target ?old-bearing ?bearing))))    ; on new bearing
  ?num <- (last-track-number ?n)
=>
  (retract ?report ?num)
  (bind ?n (+ ?n 1))
  (new_track ?n ?bearing ?time)
  (assert (last-track-number ?n))
  (assert (current-track (contact-num ?n)
                         (times ?time)))
  (assert (contact (contact-num ?n)
                   (time ?time)
                   (target-bearing ?bearing)
                   (other-info $?other))))


;
; Define rule for droping a track
; Don't remove it from fact list, just report that it wasn't detectec
;   during this execution cycle

(defrule lost-track
  (declare (salience -50))
  (current-track (contact-num ?num)
                 (times ?last-time $?))
  (sim-time (cur-time ?t)
            (time-step ?delta-t))
  (test (<= ?last-time (- ?t ?delta-t)))
=>
  (no_contact ?num ?t))
```

Figure 3 (Cont'd)



```
/* File:          main.c
   Programmer:    Pat McConagha

   This program demonstrates a rudimentary expert system
   track correlator implemented in CLIPS.
 */

#include <stdio.h>
#include "clips.h"

#define DATAFILE "contacts.dat"
#define RULESFILE "corlater.clp"

main ()
{
FILE *datafp;
float sim_time, cur_time, brng;
char time_string[50], report_string[50];
struct fact *time_fact, *new_fact;

/*  Both new reports and current track information
    are maintained in the CLIPS fact list.   */

  /* open the data file that contains new reports */
  datafp = fopen(DATAFILE, "r");

  if (datafp == NULL)
      {
      printf("Couldn't open data file.\n");
      exit (1);
      }

  init_clips();
  load_rules(RULESFILE);
  reset_clips();

  fscanf(datafp, "%f%f", &sim_time, &brng);

  cur_time = 0.0;


  /* outer loop iterates through the data file
     calls CLIPS shell once per time interval. */
  while (!feof(datafp))
      {
```

Figure 4 - The 'C' Track Correlator Driver

```
/* build and assert the current time-keeping fact */
sprintf(time_string, "sim-time %f %f", sim_time,
                      sim_time - cur_time);
time_fact = assert(time_string);

cur_time = sim_time;

do
    {

    if (brng >= 0)   /* a negative bearing simulates */
                     /* no new reports during the    */
                     /* current execution cycle      */
        {
        /* build and add a new data fact */
        new_fact = get_el(3);

        add_element(new_fact, 1, WORD, "new-report", 0.0);
        add_element(new_fact, 2, NUMBER, NULL, sim_time);
        add_element(new_fact, 3, NUMBER, NULL, brng);

        if (add_fact(new_fact) == NULL)
             printf("Error adding a data fact.\n");
        }

    fscanf(datafp, "%f%f", &sim_time, &brng);
    }
while ((!feof(datafp)) && (sim_time == cur_time));


run(-1);

retract_fact(time_fact);

printf("\n");
}
```

Figure 4 (Cont'd)

```
/* define functions called from CLIPS */
usrfuncs()
{
int same_target(),
    same_track(),
    new_track(),
    no_contact();

define_function("same_target", 'i', same_target, "same_target");
define_function("same_track", 'v', same_track, "same_track");
define_function("new_track", 'v', new_track, "new_track");
define_function("no_contact", 'v', no_contact, "no_contact");
}


#define epsilon 1.0e-3
int same_target()
{
  float brng1, brng2;
  double fabs();

  brng1 = rfloat(1);
  brng2 = rfloat(2);

  if (fabs(brng1-brng2) < epsilon)
     return(TRUE);

  return(FALSE);
}


int same_track()
{

int con_num;
float brng, time;

con_num = rfloat(1);
brng = rfloat(2);
time = rfloat(3);

printf("New report for contact # %3d on
        "bearing %5.1f at time %5.1f \n",
        con_num, brng, time);
return(0);
}
```

Figure 4 (Cont'd)

```
int new_track()
{

int con_num;
float brng, time;

con_num = rfloat(1);
brng = rfloat(2);
time = rfloat(3);

printf("Starting new track for contact # %3d on "
       "bearing %5.1f at time %5.1f\n",
        con_num, brng, time);
return(0);
}


int no_contact()
{

int con_num;
float time;

con_num = rfloat(1);
time = rfloat(2);

printf("No report for contact # %3d at time %5.1f\n",
        con_num, time);
return(0);
}
```

Figure 4 (Cont'd)

Program Input

```
1 45
1 195
2 45
2 72
3 195
3 45
3 213
4 72
4 321
4 195
6 45
7 -1
8 72
8 213
```

Program Output

```
Starting new track for contact #   1 on bearing 195.0 at time   1.0
Starting new track for contact #   2 on bearing  45.0 at time   1.0

Starting new track for contact #   3 on bearing  72.0 at time   2.0
New report for contact #   2 on bearing  45.0 at time   2.0
No report for contact #   1 at time   2.0

Starting new track for contact #   4 on bearing 213.0 at time   3.0
New report for contact #   2 on bearing  45.0 at time   3.0
New report for contact #   1 on bearing 195.0 at time   3.0
No report for contact #   3 at time   3.0

New report for contact #   1 on bearing 195.0 at time   4.0
Starting new track for contact #   5 on bearing 321.0 at time   4.0
New report for contact #   3 on bearing  72.0 at time   4.0
No report for contact #   4 at time   4.0
No report for contact #   2 at time   4.0

New report for contact #   2 on bearing  45.0 at time   6.0
No report for contact #   4 at time   6.0
No report for contact #   1 at time   6.0
No report for contact #   5 at time   6.0
No report for contact #   3 at time   6.0

No report for contact #   4 at time   7.0
No report for contact #   1 at time   7.0
No report for contact #   5 at time   7.0
No report for contact #   3 at time   7.0
No report for contact #   2 at time   7.0

New report for contact #   4 on bearing 213.0 at time   8.0
New report for contact #   3 on bearing  72.0 at time   8.0
No report for contact #   1 at time   8.0
No report for contact #   5 at time   8.0
No report for contact #   2 at time   8.0
```

Figure 5 - Execution of a Sample Data File

REFERENCES

1. Cohen, Neil and J. Reynolds. 1990. "System Test Environment: A Real-Time, Man-In-The-Loop Fleet Simulator to Support Testing of Developmental Equipment." In *Proceedings of the SCS Multiconference on Object-Oriented Simulation* (San Diego, CA, Jan. 17-19). Society for Computer Simulation, San Diego, CA, 23-27.

2. Gorlen. Keith, *OOPS*, Public Domain Software Library, National Institutes of Health

omit

B14 Session:
Advisory Systems and Tutors

SPILC: An Expert Student Advisor

D. R. Read
Lamar University
Beaumont, Texas

1. Introduction

The Lamar University Computer Science Department serves about 350 undergraduate C.S. majors, and 70 graduate majors. B.S. degrees are offered in Computer Science and Computer and Information Science, and an M.S. degree is offered in Computer Science. In addition, the Computer Science Department plays a strong service role, offering approximately sixteen service course sections per long semester. The department has eight regular full-time faculty members, including the Department Chairman and the Undergraduate Advisor*, and from three to seven part-time faculty members.

Due to the small number of regular faculty members and the resulting very heavy teaching loads, undergraduate advising has become a difficult problem for the department. There is a one-week early registration period and a three-day regular registration period once each semester. The Undergraduate Advisor's regular teaching load of two classes, 6 - 8 semester hours, per semester, together with the large number of majors and small number of regular faculty, cause long queues and short tempers during these advising periods. The situation is aggravated by the fact that entering freshmen are rarely accompanied by adequate documentation containing the facts necessary for proper counselling. There has been no good method of obtaining necessary facts and documenting both the information provided by the student and the resulting advice offered by the counsellors.

Since the requirements for entering the C.S. program are fairly straightforward, and the first two semesters for entering students are reasonably uniform, an expert system that would advise the entering student as to an appropriate schedule appeared to provide the ideal solution to both the shortage in advising personnel, as well as the information gathering and documentation problems. This paper describes the development of such an expert system: SPILC (Student Prompter Involving Limited Communication) written using CLIPS.

*The author gratefully acknowledges the aid of Prof. S. Wiemers in providing valuable information regarding the interaction between the Undergraduate Advisor and C.S. majors.

2. Goals

The goals of this project were as follows:

1. To evaluate CLIPS for possible inclusion into the Lamar University computer science curriculum,
2. To develop a usable expert system for advising entering freshmen computer science majors,
3. To use the expert advisor as a prototype for a much larger and more sophisticated program for advising and tracking all computer science majors, from entry through graduation.

The evaluation of CLIPS as an expert system tool for use in the classroom had been intended in any case, and that fact, in addition to those features listed in 3., below, encouraged its selection for the expert advisor.

3. Choice of Hardware Platform and Language

Due to the availability of PCs for both development and application of the expert system, it was decided to implement the system for that environment.

Language choice was simplified by the fact that there were only two candidates. Among other factors, the following criteria were used in deciding which candidate to use for the expert advisor:

Backward chaining support,
Forward chaining support,
I/O capability,
Simplicity and ease of use,
Low cost,
Number of copies available,
Integrated editor.

CLIPS was chosen as the implementation language for this project due mainly to its apparent simplicity and consistency of syntax, the fact that forward chaining was considered to be sufficient for a simple rule-based system, and the department had access to as many copies as it needed for use during advising periods. Since CLIPS was also being considered for possible use in several upper level computer science courses, it was felt that this project would provide an ideal test to determine how easily and quickly it could be learned and used effectively.

4. Architecture of the Expert System

The model chosen was that of a small search space with reliable knowledge and fairly reliable data (1:89-126). While the domain knowledge is very reliable, data provided by the student, as indicated below, can be suspect. Both data and knowledge are reasonably constant over time, and computational resources were considered adequate for the problem.

4.1. Knowledge Acquisition

Expert knowledge was gained from three sources: (i) the Undergraduate Advisor for the Computer Science Department who, due to her very difficult schedule, was limited to a brief (three-page) written description of the typical questions, answers, and decisions that take place during the advising of an entering freshman; (ii) the author's several years experience in advising undergraduates and participating in curriculum development and modification; and (iii) the university undergraduate catalog.

4.2. Domain Knowledge

In order to major in computer science, a first semester student must have a combined score of at least 850 on the SAT (or equivalent ACT), or rank in the top one third of his/her graduating class. A student who has prior credit from another university or college must satisfy those requirements, as well as have an overall gradepoint average of at least 2.3 on all college-level work. After a student is accepted, a departmental "recommended program of study", a standardized degree plan, and the class schedule form the basis for scheduling advice.

The advisor must also consider university policy in such areas as: (i) maximum course load allowed, (ii) a requirement regarding continuous registration for freshman English until credit for six semester hours has been earned, and (iii) a requirement that a freshman must register for physical activity each long semester until he/she has completed four such courses.

Course prerequisite information must be available, as well as information regarding continually evolving general education requirements.

4.3. Student Specific Facts

During a consultation, a considerable amount of information must be collected from each student. Much of the time no official documentation of the information received from a student is available until well after the registration period has concluded. Often the documentation, when it arrives, is found to be in disagreement with the information supplied by the student during registration. A permanent record of the student-supplied information is desired for both advising purposes as well as for comparison against official documentation. This student-supplied information includes such items as: SAT scores; TASP scores; rank in graduating class; most advanced mathematics course taken successfully; computer science course (and language used) taken successfully; age of student; whether the student has a part-time (or full-time) job, and if so, how many hours per week it requires; and the number of semester hours the student desires to schedule. Some of this student information, such as TASP scores, the highest level mathematics course taken, or rank in class, are required only conditionally.

The decision was made to have the program include the student-supplied data in a hardcopy statement, similar to the following example, to be signed by the student:

```
                 SPILC   March 23, 1990

NAME:                         Able, Albert A
SSNUM:                        '555-55-5555'
SAT math score:               450
SAT verbal score:             450
1st semester at LU:           yes

Trigonometry or higher passed in HS:     yes
Passed a High School C.S. course:        yes


To the best of my knowledge, the above information is true.
I realize that if any of the above is found to be false, I
can be excluded from the Lamar University C.S. Department's
degree program.

SIGNED:___________________________

Recommended Courses:

     C.S. 1411
     Mth  1345
     Eng  131
     Hist 231
     pega 224
```

If the program determines that the student does not meet the requirements for entering the program, a similar form is printed, indicating the problem and suggesting appropriate action.

5. Design of the Program

A partitioning of the knowledge base was undertaken to simplify both development and debugging, as well as future extension of SPILC. The initial categories for partitioning the rule base were as follows:

1. Rules which controlled the input of permanent student record information, such as name, social security number, SAT scores, etc.;
2. Rules that controlled the input of student scheduling information, such as number of hours desired and number of hours the student works in a part-time job per week;
3. Scheduling rules, which included most of the domain knowledge for the problem;
4. Output rules for printing the acknowledgment of responsibility and the student's recommended schedule.

The facts were partitioned in a similar fashion, but were further subdivided into control facts, student record facts, or scheduling facts.

This partitioning, though convenient, was not necessary for a problem of this small magnitude. It was considered desirable, however, for the purpose of significant future development of the expert system.

6. Future Plans

The prototype is to be field tested during the registration period for the Fall 1990 semester. It will then be modified, as appropriate, to improve the interface and to correct any errors or deficiencies detected at that time. It will be extended to maintain degree plans and to enable the advising of all undergraduate computer science majors.

This significant extension will require that a database be created that will contain the essential facts obtained from each student during a consultation. The database must be updated during each consultation, and the facts must be in a suitable form for input to the expert advisor during subsequent advising sessions. Since a student who is enrolled at registration time can not be certain of his/her final grade in current classes, the database must contain a record of courses for which the student is currently enrolled. That information will be used to query the student as to anticipated grade for each of the courses in which he/she is enrolled. Regular updating of the database must occur after final grades are recorded in order to continue to enforce prerequisites and to maintain an accurate degree plan for each student.

In order to advise students in their second (or later) semesters, it will be necessary to create a file containing course and prerequisite information for all courses taught at Lamar. Both courses and prerequisites are subject to modification each year, so a significant and continuing maintenance effort will be required as the program remains in continued used.

7. Summary

CLIPS provides a very convenient development environment. The CLIPSWIN interface is quite easy to use, and all of the documentation is clear and precise. The primary weakness, from the author's point of view, is the limited I/O capability. The user interface and report generation are awkward to construct without such capabilities as positioning the cursor and sending carriage control characters to the printer.

The author had considerable previous experience programming in LISP and Prolog, and had experimented with Personal Consultant™ Plus, but had no prior experience with CLIPS. In preparation for this project, approximately four to six hours was devoted to reading the user's guide (2) and browsing through the reference manual (3) before attempting any programming. After writing a few very short examples, mainly checking the I/O features and some special functions such as "member" and "subset", it was felt that enough had been accomplished to begin the program. Expertise in constructing complex rules was developed very quickly.

CLIPS appears to be quite suitable for use in an introductory course on expert systems in which students have limited programming experience. One or two class periods, with examples chosen from the user's guide, should be sufficient to enable the students to begin writing their own programs. More advanced students can be given the user's guide and allowed to learn in a self-paced manner.

It is intended that the expert advisor, after field testing, will be expanded to aid in the advising of all computer science majors at Lamar University.

REFERENCES

1. Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., and Sacerdoti, E.: The Architecture of Expert Systems. In F. Hayes-Roth, D. Waterman, and D. Lenat (eds.) Building Expert Systems, Reading: Addison-Wesley, pp. 89-126, 1983.

2. Giarratano, J. C. CLIPS User's Guide, Lyndon B. Johnson Space Center, 1989.

3. Culbert, C. CLIPS Reference Manual, Lyndon B. Johnson Space Center, 1989.

4. Giarratano, J.C. and Riley, G. Expert Systems Principles and Programming, Boston: PWS-KENT Pub. Co., 1989.

5. Personal Consultant™ Plus (2539262-0001) Rev. C, Texas Instruments Incorporated, 1986.

6. Weiss, S. M. and C. A. Kulikowski, A Practical Guide to Designing Expert Systems, Rowman & Allanheld, 1984.

N96- 12957

PREDICTION OF SHIPBOARD ELECTROMAGNETIC INTERFERENCE (EMI) PROBLEMS USING ARTIFICIAL INTELLIGENCE (AI) TECHNOLOGY

S44-32
358908
P- 10

Written By:

DAVID J. SWANSON
Systems Exploration, Inc.
4241 Jutland Drive
San Diego, CA 92117

ABSTRACT

The electromagnetic interference prediction problem is characteristically ill-defined and complicated. Severe EMI problems are prevalent throughout the U.S. Navy, causing both expected and unexpected impacts on the operational performance of electronic combat systems onboard ships. This paper focuses on applying artificial intelligence (AI) technology to the prediction of ship related electromagnetic interference (EMI) problems.

INTRODUCTION

Electromagnetic interference, radio noise, and radio frequency interference all refer to the same condition. Most commonly referred to by the Navy as EMI, this condition inhibits, prevents, or distorts clear reception of an electromagnetic (EM) signal and degrades the overall performance of an electromagnetic system. The largest single consumer of the electromagnetic spectrum is the military. Modern military operations require that a large number of electromagnetic pieces of equipment be compatibly operated within a relatively small geographical area. The complexity of shipboard antennas, military radio frequency communications, and military combat EM systems is increasing far more rapidly than the improvements in EM design technology[1].

DEFINING THE PROBLEM

With the increased use and dependence on electromagnetic equipment, the accurate prediction of EMI has become a major tactical concern as well as a system design issue. More EM equipment is on U.S. Naval vessels today than ever before and most of it is considered critical to the vessel's success and survival in combat and routine day-to-day operations. While the U.S. Navy has received substantial benefit from the technological advancements, shipboard EM systems have become increasingly complex and vulnerable to EMI effects. Although shipboard EMI is not a new issue, the U.S. Navy is currently undergoing what the President of the U.S. Navy Board of Inspection and Survey called an "Electromagnetic Interference Pandemic"[2]. This means that every U.S. warship suffers from mild to severe electromagnetic interference that could threaten safety and decrease the ability of a ship to successfully complete its mission. The Navy has already witnessed several EMI induced disasters.

Three examples include:

* HMS SHEFFIELD. To avoid EMI to satellite communications, missile defenses were turned off resulting in the loss of this ship in 1984. Losses included over $200 million in damage and the death of many crew members.

* USS FORRESTAL. EMI triggered an aircraft rocket detonation on this aircraft carrier in the late 1960s. Losses included 134 crew members, 32 aircraft, and $172 million in damage to the carrier.

* NAVY CRUISER. A missile hit a friendly cruiser in the late 1960s due to electromagnetic interference. Losses included over $100 million in damage, the destruction of the topside of the ship, and the injury of many crew members[3].

In an effort to mitigate interference problems, the Navy has sponsored research and development to investigate various methods of solving the shipboard EMI prediction problem.

MATHEMATICAL MODELING

One standard approach to EMI prediction uses computationally intensive mathematical models. These mathematical models will produce reliable forecasts if the number of possible EMI sources and victims is small. Unfortunately, in U.S. warship communications and radar systems, the number of EMI sources is vast, varied, and constantly changing, making this mathematical approach cumbersome and impractical. An example that demonstrates the inefficiency of the mathematical model approach involves hull-generated intermodulation interference (IMI) signals. IMI signals are multiple transmissions that combine in a nonlinear fashion in and around the topside of a ship and reradiate as unwanted signals. A mathematical model is used to determine the interference frequency. The means for predicting when and which signals cause interference involves analyzing an overwhelming number of transmitter frequency combinations[4]. Due to the large number of frequencies that have to be considered, the testing process is labor-intensive, costly, and can take up to 24 hours to complete, although automated testing systems are being explored that are expected to reduce the overall testing time to about 6 hours[5].

It is frequently too costly, time-consuming and impractical to use these mathematical models in a rapidly changing tactical situation. In an effort to resolve EMI obstacles two alternatives are often employed.

CURRENT EMI SOLUTIONS

Two approaches have been relatively successful in containing and eliminating EMI. These approaches attempt to ensure EM equipment will function as designed without adversely affecting surrounding EM systems. The first approach relies on maintenance. Wait until an EMI problem occurs and then attempt to correct it. The second approach stresses prevention. Impose rigid design specifications on the system during the planning stages in an attempt to "over-engineer" or design-out all possible interference problems. Both of these approaches have been reasonably successful in reducing EMI in the past, but as additional EM equipment is installed aboard U.S. warships, these methods are not able to cope with the complexity and complications resulting from the presence of the large number of electromagnetic devices[6].

Once again, forecasting is possible, but only in an environment containing a small number of possible sources and victims of EMI. To meet the challenge of electromagnetic compatibility in an increasingly dense electromagnetic environment, the Navy is directing its attention to the application of AI technology to this problem.

AI AS AN ALTERNATIVE SOLUTION

Artificial intelligence technology has been widely successful in bringing ill-defined or combinatorially explosive problems into a tractable state[7,8]. AI technology differs from conventional programming technology in several ways.

One of the fundamental differences is AI techniques solve problems by manipulating symbols and symbolic relationships instead of performing standard mathematical computations. Another important distinction between AI techniques and conventional programming techniques is the use of heuristics instead of algorithms. Heuristics are useful principles or guidelines applicable in an area that may not be strictly defined.

Heuristics are typically used in areas that are resistant to mathematical approaches or algorithmic solutions[9]. The algorithmic approach will always produce the optimal solution but may take an unacceptable amount of time. The heuristic approach will generally produce an acceptable solution within a much shorter timeframe.

The most popular and effective way to express heuristics has been in the form of pattern/action decision rules, called "production rules"[10]. This methodology centers on the use of statements of the form IF condition THEN action. Production rules are a superior paradigm for use in describing situations or processes driven by changing data. Production rules can specify how the program should behave in the presence of changing information without detailed advance knowledge about the flow of control. Symbolic reasoning, heuristics, and the use of production rules are an appealing approach to problems that are resistant to mathematical approaches or algorithmic solutions such as the EMI prediction problem.

In late 1986, the Naval Ocean Systems Center (NOSC) San Diego, California, began exploring alternative approaches to EMI prediction. At that time, NOSC initiated the Adaptive Electromagnetic Control System (AEMCS) project. The focus of this effort was to develop a prototype decision aid that would forecast potential EMI problems on individual U.S. Navy destroyers. AI programming techniques and rapid-prototyping were the research and development approaches selected to explore both the problem and various partial solutions. The prototype itself was written in C and PROLOG programming languages and ran on IBM ATs. An EMI expert was consulted in the beginning of the AEMCS project to ascertain EMI prediction heuristics. Surveys were conducted on several ships to obtain information regarding the equipment and current EMI problems. The AEMCS prototype system required the operator to enter into an IBM AT computer the frequencies for all operating transmitters and receivers. Other EMI prediction factors, such as transmission power and transmitter location, were addressed implicitly within the production rules. Once the operator wanted an EMI forecast, facts about transmitters, receivers, and their respective frequencies would be asserted into the PROLOG EMI analysis system. If a production rule concluded there was a possible EMI conflict, then "a possible conflict fact" would be asserted into working memory and text concerning the problem would be sent to the terminal. If the operator wished to get further information on a potential conflict, the conflict would be selected and a description of the effect with possible resolutions would be displayed.

When the AEMCS system prototype was installed aboard the first ship, it was well received. Later, the AEMCS system was enhanced in response to suggestions from the users and was installed on several other ships.

EXPANSION OF THE AI APPLICATION

During 1989 NOSC initiated work on an EMI prediction system (EPS) prototype with a much larger scope than the AEMCS project. The focus of this effort was to better define the tactical EMI prediction problem and develop an embeddable prototype decision aid that would forecast potential ownship and ship-to-ship EMI. The project was to apply and expand the knowledge gained from the AEMCS project to the prediction of EMI problems within a preselected group of naval vessels. The EPS prototype was intended to be embedded within an electronic warfare command, control, and communication program, the Electronic Combat Module (ECM).

A number of different expert system development tools and languages were considered. The C Language Integrated Production System (CLIPS) was finally selected as the development tool for the project, using a SUN 4 as the development platform. CLIPS was selected because of its forward chaining inference method based on the Rete algorithm and its performance. It was expected that up to 150 EM devices might have to be considered at one time. Analyzing 150 devices was a formidable and computationally intensive problem and the expectation was that CLIPS would exhibit superior performance while analyzing a large number of different devices for potential EMI conflicts.

After CLIPS was selected as the development tool, a rudimentary knowledge base design was established. The design incorporated into the EPS prototype the heuristics for predicting historically known EMI problems among various ship classes. See Figure 1. The prediction of historical EMI problems were focused on since the problem forecasts could be verified and the historical information forecasts were the most useful to shipboard personnel. Ownship EMI problems were also concentrated on since these problems currently represent the most mission inhibiting collection of EMI problems. Heuristics for determining general receiver EMI, such as adjacent channel interference and odd-order IMI, were also incorporated.

```
(defrule SPS94-SSR13
"The broadband noise that is generated by RF transmissions
illuminating metal-to-metal contacts raises the ambient noise level
surrounding the ship throughout a wide spectrum of frequencies.
This reduces the signal-to-noise ratio of the incoming desired
signals resulting in reduced receiver sensitivity and loss of
signal reception."

  ;; If the SPS-94 radar and the SSR-13 receiver are
  ;; operating simultaneously on a Ticonderoga class
  ;; cruiser then assert the existence of a possible
  ;; EMI problem.

(?d1&:(eq ?d1 sps-94) ?
            ?ship1
            ?shipclass&:(eq ?shipclass cg-47)
            ? ? ? ? ? ? ? ?)

(?d2&:(eq ?d2 ssr-13) ?
            ?ship2&:(eq ?ship2 ?ship1)
            ?shipclass&:(eq ?shipclass cg-47)
            ? ? ? ? ? ? ? ?)
=>

  ;; Bind a pattern matching variable
  ;; and assert a possible EMI problem.

(bind ?gen (gensym))

(assert (emi SPS94-SSR13 ?gen ?d1 ?ship1 ?d2 ?ship1
        "Lost or reduced ssr-13 reception")))
```

Figure 1. Historical EMI Problem Rule.

After many design refinements, the current design of the EPS prototype encompasses historical EMI problems for most classes of surface ships. This design is similar to the AEMCS design in that the EMI forecasts concentrate on individual ships rather than ship-to-ship EMI problems.

The architecture of the initial EPS prototype was not complicated. A file containing a list of facts, or characteristics, about all transmitters and receivers operating on the various ships was created by the ECM program. A fact list is made up of the device name, device type, ship name, ship class, function, frequency in MHz, 3db-bandwidth, receiver bandpass, auxiliary received frequency, relative priority, power, and antenna gain. See Figure 2.

```
(DEVICE-1 TRANSCEIVER YORKTOWN CG-47 ECM 9000.0 15.0 10.0 0.0
HIGH 200.0 UNKNOWN)

(DEVICE-2 TRANSMITTER MERRILL DD-963 TACAN 286.5 2.0 1.0 316.7
MEDIUM 15.0 UNKNOWN)

(DEVICE-3 RECEIVER OBRIAN DD-963 COMMS 245.3 2.0 1.0 0.0
LOW 15.0 UNKNOWN)
```

Figure 2. Facts are Lists of Device Characteristics.

Upon execution, the EPS prototype asserts facts into working memory and the EPS is then run. Another file is created during execution that contains the resulting EMI problem forecasts. In this case the EMI forecasts are lists. The first element in Figure 3 is a pattern matching symbol, followed by rule name, conflict index, source device name, source ship, victim device name, victim ship, and effect.

```
(EMI URN54-SPS92 GEN1 URN-54 YORKTOWN SPS-92 YORKTOWN
"INTERFERENCE TO THE VIDEO OF THE SPS-92 RADAR")

(EMI HF-SPS5 GEN2 T2213 RAY SPS-5 RAY "SPOKING")
```

Figure 3. EMI Problem Forecasts are Represented as Lists.

The ECM takes this file with the EMI forecasts and displays them through the ECM's man machine interface (MMI). In some cases there are workarounds to the EMI problems and these can also be displayed through the MMI.

The current version of the EPS prototype is completely embedded within the ECM program. Files are no longer used to assert facts or capture EMI forecasts. The EPS system is controlled through a

C program that obtains the required information, asserts it into the system and takes the EMI forecasts and displays them through the MMI. As devices are shut off or frequencies are changed, the EPS system responds by creating a new fact containing the change and asserts it into the EPS facts list. Production rules retract old facts and EMI forecasts change when a frequency, power level, or ship distance changes.

Efforts currently focus on obtaining heuristics that relate to the function and priority of various shipboard devices. In a high-threat area, all shipboard self-defense systems are given the highest operating priority. Suppose a high powered high frequency (HF) communication transmitter interferes with a shipboard self-defense system. In the context of ship survival, tactics dictate securing the HF transmitter rather than the self-defense system, if no workaround is available. The result of incorporating these heuristics into the system is that the system has judgement concerning possible solutions to EMI problems.

Information about historical EMI problems is obtained from the Shipboard Electromagnetic Compatibility Improvement Program (SEMCIP). SEMCIP is at the forefront of efforts to correct Naval shipboard EMI problems. Most historical EMI problems concern simultaneous operation of multiple shipboard systems. In the SEMCIP database, which contains various problem descriptions, one of these systems is considered the source of the EMI and the other is the victim. Figure 4 translates this source-victim format into a production rule.

```
(defrule SPS94-HFRECEIVERS
"SEMCIP reference number 414-82. The transmissions from the SPS-94
radar can cause broadband noise (BBN) to be generated around the
topside of a Ticonderoga cruiser. This occurs when there is arcing
across loose metal-to-metal junctions due to illumination of the
junctions by transmissions from the SPS-94. This BBN raises the
ambient noise level surrounding the ship across a wide spectrum of
frequencies, reducing the signal-to-noise ratio of incoming signals
and consequently reduces the sensitivity of any HF receiver(s).
The solution is to eliminate the BBN by insulating, grounding, or
removing loose metal-to-metal junctions where induced RF energy has
caused arcing."

  ;; The following clause will be true if the SPS-94 is
  ;; operating on a Ticonderoga class cruiser.

(?dl&:(eq ?dl sps-94) ?
             ?shipl
             ?shipclass&:(eq ?shipclass cg-47)
             ? ? ? ? ? ? ? ?)
  ;; If there are High Frequency (3 - 30 MHz) receivers
  ;; operating on the same cruiser at the same time,

(?d2 ?type&:(eq ?type receiver)
             ?ship2&:(eq ?ship1 ?ship2)
             ?shipclass&:(eq ?shipclass cg-47)
             ?
             ?frequency&:(&& (<= ?frequency 30)
                             (> ?frequency 3))
             ? ? ? ? ? ?)

  ;; then assume a possible EMI problem exists
  ;; with the source of the EMI being the SPS-94
  ;; and the victims being any HF receivers.

=>

(bind ?gen (gensym))
(assert (emi sps94-hfreceivers
             ?gen ?d1 ?ship1
             ?d2 ?ship1
             "Possible mild to severe EMI/IMI to HF receivers"
             )))
```

Figure 4. Source-Victim Production Rule.

The prototype EMI prediction system has over 100 production rules, most of which describe severe historical EMI problems. The prototype can analyze 75-100 transmitters and receivers within a matter of minutes, using a SUN 4 under UNIX. Shipboard testing is scheduled to begin in the Fall of 1990. The system will be used by shipboard electronic warfare commanders.

CONCLUSION

Over the last 40 years, the U.S. Navy has become increasingly dependent upon systems that exploit the electromagnetic environment. Electromagnetic technology has evolved from vacuum tube technology in the 1950s to very large scale integration technology in the 1990s. More capable and sophisticated shipboard communication equipment, radars, and other sensors have evolved. As a result, shipboard EMI has become a severe problem. The traditional approaches to EMI prediction and the achievement of system electromagnetic compatibility are impractical for shipboard use and are frequently too costly and time-consuming to use in tactical or day-to-day operational situations. In an effort to create a low-cost, effective EMI prediction system, alternative approaches are being explored using AI technology. AI technology is currently being applied successfully to portions of the shipboard EMI prediction problem. These research efforts have resulted in better Naval shipboard frequency management and are serving in the continued effort to mitigate shipboard EM interference conflicts.

REFERENCES

[1] Li, S., Logan, J., and Rockway, J., "Ship EM Design Technology," Naval Engineers Journal, May 1988, p.154.

[2] Orem, J., "The Impact of Electromagnetic Engineering on Warship Design," Naval Engineers Journal, May 1987, p.210.

[3] Grich, R. and Bruninga, R., "Electromagnetic Environment Engineering - A Solution to the EMI Pandemic," Naval Engineers Journal, May 1987, p.202.

[4] Maia, P. and Smith, J., "A Method for Predicting Intermodulation Product Levels," IEEE 1985 National Symposium on Electromagnetic Compatibility, June 1985, p.408

[5] Mieth, W., "A Cost-Effective Solution to Measurement of Hull-Generated Intermodulation Interference on U.S. Navy Ships," IEEE 1989 National Symposium on Electromagnetic Compatibility, May 1989, p.186

[6] Duff, W. and White, D., "A Handbook Series in Electromagnetic Interference and Compatibility," Vol. 5, Germantown, Maryland: Don White Consultants, 1972.

[7] Waterman, D., "A Guide to Expert Systems," Reading, Massachusetts: Addison-Wesley, 1986.

[8] Calabough, J., "Software Configuration - An NP Complete Problem," ACM Special Interest Group on Business Data Processing, Computer Personnel Research Proceedings, Conference on Expert Systems in Business, March 1987.

[9] Kunz, J., Kehler, T., and Williams, D., "Applications Development Using a Hybrid AI Development System," The AI Magazine, Fall 1984, p.53.

[10] Fikes, R. and Kehler, T., "The Role of Frame-Based Representation in Reasoning," IntelliCorp Technical Article from the special issue of Communications of the ACM on Knowledge-based Systems, September 1885, p.4.

Building an Intelligent Tutoring System for Procedural Domains

By Andrew Warinner, Diann Barbee, Larry Brandt, Tom Chen, and John Maguire
Global Information Systems Technology, Inc.
1800 Woodfield Drive,
Savoy, Illinois 61874

Introduction

Jobs that require complex skills that are too expensive or dangerous to develop often use simulators in training. The strength of a simulator is its ability to mimic the "real world", allowing students to explore and experiment. A good simulation helps the student develop a "mental model" of the real world. The closer the simulation is to "real life", the less difficulties there are transferring skills and mental models developed on the simulator to the real job. As graphics workstations increase in power and become more affordable they become attractive candidates for developing computer-based simulations for use in training. Computer-based simulations can make training more interesting and accessible to the student.

Unfortunately, good simulations do not necessarily make good trainers. One of the main tenets of most current learning theory is that the development of new knowledge is greatly constrained by what an individual already knows[1]. Simulations may require complex skills that are difficult to develop individually in sophisticated simulation. The student may not be able to use the simulation until the prerequisite knowledge and skills have been learned. Computer simulations are more flexible than dedicated, "task specific" simulations since they can simulate situations that are not strictly "realistic" but can reduce the complexity of the simulation in order to develop basic skills and concepts.

Although a simulation is a learning environment, it offers the learner no instructional assistance. We believe learning is greatly enhanced when instructional techniques are added to a simulation. For the past three years we have been exploring the challenges of incorporating "intelligent tutoring systems" (ITS) into computer-based simulations. Developing an intelligent tutoring system for a simulation really requires the development of two cooperating expert systems: a *domain* expert system that serves as a basis for evaluation of the student and an *instructional* expert system that can compare the student to the domain expert and prescribe training.

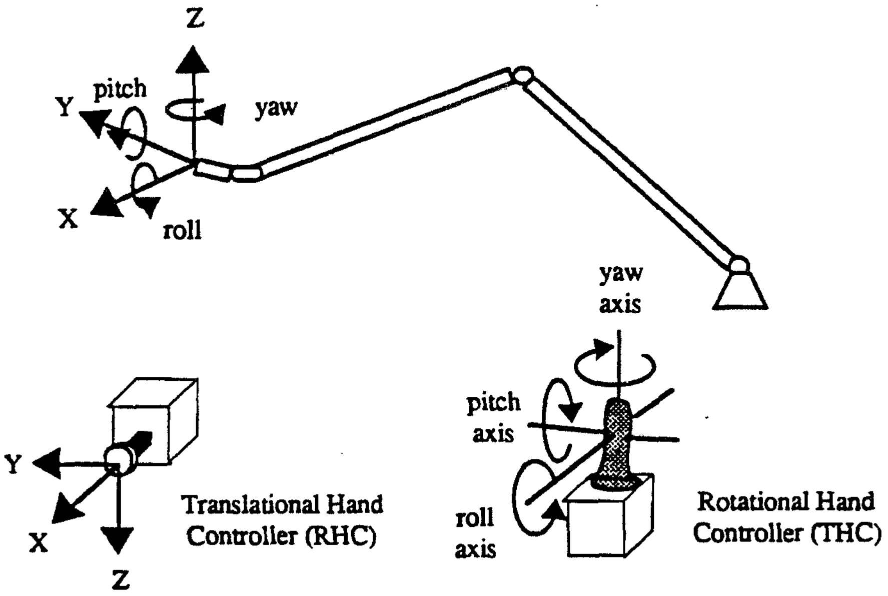


Figure 1. Controlling the RMS is Orbiter Unloaded Mode

The Domain: The RMS

The Remote Manipulator System (RMS) is the mechanical arm of the Payload Deployment and Retrieval System (PDRS) of the Shuttle. It is used to grapple a payload stowed in the Shuttle's cargo bay and lift it into orbit or grapple a payload in orbit and berth it in the cargo bay. Like a human arm, the RMS has three joints, a shoulder, elbow, and wrist, each with varying degrees of freedom (possible directions of movement). The arm is attached at the shoulder to the longeron of the Shuttle bay and is over fifty feet in length. At the end of the RMS is "end effector". The end effector used to grasp and hold the payload. Like a human arm, the RMS has physical limits on the roll, pitch and yaw of each joint. The RMS has movement limits imposed by a computer that monitors the RMS to reduce the possibility of damage. The RMS can be moved into positions where it loses one of its degrees of freedom (i.e. when movement of a joint in a specific direction becomes impossible). These configurations are called "singularities". The operator must reposition the RMS when it is in a singularity to regain its freedom of movement.

The RMS operator controls the the arm from the rear of the Shuttle cockpit. It is controlled with two hand controllers: a "translation hand controller" (THC) and a "rotational hand controller" (RHC). The operator can view the payload and RMS from windows or on a closed circuit TV (CCTV) from several cameras positioned about the Shuttle.

The RMS has several modes of operation. The RMS can be entirely controlled by the Shuttle's general purpose computer (GPC). The GPC can assist the shuttle operator in operating the arm or the operator can control the arm without computer assistance. These different modes of operation use different coordinate systems to describe the position of the RMS, the Shuttle, and the payload. The different modes also change the effects of the hand controllers on the position of the RMS (see Figure 1).

Successful operation of the RMS requires motor skills, complex cognitive skills, and knowledge of the mechanics of the RMS. To master the RMS the operator must learn the limits of the RMS and how to control its different modes. An understanding of the different coordinate systems and the ability to visualize arm and payload movements in space relative to the Shuttle are also important for successful RMS control. Operators must learn to manipulate the arm efficiently and safely.

Over the past three years, NASA has developed a computer-based simulation of the RMS called the Prototype Part Task Trainer (P2T2). Running on a color graphics workstation, P2T2 simulates the RMS and its different modes of operation using the same algorithms as the GPC. P2T2 simulates the different camera views available from the CCTV as well as the RMS control panels. P2T2's hand controllers are exact replicas of the THC and the RHC on the Shuttle.

Our goal is to embed an intelligent tutoring system into P2T2 to make it a more effective training device. The ITS/P2T2 will be a stand-alone trainer capable of teaching the domain of RMS operation. We will use CLIPS as the inference engine of of the ITS. CLIPS has several advantages over other inference engines:

- ability to be embedded in other applications, P2T2 in our case

- CLIPS is written in C and runs under UNIX®, P2T2 is written in C and runs under a variant of UNIX

- source code is provided, allowing us to make special modifications

Since we must build our ITS into P2T2, CLIPS ability to be embedded is important. Performance is another critical concern. Our ITS needs real-time performance in order to monitor and instruct the student.

Intelligent Tutoring Systems

The primary difference between intelligent tutoring systems and more traditional computer-based training is the "student model", a representation of the skills and knowledge possessed by the student. An intelligent tutoring system contains a student model, a computer-based training lesson does not.

The instructional expert uses the student model to gauge the student's progress and prescribe instruction. The domain expert compares the student to the "correct" performance it generates and provides the results to the student model. Since the student model is the used by both the instructional expert and the domain expert, the student model must have a representation that is accessible to both experts. Figure 2 illustrates how the two expert systems in the ITS act on the student model.

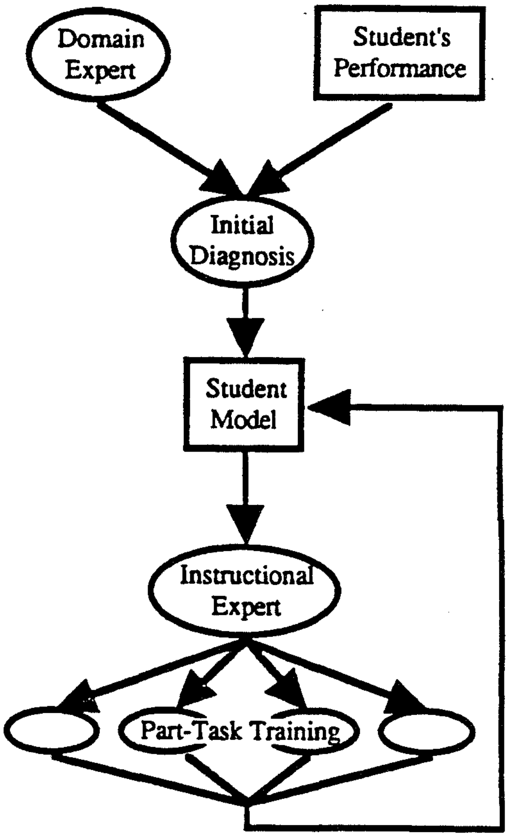


Figure 2. The Student Model, the Domain Expert and the Instructional Expert

We have chosen to represent the student model as a hierarchical network of skills and concepts necessary to master the RMS. Each skill or concept can have supporting subskills and subconcepts. A subskill or subconcept may support several skills or concepts. We

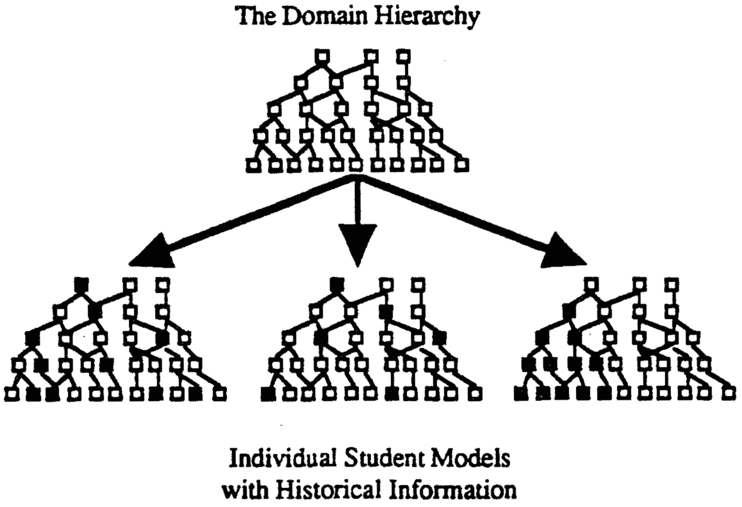


Figure 3. The Domain Hierarchy and the Student Models

call this taxonomy of the RMS domain the "domain hierarchy". Each skill or concept is represented by a node in the domain hierarchy. The student model is a copy of the domain hierarchy that stores information about the student's mastery or misuse of each skill or concept. Figure 3 illustrates the domain hierarchy and the student model.

The domain hierarchy/student model is a good representation for both diagnosis and instruction. Part-task training can use the hierarchical taxonomy of the domain to organize instruction. Diagnostically, the student model functions as a decision tree to which we apply algorithms drawn from electronic fault isolation. The diagnostic and instructional functions of the student model will be explained in more detail.

The Domain Expert

The domain expert provides the means to analyze the student. It must "understand" its domain. The domain expert must solve problems as an expert as well as be able to understand the student's actions and compare them to its solution. We have found the best representation for the domain expert is the "procedural network". Procedural networks have been used before in intelligent tutoring systems, for example, the "BUGGY" ITS developed by Brown and Burton[2]. The procedural network is a powerful representation of how the skills and tasks of the domain are related. Procedural networks are a good representation for an ITS that tutors a procedural or task-oriented domain[3]. Briefly, the advantages of a procedural network are:

- goal-based representation of the task or procedure allows for a flexible evaluation of student performance

- real time evaluation of the procedure

- multiple levels of abstraction in the procedure

- mechanisms for representing the partial ordering of procedures

- a representation of the "world" as it relates to the procedure

Procedural networks can be constructed dynamically. Our ITS will not dynamically construct its procedural network for two reasons. First, we have chosen to restrict knowledge acquisition to a small set of tasks in the RMS domain. Second, the dynamic construction of procedural networks uses difficult techniques such as plan criticism and plan optimization. Dynamically constructed procedural networks might contain flaws that would limit their usefulness during student evaluation. Our architecture does not preclude the dynamic construction of procedural networks if they are needed in the future.

Hierarchical Reasoning in Procedural Networks

The hierarchical nature of procedural nets makes them ideal for reasoning about the procedure at different levels. Student diagnosis can measure skills and performance at different levels of the procedure. For example, we might want to measure an overall quantity like the time to perform a section of the procedure. The hierarchical nature of the procedural network allows us to measure skills at different levels in the procedural network without examining and interpreting the individual actions that accomplish that section of the procedure.

Flexible Framework for Plan Recognition

One of the most difficult tasks in procedure evaluation is understanding the student's progress through the procedure. Often procedures contain some flexibility in the order of steps or tasks performed. Procedures can offer opportunities for the student to correct his or her mistakes and continue. A step-by-step comparison of the student and the expert's solution is too rigorous. If the procedure contains tasks that can be performed in any order, we can't rely on a step-by-step ordering of the student's actions for evaluation. The procedural network's orsplit nodes are a good representation for such flexible plans. The procedural network's andsplit nodes can represent the strict ordering of procedure steps.

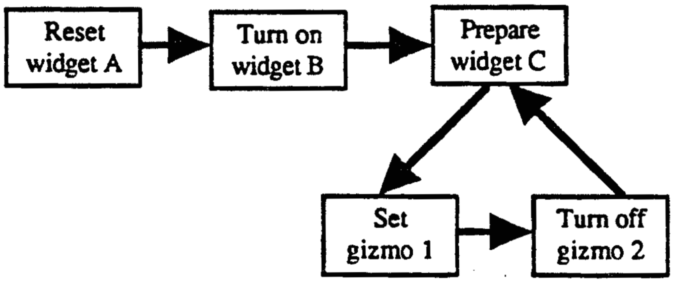


Figure 4. A procedural network example

For example, suppose a section of the procedural network contains this procedure of independent tasks:

1. Reset the widget A (press button 1)
2. Turn on widget B (turn knob 1 to "on")
3. Prepare widget C (accomplished by subprocedure)
   3.1. Set gizmo 1 (turn knob 2 to "5")
   3.2. Turn off gizmo 2 (turn switch 1 to "off")

Suppose that the widgets and gizmos are independent mechanisms: manipulating one widget does not affect the operation of any of the others. If the procedure was executed in strict sequence it would result in the following sequence of actions:

1. press button 1
2. turn knob 1 to on
3. turn knob 2 to 5
4. set switch 1 to off

But suppose the student executes the actions in this order:

1. set switch 1 to off
2. press button 1
3. turn knob 2 to 5
4. turn knob 1 to on

The procedural network can interpret this sequence of actions as accomplishing the procedure even though the actions are not in strict order. See Figure 4 for an illustration of this procedure.

Procedural Networks and Real-Time Evaluation

Another advantage in using procedural networks as a representation is that they can be used to evaluate the procedure as it is performed. Real-time evaluation is needed if some kind of coaching feedback is provided to the student. Student evaluation becomes a process of parsing the student's actions and comparing them to the procedural network. This can be done in a top-down fashion to the necessary level of detail. The state of the procedural network at any point in time is a complete description of the state of the world as well as the state of the procedure. As the student moves through the procedure, the interpretation of his or her actions is based on how they changed the state of the world.

Representing the World State

As mentioned above, the procedural network is not only a representation that describes the procedure but also the state of the world. The procedural network describes how each step of the procedure affects the state of the world. This description of the procedure allows reasoning by the modules of the ITS on the effects and relationship of parts of the procedural network.

Procedural Networks

Procedural networks were first characterized by Sacerdoti[4]. They are closely related to augmented transition networks and generalized and-or graphs. The procedural network is ordered by its links to among nodes. Nodes may have predecessors, successors, a parent and children. The successor and predecessor links order the procedure. The parent and child links denote subprocedures that must be executed to achieve the effects of the parent procedure. The parent and child links allow the procedural network to be ordered hierarchically. The procedural network is composed of four basic classes of nodes described below.

Procedure start and procedure end nodes are delimiters of the procedure. They are used for both procedures and subprocedures.

Goal nodes and subprocedures organize the procedural nets hierarchically. Goal nodes are accomplished by subprocedures that are linked as children. In the example illustrated in Figure 4 the node "Prep widget C" is a goal node that is accomplished by the subprocedure "Set gizmo 1" and "Gizmo 2 off" (note: the procedure start and procedure end nodes have been eliminated from the figure).

Andsplit and andjoin nodes delimit a collections of steps which may be performed independently. The andsplit and andjoin nodes themselves delimit the independent steps.

Orsplit and orjoin nodes are similar to the andsplit/andjoin nodes. They delimit a set of steps only one of which must be performed successfully. The orsplit and orjoin nodes delimit the steps.

Node effects are lists of effects on the world state. They represent the changes caused by completing the procedure step. The node effects will change machine values, positions, and other world state information.

Link predicates are used to control branches of the procedural network based on the state of the world. Since the procedural network is not constructed dynamically, link predicates enable or disable branches of the procedural network. For example, a branch for an error correction procedure may be enabled or disabled depending on the state of the equipment.

Procedural ordering links are used to represent ordering information not captured by the successor and predecessor links. The procedural ordering links are used to express ordering of the procedure not required by the machine states. The procedural ordering information is kept separate from the world state representation.

Comparing the Expert and the Student

As the student performs the procedure, the domain expert monitors his or her progress with the procedural network. When the student has finished the procedure or the instructional expert has intervened, the domain expert can refer to the procedural network to see what portions of the procedure were completed correctly and what portions were not completed correctly. The domain expert must now assess the causes of the student error and update the appropriate skills and concepts in the student model.

The domain expert must now translate the results of the procedural network into information about specific skills and concepts in the student model. The procedural network lends itself to a classification of student errors[5]. This classification uses the structural and world state information represented in the procedural network. Student errors fall into four classes:

- problem violations
- irrelevant procedures
- incorrect procedures
- ordering violations

Invalid action - This is an action that the student has taken that is not valid anywhere in the procedural network. Since the procedural network characterizes all possible paths through the network and all the possible actions that might be taken somewhere in the procedure, the domain expert can detect any action that does not fall on a path.

Problem violation - A student may take actions that are appropriate to achieve a goal but are inappropriate for the initial state of the world.

For example, suppose we have a procedure:

```
1. Power up the widget (goal)

(if widget is type A)
    1.1 Set power switch to "on"
    1.2 Press widget reset button

(if widget is type B)
    1.3 Set widget dial to "0"
    1.4 Set power switch to "start"
    1.5 Press widget reset button
    1.6 Set power switch to "on"
```

If we told the student that the widget is type A and he performs any of the steps 1.3 - 1.5 he has made a "problem violation".

Irrelevant procedure - Since we are not dynamically constructing the procedural network, we will use link predicates to disable unnecessary parts of the procedural network. Domain expert can detect if the student attempts to execute these disabled branches and report them as "irrelevant plans".

For example, suppose we have the following procedure:

```
1. Prepare gizmo (goal)

(if gizmo status is "error")
    1.1 Set gizmo power button to "off"
    1.2 Set gizmo power button to "on"
    1.3 Press gizmo reset button

(if gizmo status is "ok")
    1.4 Press gizmo button 1
    1.5 Press gizmo button 2
    1.6 Set gizmo switch to "on"
```

The "if" statements represent link predicates that enable or disable branches in the procedural network. If the student attempts to perform the steps of the error subprocedure, the domain expert will recognize them as "irrelevant plans".

Incorrect procedure - If the student omits a step in a procedure, the domain expert can detect this as an unsatisfied node in the procedural network. Domain expert will classify the missed step as an "incorrect procedure".

Ordering violation - We have added the procedural ordering links to the procedural network to represent ordering of the procedure not required by the world state. Domain expert will use these procedural ordering links to detect violations in the ordering of actions that are not mandated by node effects.

For example, suppose we have the following procedure:

1. Prepare the widget (goal)
   1.1 Set switch to "A"
   1.2 Press button 1
   1.3 Turn dial to "5"

Suppose the switch, button, and dial are independent of each other; the operation of one does not affect the others. This would be represented in the procedural network as an andsplit/andjoin branch. We can add procedural ordering links to represent the fact that we want the steps 1.1, 1.2, and 1.3 performed in strict order. Suppose the student performed the actions in this order:

1. Pressed button 1
2. Turned dial to "5"
3. Set switch to "A"

Domain expert can diagnose this as a ordering violation error but it will not classify it as an incorrect procedure error since the student has not violated the andsplit/andjoin construct in the procedural network.

Error Evaluation

After the domain expert has classified the errors observed in the procedural network, those errors must be mapped to corresponding skills and concepts in the student model. Each step in the procedural network has pointers to skills and concepts necessary to successfully perform that step in the procedure. As we noted before, the error classifications can help interpret the mistakes observed in the procedural network. In addition, we can use the historical information in the student model to assist in the diagnosis. Each step in the procedural network has links to several skills and concepts in the domain hierarchy:

- "knowledge" nodes that represent that the student is aware of the procedure step

- "condition" nodes that represent the student's knowledge of the conditions when the procedure step should be performed

- "skill" nodes that represent the satisfactory performance of the procedure step

- "effects" nodes that represent the effect of the procedure step on the state of the world

For example, suppose the domain expert detected a "problem violation" error. There are several plausible explanations for this error:

- the student is not aware of the conditions under which the procedure step should be performed

- it was a transient error; the student ignored or misinterpreted the conditions

- the student is ignorant of the effects of the procedure step on the state of the world

Each of these plausible explanations are represented by nodes in the domain hierarchy. To some extent the explanations are mutually exclusive. How does the domain expert choose between them? The domain expert can use the historical information from the student model. Continuing our example, suppose the student model shows that the student has never been exposed to concepts that represent "knowledge" of the procedure step, the domain expert can rule out the possibility that it was a transient error. On the other hand, if the student model shows that the student was familiar with the procedure step but has not used the procedure step in some time. The domain expert will favor the explanation that it was a transient error.

We have found that an analysis of the procedural network can provide information about only a subset of the skills and concepts in the student model. The domain expert can only infer information by observing the student. But the student model contains high-level abstractions and low-level skills and whose use cannot be observed directly. For example, a high level concept like "safety" cannot be associated with a single procedure step. An abstract concept "knowledge of RMS coordinate systems" would be difficult to deduce from simply observing the student. In general, the domain expert is able to draw conclusions about intermediate skills and concepts in the student model[6].

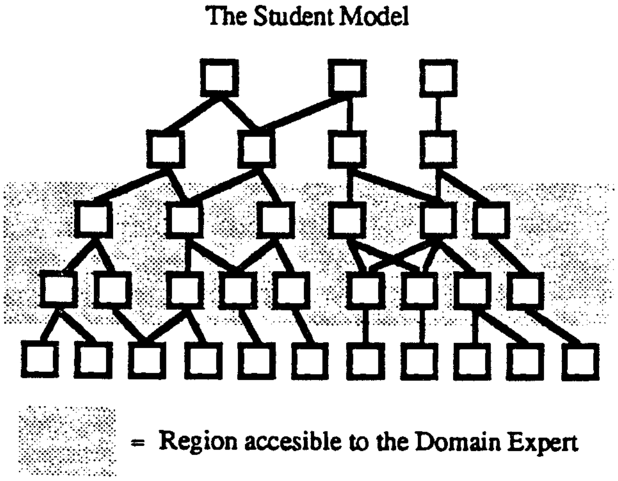


Figure 5. Regions available to the Domain Expert's diagnosis

The Instructional Expert

So far we have discussed the diagnostic aspects of an intelligent tutoring system. The diagnostic functionality is only half of an ITS, the other half is its tutoring functionality. An ITS can be viewed as a expert system compares the "expert model" of the domain to the "student model" that represents a novice student. The ITS then determines "operations" that will transform the student model to match the expert model.

Once the domain expert has updated the student model based on the result of its diagnosis, the instructional expert takes over. The instructional expert must examine the state of the student and apply remediation to the weaknesses it finds there.

We have chosen to provide tutoring to the student by means of part task training. Part task training is based on a systematic analysis of the instructional domain. The analysis identifies the skills, strategies, and knowledge necessary for expert performance. It also identifies the hierarchical relationships among the skills and knowledge. As an example of this, Figure 6 illustrates a part-task analysis of the RMS domain. The skill "Payload Deployment" is composed of the skills "Payload Release", "Payload Unberthing", "Move to Grapple Position", "Grappling the Payload", and "Ungrappling the Payload". The skill "Payload Deployment" is an "integration" skill. It requires mastery or "integration" of some subskills. The subskills may themselves be decomposed into other skills.

Once an analysis of domain is complete, training is designed to develop proficiency in the skills and concepts found in the domain hierarchy. A part task is designed to teach exactly that skill or strategy. When the student is proficient in all the subskills of an integration skill then the student can be trained in the integration skill[7].

Both the diagnostic functionality and the instructional functionality can exploit the hierarchical organization of the expert domain. The hierarchical organization can be

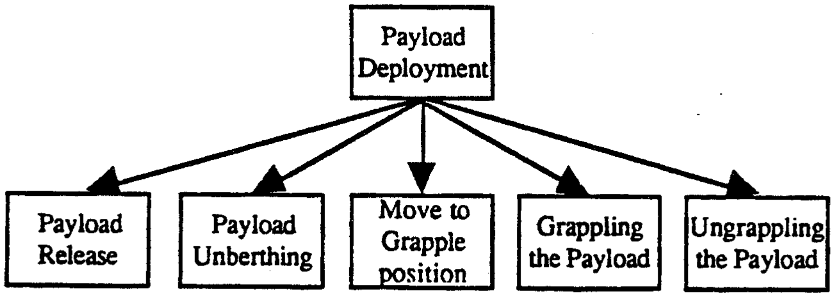


Figure 6. Sample Part-Task Analysis of the RMS Domain

used as a sort of "decision tree" using the historical information in the student model. The hierarchical structure of the student model is used to organize and relate the skills and knowledge of the domain for the instructional expert.

Final Diagnosis and Instruction

At any given time, the student may have some misconceptions or lack proficiency in skills in the domain. How does the instructional expert decide when and what can be tutored? The question of when the student should be tutored can be answered by the historical information stored in the student model. We have adapted algorithms from electronic circuit fault diagnosis to answer the questions of what should be tutored and when it should be tutored.

The fault isolation algorithms utilize the hierarchical structure of the student model/domain hierarchy. There is a certain amount of "overlap" in the hierarchical structure of the student model. Some subskills are required by several skills. The fault isolation algorithms can use these interrelationships to help us distinguish the "source" of error from its "symptoms" in the student model. Suppose we know that skill A^1 and A^2 have the subskills B^1, C^1, and C^2 in common and the domain expert has determined that student has misused them (see Figure 7). The problem may be in the skills A^1 and A^2 or in their supporting subskills. Our fault isolation algorithms attempt to explain the deficiencies by looking for areas that the deficiencies have in common. These common areas might be the real cause of the deficiencies. In our example, the fault isolation algorithm would consider the skills B^1, C^1, and C^2 as the real source of the student's misconceptions and the skills A^1 and A^2 as symptoms. The fault isolation algorithms attempts to find the simplest explanation that accounts for the most errors in the student model. Furthermore, they can recommend a skill to be tested that will eliminate the most uncertainty about where the real source of error lies in the student model.

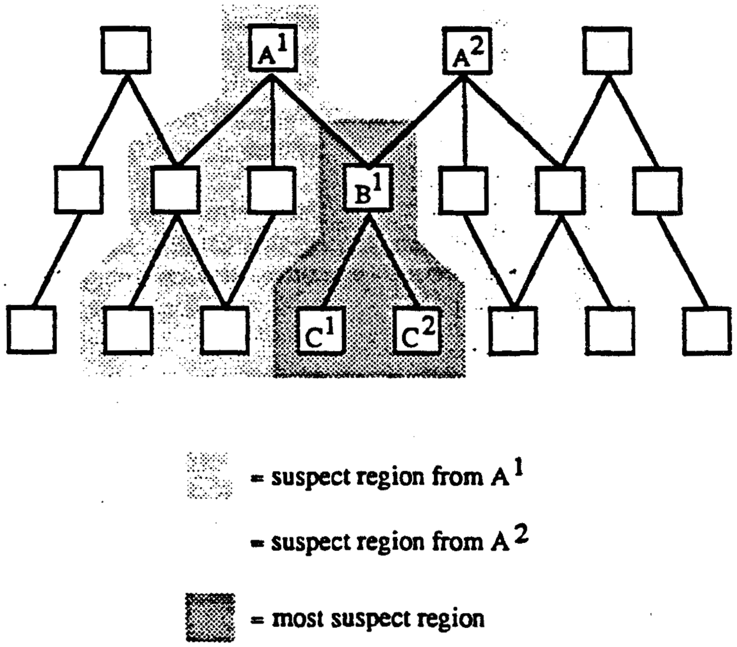


Figure 7. Applying fault isolation techniques to the Student Model

The fault isolation algorithms provide:

- a skill or concept that it has isolated as the source of the student's misunderstanding

- or a region in the domain hierarchy where errors are located and a specific skill or concept that is the mostly likely source of error

The instructional expert must now determine which part-task training will remedy the deficiencies observed in the student. One of the functions of the domain hierarchy is to serve as a map to the part-tasks. Given a set of skills and concepts misused by the student, the instructional expert can find a set of part tasks that will instruct the student.

The instructional expert must organize the part-task training it presents to the student. The instructional expert uses the structure of the domain to sequence the presentation of part-task training. For example, suppose the instructional expert must teach a region of the domain hierarchy as in Figure 7. The instructional expert has determined that it must teach the skills B^1, C^1, and C^2. Our part-task training philosophy dictates that subskills should be trained before the skills they support. The instructional expert then chooses to tutor C^1 and C^2 before tutoring the integration skill B^1. The part tasks are sequenced from the subskills to the parent skills, and so on, up the domain hierarchy.

As we pointed out before, the procedural network and the analysis of the domain expert can only provide information about a subset of elements of the student model. Instruction is a valuable source of diagnostic information about the regions of the student model that are inaccessible to the procedural network/domain expert (as in Figure 8). Part-task training can be designed to elicit information about the inaccessible areas of student model: "Avoid guessing - get the student to tell you what you need to know"[8]. This diagnostic information is all the more valuable since it is directly solicited and not deduced with possibly error-prone analysis.

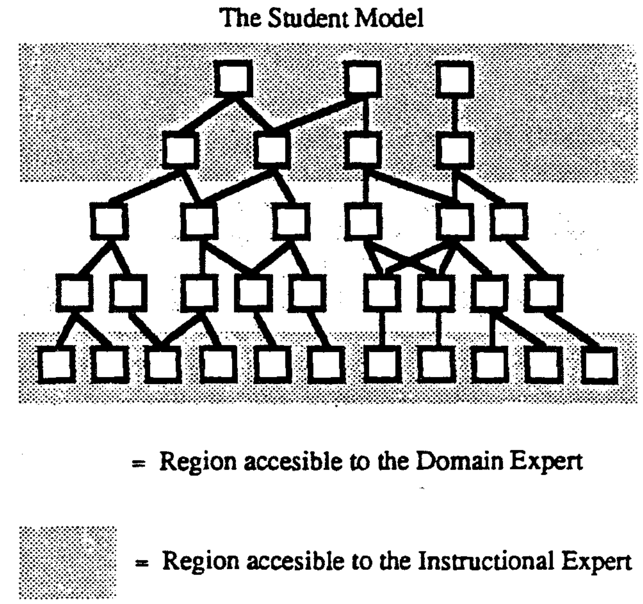

The Student Model
= Region accesible to the Domain Expert
= Region accesible to the Instructional Expert

Figure 8. Regions accessible to the Domain Expert's and the Instructional Expert's diagnosis

Conclusions

The two expert systems in our ITS use a common representation of the student. The domain expert can observe and understand the student's actions with a procedural network. The procedural network lends itself to an initial classification of observed errors. The classified student errors can then be interpreted for information about specific skills and concepts in the student model. The student model can further refine the possible causes of the student errors. Our ITS exploits the hierarchical structure of the student model for both further diagnosis of the student and remediation of the student. The hierarchical representation of the student model is a sound representation for instruction, specifically part-task training, as well as diagnosis of student deficiencies. Fault isolation algorithms can use the hierarchical student model as a decision tree. The instructional expert uses the hierarchical structure of the student model to control the sequence of training.

References

[1]Spiro, R. J., Vispoel, W., Schmitz, J., Samarapungavan, A., and Boerger, A., *Knowledge Acquisition for Application: Cognitive Flexibility and Transfer in Complex Content Domain,* in *Executive Control Processes* (ed. B. C. Britton), Erlbaum, Hillsdale, New Jersey, 1987, pp. 177 - 199.

[2]Brown, J. S., and Burton, R. R., *A Paradigmatic Example of an Artificially Intelligent Instructional System*, International Journal of Man-Machine Studies, vol. 10, pp. 323 - 339.

[3]Rickel, J., *An Intelligent Tutoring Framework for Task-Oriented Domains,* Proceedings of ITS-88, Montréal, 1988, pp. 109 - 115

[4]Sacerdoti, E. D., *A Structure for Plans and Behavior*, Elsevier-North Holland, New York, 1977

[5]Rickel, J., *ibid.*

[6]Self, J. A., *Bypassing the Intractable Problem of Student Modelling*, Proceedings of ITS-88, Montréal, 1988, pp. 18 - 24.

[7]Frederiksen, J. R., and White, B. Y., *An Approach to Training Based Upon Principled Task Decomposition,* Acta Pyschologica 71 (1989), pp. 89 - 146.

[8]Self, J. A., *ibid.*

Integrating PCLIPS into ULowell's Lincoln Logs Factory of the Future

The Center for Productivity Enhancement
University of Lowell

by
Brenda J. McGee
Mark D. Miller
Dr. Patrick Krolak
Stanley J. Barr

ABSTRACT

We are attempting to show how independent but cooperating expert systems, executing within a parallel production system (PCLIPS), can operate and control a completely automated, fault tolerant prototype of a factory of the future (The Lincoln Logs Factory of the Future). The factory consists of a CAD system for designing the Lincoln Log Houses, two workcells, and a materials handling system. A workcell consists of two robots, parts feeders, and a frame mounted vision system.

1. INTRODUCTION

The University of Lowell's Factory of the Future, consists of an intelligent Computer Aided Design (CAD) system, a graphical simulator, and a physical factory. Designed to be autonomous; needing minimal assistance from an operator, the factory is a state of the art prototype for automated manufacturing. This factory consists of two physical workcells, which are connected by a computer controlled material handling system. Each workcell has two robots, vertically mounted cameras which are controlled by a vision system, and parts feeders which have sensors to monitor workcell inventory. The CAD system provides the user interface for designing the houses. The design is sent to a CLIPS scheduling expert system. Thereafter other CLIPS expert systems, aided by the vision system, operate and synchronize the robots and other hardware to manufacture the design. For efficient execution of these parallel expert systems there is a need for a fast, reliable, user-transparent, hardware and operating system independent networking production system. PCLIPS (parallel CLIPS)[1], developed at the Center for Productivity Enhancement, has these qualities allowing concurrent independent CLIPS expert systems to exchange messages in the form of facts. The crucial feature of PCLIPS is a command called *rassert* or *remote assert*. *Rassert* allows a CLIPS process to assert facts into the fact databases of every other CLIPS process, thus communicating cooperatively with one another, ultimately resulting in an intelligent manufacturing workcell environment.

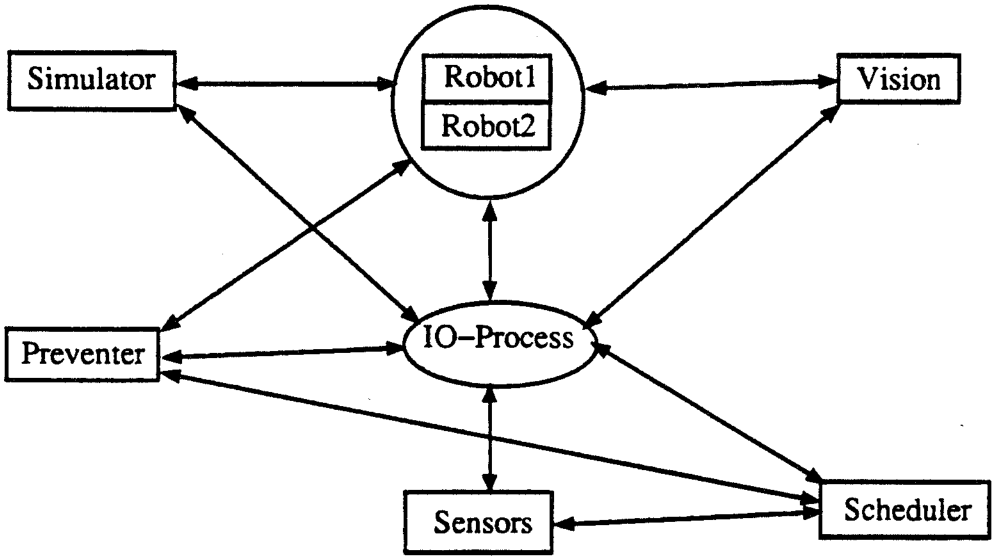


Figure 1. Workcell Processes

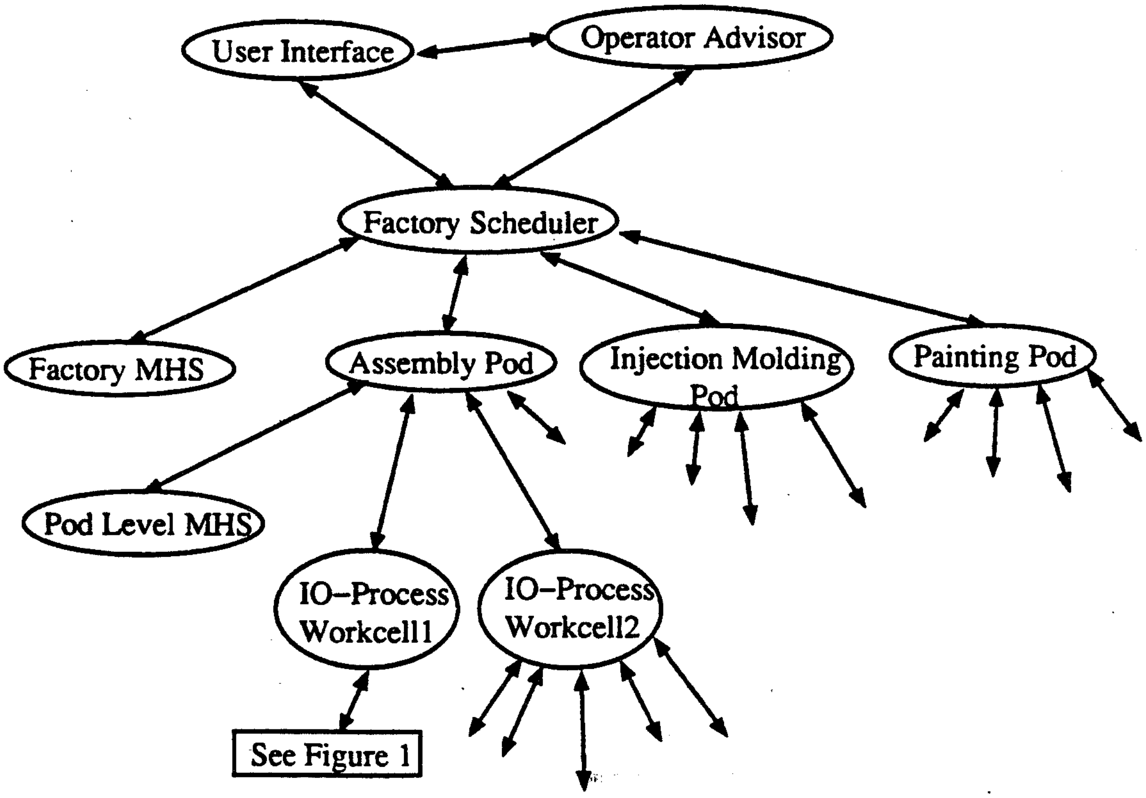


Figure 2. Factory Control

2. PCLIPS and Lincoln Logs: The Concept

Interprocess communication for Lincoln Logs was originally accomplished through a mailbox system, implemented on VMS[1]. Each process in the factory created its own mailbox, and a pointer to the mailbox of any other process that it needed to talk to. This reserved space in memory where messages were left and picked up, using QIOs. This method had two limitations. The first was that it was system dependent. It would only work on VAXEN[2]. The other limitation was the incompatibility between our interprocess messages and CLIPS, which we were implementing at the process level. PCLIPS was chosen, therefore, to replace this mailbox system.

PCLIPS has several advantages. The network operations and protocol requirements for the network are transparent to the user, thus eliminating that concern from the expert system developer. It also works on heterogeneous computer systems, enabling the expert system developer to design platform independent software. Finally, the inter-process messages are in the native format of CLIPS (facts), thus eliminating the earlier need for translating inter-process messages ino facts.

The first issue that we had to resolve was a standard format for interprocess messages since the use of the *rassert (remote assert)* command globally broadcasts each fact, or interprocess message, to every other process running PCLIPS. We used the following format:

(IPM receiver sender $?)

The atom *receiver* is the name of the process who the message is intended for. This is either the specific name of the process (ex. VISION), or the string ALL. An IPM fact with ALL in the receiver position is a message intended for all processes running.

The atom *sender* is the name of the process which broadcasted the fact. When an inter-process message is broadcast, each process picks up the fact and fires a rule in order to test whether or not that fact is meant for that process. Code from the Vision process will serve as an example, as the code in each process is simillar.

```
(defrule interprocess_message
        ?gnim <- (get_next_int_message)
        ?IPM <- (IPM VISION|ALL ?sender ?rm1 ?rm2 ?rm3 ?rm4 ?rm5 ?rm6 ?rm7)
=>
        (retract ?IPM ?gnim)
        (assert (rmessage ?sender ?rm1 ?rm2 ?rm3 ?rm4 ?rm5 ?rm6 ?rm7))
)
```

[1] VMS is a trademark of Digital Equipment Corporation

[2] VAXEN is a trademark of Digital Equipment Corporation

If the fact is not meant for that particular process, a rule is fired that retracts that fact from the list.

```
(defrule IPM_not_for_this_process
        ?IPM <- (IPM ~VISION&~ALL ?sender $?)
=>
        (retract ?IPM)
)
```

Since all the processes are event triggered, there are times when a single process will complete all its current tasks, and will have to wait until a new event occurs. In order to avoid a busy wait, we took advantage of the *salience* option in CLIPS and created a rule that suspends a process until a new event occurs. Since we used the lowest salience possible, this rule will only fire when there is nothing else on the agenda, thus eliminating the possibility of the process being suspended in the middle of a task. When all the rules have fired, whether or not the IPM was for that process, the process goes back into a wait state until the next global fact arrives.

```
(defrule wait
        (declare (salience -10000))
        ?w <- (wait for IPM)
=>
        (retract ?w)
        (call (suspend))
)
```

CLIPS has also been integrated into the factory of the future in the decision making process.

3.1 Preventer (Collision Prevention)

At this time, our collision prevention algorithm allows us to use two robots in a workspace. The Preventer process performs collision prevention by calculating where each robot arm, gripper and part will be located during placement. A robot requests access to the workspace, through an *rasserted* fact. The Preventer then calculates the path the robot will follow to get to it's destination, and determines the potential for a collision or obstruction between any of the following: The two arms, the parts in the robot grippers, and the vision inspection system. The vision system needs a clear view of the part it is inspecting. Otherwise, it may report invalid information.

If the Preventer determines that a collision is possible, it will enforce mutual exclusion to the workspace by delaying *rasserting* the *access granted* fact to the Robot Process until the situation has changed, and the robot has a clear path to its destination.

3.2 Vision (Vision Inspection)

Vision Inspection, done with an overhead camera, occurs after a robot has successfully placed a piece on the work pallet. The Vision system waits for an *rassertedfact* from the Robot process. The fact contains information about the part that needs inspection, namely the part type, it's location, and orientation on the pallet. If the Vision System does not approve of the part's position, it alerts the robot with a fact that includes the calculated offset of the part. When alerted, the robot will re-enter the workspace and attempt to correct the problem. Once the Vision system approves a part, the robot moves on to its next tast.

3.3 Robot (Robotic Control)

We have created a Robot Planner using CLIPS. When the planner, or process starts up, it *rasserts* a task request to the workcell scheduler. When the scheduler returns the task message, the planner breaks the task down into a series of operations. The example we will follow is a Place Part task.

First, the planner must determine the part's location (in the parts feeder, on the jig, on the pallet, etc.). Based upon this information, it then determines its approach path to the object. Once it has the part in its grasp, and the gripper is clear of the part holder, a path to the workspace is calculated. At this point, the robot process must request access to the workspace, which it does by *rasserting* the request to the Preventer process. Once the robot has been given clearance, it calculates a path to the release point, follows the path, and releases the part. It then moves clear of the workspace, and *rasserts* a request for a vision inspection. If the vision system reports the part placement to be outside the tolerance limits, the robot will re-enter the workspace and attempt to correct the error. When the vision system approves the part, the robot sends a task completion fact to the scheduler. It then checks its agenda for any other work. If none exists, it sends another task request message to the scheduler.

The flow of the planner is controlled by two facts, *state* and *action*. When the planner enters a particular state, there are several actions which must be performed sequentially to assure a correct execution. There are several examples of built-in error handling. Whenever an error occurs, the planner will immediately move to the error handler. We force this to occur through the use of a high salience for the error handler initiator.

```
(defrule first-grasp-error-handler
        (declare (salience 100))
        (error occurred)
        (state get-part)
        ?action <- (action grasp-part first-attempt)
=>
        (retract ?action)
          (assert (action grasp-part second-attempt))
)
```

3.4 Sensors (Sensor Fusion)

The Sensor process allows the operator to be informed when there is a change of state in the parts feeder; as well as allowing the operator to shutdown a particular feeder. This control is accomplished by monitoring infra-red sensors near the base of each feeder. The Sensor process continuously monitors these sensors, and *rasserts* facts to the scheduler if a state change occurs. The Sensor process also has the ability to introduce errors into the system in order to test the system's ability to cope with malfunctions.

3.5 Scheduler (Task Scheduler)

The Scheduler Expert System is a dynamic task optimizer. The scheduler reads in a natural language description of the house. After parsing the description, the scheduler dynamically assigns tasks to the requesting Robot Processes. Due to the dynamic nature of the scheduler, it can change the schedule as workcell conditions change, enabling it to track workcell inventory, throughput, and resources. The Scheduler's main goal is to maximize the workcell yield. It achieves this goal by optimizing workcell events to allow parallel execution of robot operations. When mutual exclusion is enforced, one of the robots must wait for the other robot to exit the work space, cutting down on throughput.

3.6 IO-Process (Interprocess IO-controller)

The IO-Process is the parent of all workcell processes. It allows the operator to configure the workcell for the resources available (i.e. material handling system, vision, robots, simulator, etc.) It then starts up the workcell process and remotely asserts a startup fact in each. Afterwards, it monitors all the workcell processes and notifies each workcell process of changing resources. When the job is finished, the IO-Process terminates all workcell processes by *rasserting* a shutdown message.

3.7 Simulator (Workcell Simulator)

The Workcell Simulator provides a mechanism for testing control software without the need for workcell hardware. The Simulator graphically mimics the actions of both robots on a color workstation. While the Simulator is running, the Robot Processes simply redirect their output to the Simulator instead of the physical robots. The Simulator provides handshaking capabilities similar to the physical robots, which allows the operator to simulate a robot error, for testing the reliability of the workcell software.

3.8 Material Handling (Automated Materials Handling System)

The system loads and unloads work pallets into each workcell. It also has the ability to transport pallets from one workcell to another for completion of jobs, if the need arises. Error detecting and handling capabilities have been built into the expert system which controls the MHS. If there is an error, it can determine exactly what the problem is.

3.9 Pod (Pod Scheduler)

The Pod Scheduler is the middle man between the factory scheduler and the individual workcell processes. It not only gives assignments to individual workcells, but also controls the overall execution of workcells that are performing similar tasks. When, the Pod scheduler receives a build request from the Factory Scheduler, it determines which workcell should take on the responsibility of carrying out the request. If the chosen workcell is unable to carry out this request for some reason, it will then choose another workcell to take over the job. There is also a materials handling system at the Pod level that is under the control of the Pod. This setup enables movement of the pallets among the workcells at the Pod Level.

4. Future Directions

The Lincoln Logs Factory of the Future will continue implementing improved versions of PCLIPS as they are developed. One limitation of the current version of PCLIPS is its lack of routing capabilities for remotely asserted facts. Every *rasserted* fact is broadcasted to every other process running PCLIPS. As our factory grows, and subsequently the number of processes running PCLIPS, routing mechanism will have to be implemented to avoid network and CPU saturation. We will also continue the development of our process level expert systems, with a focus on designing and implementing an advisory framework to provide operator, advisor and supervisor assistance at every level of the factory.

5. REFERENCES

[1] Miller, Ross, "PCLIPS: A Distributed Expert System Environment," First CLIPS Users Group Conference, Houston, Texas, August 1990.

[2] Alpha II Reference Guide. MICROBOT Inc. Mountain View, CA. January 1984.

[3] CLIPS Reference Manual. Mission Support Directorate, NASA/Johnson Space Center. Houston, Texas. Version 4.2, April 1988.

[4] RAIL Standard Vision Documentation Package. (AI Part #510-500610). AUTOMATIX Inc., Billerica, MA. March 1987

[5] Kosta, C.P., Wilkens, L., and Miller, M., "A Three-Dimensional C.A.D. system". Center for Productivity Enhancement, University of Lowell. Working Paper #FOF-87-101. Lowell, MA. February 1988.

[6] Miller, M., "Multiple Robot Scheduling". Center for Productivity Enhancement, University of Lowell. Working Paper #FOF-87-103. Lowell, MA. November 1987.

[7] Miller, M., Kosta, C. and Krolak, Dr. P., "Computer Assisted Robotic Assembly" 3rd International Conference on CAD/CAM, Robotics, & Factories of the Future.

[8] Dean, Thomas L., "Intractability and Time-Dependent Planning" 'Reasoning about Actions & Plans, Proceedings of the 1986 Workshop', Morgan Kaufmann, Los Altos, California.

[9] Dougherty, Edward R., Giardina, Charles R., 'Mathematical Methods for Artificial Intelligence and Autonomous Systems' Prentice Hall, Englewood Cliffs, New Jersey. 1988.

omit

A15 Session:
Intelligent Control

S47-63

N96-12960

359010

p.24

An Object Oriented Generic Controller using CLIPS.

By Cody R. Nivens *

----

* Cody R. Nivens is a member of the Information Systems Staff of California Polytechnic State University, San Luis Obispo, California.

ABSTRACT

In today's applications, the need for the division of code and data has focused on the growth of object oriented programming. This philosophy gives software engineers greater control over the environment of an application. Yet the use of object oriented design does not exclude the need for greater understanding by the application of what the controller is doing. Such understanding is only possible by using expert systems. Providing a controller that is capable of controlling an object by using rule-based expertise would expedite the use of both object oriented design and expert knowledge of the dynamic of an environment in modern controllers.

This project presents a model of a controller that uses the CLIPS expert system and objects in C++ to create a generic controller. The polymorphic abilities of C++ allow for the design of a generic component stored in individual data files. Accompanying the component is a set of rules written in CLIPS which provide the following: the control of individual components, the input of sensory data from components and the ability to find the status of a given component. Along with the data describing the application, a set of inference rules written in CLIPS allows the application to make use of sensory facts and status and control abilities.

As a demonstration of this ability, the control of the environment of a house is provided. This demonstration includes the data files describing the rooms and their contents as far as devices, windows and doors. The rules used for the home consist of the flow of people in the house and the control of devices by the home owner.

INTRODUCTION

In the evolution of control mechanisms, it has become apparent that a higher level of knowledge of the system controlled must be embedded in the controller. This project uses the control of a house as an example of a knowledge-based controller. This is done by using the abilities of the CLIPS programming language to utilize user defined routines to input sensor information and to control external devices.

A real-time expert system can be defined as a system that decides in time to undertake a corrective action. Uses of such systems range from the home system described by this project to the control of nuclear power plants and space stations. Such systems have a set of common characteristics: compartmentalization; processes which run over minutes and hours; events which occur on a regular basis; exceptions to standard procedures which augment presently scheduled events; and a set of general rules on how operations in the controlled environment can be influenced by outside factors.

These principles illustrate the use of expertise: Specifically the body of knowledge acquired about the behavior of a complex system. The use of a rule-based knowledge system as a controller must have the following: the ability to control external devices; the ability to receive sensory information in a timely manner; the ability to make decisions within certain time limits; finally, the ability to expand as more knowledge of the behavior of the system becomes available. These principles are only a few that must be examined and met for such a controller to be effective.

The home enviroment is becoming a laboratory for the design of user-friendly control systems. Such systems are programmed in one of several procedural oriented languages and as such they are difficult to expand to meet the needs of the user. A solution to this problem is the use of real-time expert systems. These systems provide the logic in a style that is easy to update and understand. A carefully crafted expert system could be updated and changed by the home owner with little need for their understanding of the rule system.

This paper discusses the combining of CLIPS with objects defined in C++ to create an intelligent controller. The C++ objects define what is controlled. It is mated together with the CLIPS expert system, with CLIPS supplying the expertise for the control of the object. This is done by a loop mechanism which alternates between CLIPS, the C++ objects, and an interrupt information structure. CLIPS controls the object via external functions which access the objects controlled. The user interface employs the objects as a selection mechanism and the assert routine of CLIPS.

OBJECT ORIENTED PROGRAMMING

Object oriented programming is ideally suited for use in intelligent controllers for several reasons. There are several reasons for this. First, an object oriented programming language allows for the creation of an abstract data type. Second, components of a program can inherit functions and data from other objects allowing for the reuse of previous code. Finally, an object oriented programming language provides for the use of polymorphic characteristics. The abstract data type is the key feature of an object oriented syste.

An abstract data type is called a class. A class is composed of the data structure associated with the implementation of the data type and a set of member functions which manipulate that data structure called member functions. There are several advantages to creating a new data type: the hiding of the implementation of a design from its user, encapsulation of both the data and the code that manipulates it, and the restriction on access to the data reducing inter-module dependences. Member functions allow limited access to the data of a class. These are messages that the class accepts for manipulating itself. The function passes the parameters necessary to complete the desired operation. Member functions can be overloaded by using the same name with different parameters. This feature allows descriptive function names to have different routes to the same service.

Inheritance and polymorphism are interrelated in their uses. Inheritance allows both code reusability and the derivation of new data type types that share both the code and data of its base. Polymorphism uses this ability to create derived classes which use functions of the base class and redefine functions in the derived class. Functions which can be redefined are called virtual functions. The virtual function differs from the normal function in that the binding to the function occurs at run time as opposed to the static binding at compile time. There are two major uses for this feature. First, the redefined function of the derived class is used when the base class calls the function. Second, a pointer to a base class can be used on a derived class with the functions redefined by the derived class being used. This allows the calling program to use a derived class without knowing what it is. For example, an array of components which having the same base class can all be sent the same function call even though each component in the array is a different derived class.

For example, consider a set. The implementation of a set in the C++ language consists of two classes as defined in figure 1. The first class is a set element. The second class is the set itself. Two types of set elements are defined in figure 2 to show how the inheritance and polymorphic abilities of C++ work. The main program and output is defined in figure 3. Note that 'a' is said to be an instanciation of the set class. This is similar to saying that x is an instanciation of an integer, but is not an integer class.

DEVICE CONTROLLERS

Computer based controllers fall into three broad categories. First, the group of controller are those controllers that are based on a clock signal. These controllers deal with the use of a set sequence of events that are triggered when a predetermined time arrives. An example of this is a steel mill which heats a piece of metal for a predetermined length of time. A second type of controller is based on sensory input. These controllers must provide a response based on input from the environment of a device. Examples of this type of controller are the closing of valves based on the level of liquid in a tank. The last type of controller is interactive. These controllers generally deal with human input and have their responses geared towards the average person using the device. An automated teller machine is an example of this type of controller.

INTELLIGENT CONTROLLERS

An intelligent controller will be defined as a controller that has the ability to arrive at decisions based on external facts and internal rules of the behavior of the system being controlled. To illustrate such a controller, a model of how the controller relates to the controlled component is needed. The simplest way to achieve this is to consider the controller as an indivisible computer. The inference engine is the cpu, the rules are the programs, and the fact lists are the data. I/O for such a computer consists of external assertions of facts and the execution of commands from the consequent portions of rules.

The use of a central processor for the CLIPS engine is a very useful metaphor. The Rete algorithm uses tokens of the changes in working memory to communicate which rules may fire. Such a system is similar to the concept of an associate memory system. All changes within the memory system happen at one time. The tokens affect only those rules that use the changed component of working memory. Such a scheme allows for a large number of rules and facts to be compiled into a network whose access time is dependent on the changes in working memory.

The model of the cpu would have to be extended to include the use of interrupts. In CLIPS, interrupts could be handled by rules that are fired by the assertion of a specific fact. The chain of events that follows from the interrupt can be determined by the precedence of the rules. The use of the salience feature allows for the running of priority tasks based on interrupt information. Each set of interrupt rules would have a salience level associated with it. It should be noted that the CLIPS system handles input from the interrupts, not the interrupts themselves.

Programming the Device Controller

Programming the CLIPS machine for the use of several independent processes involves little change in method from conventional programming. The major difference between normal programming and this model is the use of a set of rule chains to determine the "program." The need for scheduling, enqueing or dequeing for resources, or rendezvousing between tasks is eliminated. All these things are handled by the working of the Rete mechanism. Two tasks which have independent chains of inference can perform a rendezvous via the assertion of a common fact.

For example, the standard consumer/producer problem can be defined in CLIPS by two rules as shown in figure 4. The producer/consumer cycle starts with an assertion of the specific producer facts and the start fact for the producer rule. The cycle between the producer/consumer is controlled by two facts which are asserted when the particular phase of the cycle is done. Such a system does not have the ability to enque messages, but such abilities can be accessed via an external procedure.

Interrupts

Interrupts and device input are handled in a similar manner. The use of the add_exec_function allows a user defined routine to be used between the firing of rules. This function then has the option of asserting information based on the state of an interrupt or device. The control of such assertions can be handled by two routines defined by the define_function routine. One function enables interrupts from devices and external interrupts. A second function disables the asserting of new facts. A supporting function returns the state of interrupts. Interrupt precedence can be controlled via the salience clause of a CLIPS rule. This allows specific interrupts to have control of the system while they are working. An example is shown in figure 5.

Input/Output

Traditional device input is handled by the add_function routine of CLIPS. This function allows for the creation of a routine which can be used in the RHS of a rule. The function defined would then assert a fact based on the responding device. Output is handled in a similar manner: the defined function would take a multi-variable pattern and consult the appropriate component being controlled.

THE GENERIC CONTROLLER

The generic controller is an object which uses an expert system to provide control to some other object. The controller class has the following components: a CLIPS expert system, a component to control, a simulation to run the component through, an alarm manager for time signals and alarm activations, a command object to pass commands between CLIPS and the controlled object, windows to display output for the user, and a set of I/O ports for information on the component controlled and through which to control the component.

The basic use of the controller consists of loading the information on the windows, the ports, the component information, the simulation information and the files that the CLIPS system will use for a trace of all its output, as well as the rules and data of the controller and the application. Next, either the controller is run in real-time mode where the alarm manager and ports deal with the real-time and hardware of the system, or the controller is run in simulation mode where the time and port values are artificial.

In either case, the controller goes into a loop where the following events occur endlessly. First, the CLIPS expert system is executed for a set number of inferences (rule firings.) Second, if a command was executed by the executive function then the status is updated. Third, the keyboard is checked for user input. If input is found, it is passed to the controlled component to interpret. If the interpretation returns a command string, the string is asserted into CLIPS after the current time is attached to it. Next, the sensor inputs are checked for new data. If input is present, it is asserted into CLIPS after the time is stamped onto it. Finally, the alarms are checked and the time is updated if necessary.

THE CLIPS CLASS

The CLIPS class is not an implementation of the CLIPS expert system, but is an interface to the C routines that define the CLIPS system. The encapsulation of CLIPS in a C++ class has enabled the restriction of the many available routines that provide access to the CLIPS environment. The member function of the CLIPS class provides for the following areas of access. First, the embedding functions of clear, reset, execute and load are given standard names and definitions of their use.

The CLIPS class also provides for the use of I/O routers. These functions allow for access to external I/O devices. The use of this function requires that the functions passed to the I/O router not be a member function of a class. The reason for this is that while the address of the member function is known, the instanciation of the class it is being used by is not known. As such, the I/O router functions are defined as friend functions to the controller class.

The next area that the CLIPS class provides a common interface for is the use of executive functions. The executive function is one that is called by the interpreter of CLIPS rules between rule firing. In this project, the executive function is responsible for asserting sensory information if it is present.

The next member function that the CLIPS class contains is concerned with defining a function that CLIPS can call from the right-hand side of a CLIPS rule. This function can do work outside of the CLIPS environment, possibly returning a value as a predicate function. There are three functions defined in this project: do_command, seek, and set_alarm.

The interaction between CLIPS and external routines are defined in two member functions: The first asserts a string into the CLIPS environment, and the second loads a command object with the parameters passed to a function when it is called by a CLIPS rule.

The last set of functions in the CLIPS class are involved with debugging and status display. These routines deal with the activation of watches on facts, rules and activations. They also provide functions for the display of the CLIPS fact environment and the current agenda of rules to fire.

THE COMPONENT CLASS

The component class is the class which describes the object being controlled. This class provides a generic holder for information on how a system relates to itself. This scheme is a hierarchical system. Objects at one level only access those at a lower level and the parent of the present object. Access across branches of the component tree are not possible in this system. A component provides an object display, I/O, and relational information.

The display information of a component is divided into four parts. The first part is a window display of the contents of a component in a window. The second part is a display of the status information about the component. The next area is a display of the related objects of the component. This part consists of an overlay which fits the related objects into a cohesive whole. The last area consists of the display windows and the index to the window in which the overlay and related components of the component are displayed.

The I/O information consists of several values. The input port id determines which related object is the next component in the component-path name of the input item. If there are no related objects then the value from the port is the state of the device or sensor. The output value, the command or value related to the place of the component in the system begin controlled, is sent to the output port. If there are no related objects, the output value is the state of the component. The I/O ports are an array of ports that are used for input/output operations. These allow for an index to determine which input port and which output port should be used. The I/O ports are used by the interrupt mechanism to establish an interrupt path to a component. This is done by enqueing the id of the component in the set of related objects of the parent component.

The related object information consists of the related objects, their number and which are currently selected. This information is used to create command strings that are asserted into the CLIPS system. The related object information identifies which is the master (root) component and which component is active (being selected from.)

The use of individual I/O ports, command levels and display windows allows the programmer to create generic components that are independent of the device being controlled. For example, the application of this project is a house controller. In the test case, there are 3 rooms, 10 lamps, 13 outlets, 12 sensors, and 12 command components. All can be represented by generic components. All I/O in the system is done by the generic component; no further programming is needed. A draw-back is that the number of components goes up with an increase in command complexity with any device. The simple solution to this is to create new device components derived from the base component.

SUPPORTING CLASSES

The alarm manager class has four major functions. First, it is responsible for the time and date clock. Second, it holds the times of alarms that are active in the CLIPS environment. Third, when an alarm occurs, the alarm manager asserts a time fact into CLIPS for the time of the alarm. Finally, the alarm manager class is responsible for the time stamp when an event occurs.

The command class acts as a data carrier for communications between CLIPS and the component. There are two parts to a command: the count of lines in the command, and the lines themselves. The command class is defined as an array of strings. The dimensions of the array are dynamically enabled when the class is instanciated. It must be noted that the CLIPS version used in this project has multiple field variables containing extra lines of information, specifically, the fact name-field (the first field in the fact.) Hence, the offset must be one greater than the position of the field in the multivariable of the CLIPS rule. This can be used to allow one routine to interpret many commands, as the command is always the first field.

The port class defines an input/output medium. The port can either be used for real I/O or for simulated I/O. Real I/O is device and implementation dependent. The simulation of the port input is done via an index that the port acquires along with a simulation when it is instanciated. This id is passed to the simulation which returns -1 if either the index is lower than the ports simulated or there is no input for the port ready. The ports used for the house application are shown in figure 6. Interrupts use the ports to signal that a value is present. This is done by the interrupt routine which calls the component. It changes the state of the component and creates an interrupt trail via a member function of the parent of the component.

The simulation class contains an array of values that are assigned to ports dependent on the time that the simulator has for the next input. The first member function deals with the loading of the simulation values from an input stream. There are two functions which deal with stepping the simulation and testing if the simulation is done. Two further functions deal with returning the simulation time and the simulation value given a port index. The private variables of the simulation define the number of simulations, the offset for the port index, the current simulation time, and the index of the next simulation event.

SYSTEM RULES AND FACTS

The system rules are divided into four areas: changes in sensory information, time and date maintenance, alarm durations, and activation of alarms.

The first set of rules in the system CLIPS file deals with sensory information. This section is divided into two parts. The first deals with the rules involved with the processing of sensory input. There is only one rule: sensor-reset. This rule resets the sensor input states when the sensor cycles from ON to OFF or OFF to ON.

The second set of rules dealing with sensory information seeks status of components in the system. There are three rules in this set: seek-status, status-seek, and reset-seek. Seek-status is used to reset the knowledge system given existing state facts. This allows for the periodic checking of the consistency of the knowledge base against the controlled component. Status-seek processes the results of a seek operation by creating a state fact. Seek-status and status-seek work with a control fact: seek-state. Seek-state carries a list of selector elements, which quizzes related objects and their descendants for their status. Reset-seek retracts the seek-state fact if no other rules are activated by the fact. The structure of the system facts are listed in figure 7.

The second part of the system rules is composed of guidelines related to the maintenance of time and date. When the date changes at midnight, the alarm manager asserts the new date.

This assertion causes the rule change-date to fire. This rule asserts seek-state on all components and process-alarms to set up the alarm manager for the next 24 hours. The reset-time rule removes the time fact if no other rules are activated by it. The time fact is asserted by the alarm manager when an alarm occurs.

The third set of rules are those involved in processing alarm times. There are three rules. Process-alarms is activated by the process-alarms fact asserted by the change-date rule. Set-alarm-time sets the time of a newly activated alarm. Reset-process-alarms removes the process-alarms fact if no other rules are activated by it. A more complex system of rules would process alarms on an hourly basis.

The next section of the system rules is concerned with rules which govern the use of durations. Durations are alarms which run from one time to another. This section is divided into three parts. The first part is the rule set-duration. This rule is activated by process-alarms asserted by the rule change-date. The second part consists of the rules start-duration and reset-start-alarm. Start duration fires when the alarm created by the duration is activated. It asserts start-alarm fact containing the id of the alarm activated. This is asserted for application rules use when alarms are activated. Reset-start-alarm removes the fact if no other rule is activated by it. The last part consists of the rules end-duration and reset-end-duration. End-duration removes the alarm associated with a duration. It fires when the duration reaches its end. It also asserts the end-alarm fact with the id of the duration associated with the retracted alarm. Reset-end-alarm removes the fact end-alarm if no other rules are activated by it.

The final set of rules consists of the rules for the firing of alarms. There are nine rules which correspond to the types of alarms. All alarms have the following in common: an id, a type, a possible repetition count, a date and time to fire, and information specific to the application which is used to command the controlled component. The alarm types are listed in figure 8. Alarm fact structures and constants are listed in figure 9.

THE APPLICATION

The application of this project consists of a house controller. The basic design focuses around the use of the X-10 house controller. X-10 is an industry standard for the control of components in a home. The application consists of a three room building. Each room has at least one door, one or more windows, lamps and outlets. For each room, there is an overlay file, a list of devices in the room as well as CLIPS facts on the room. The house as a whole also possesses an overlay.

The controller is used in a command mode by selecting which room to work in. Next the type of device to control is selected.

The device is then selected. Finally, the command to perform on the device is selected. When this is done, a command is sent to the CLIPS controller. The controller in turn sends a command to the component to perform the operation.

The application and the controller have performed well in simulation runs. It will soon be implemented in a model system consisting of the basis house that is now defined along with X-10 controlled devices. The outcome of this implementation will be presented at the CLIPS Users Conference.

HOUSE RULES AND FACTS

The house rules file is divided into three parts. The first part deals with door direction and specification information. The second part deals with room and house occupancy. The last part contains exception rules for possible error conditions.

The door direction rules are outside-door-dir and door-dir. Outside-door-dir is concerned with determining if a person is entering or leaving the house. Door-dir determines which room a person is entering and leaving.

The next set of rules deal with house and room occupancy. The first rule is changing-rooms which adjusts the appropriate room occupancy counts. The next rule is person-entering-house. It adjusts the house occupancy count and the room being entered or left.

The last set of rules contain two exception rules. The first is person-too-many-room. This rule resets the room count and issues an exception message to standard out. The second rule is person-too-many-house. This rule resets the house and appropriate room count and issues an exception message to standard out. Figure 10 shows the house controller fact structures.

SUMMARY AND CONCLUSIONS

The use of CLIPS as a real-time controller in a house has be examined. The CLIPS expert system is suited to this work because of its abilities to define external functions and executive functions which allow the insertion of interrupts into the working storage of the system. This allows the CLIPS system to be viewed as a computer with programs, interrupts, and input/output capability.

The use of rule-based systems as opposed to procedurally-based systems gives a programmer greater control over the logic embedded in a system. As the logic of a system goes beyond a certain limit of comprehension, rules for clarity become necessary. Traditional control systems in conventional languages are based on simple formula describing the system. In an application such as a home, a descriptive formula is all but impossible. Yet, it is possible to describe the behavior of the system in pseudo-English. This pseudo-English allows the programmer to develop rules that describe the behavior of the system. These rules are then given directly to the controller without need for additional programming or development.

The use of an object oriented programming language allows the creation of descriptive fact structure related to the component being controlled. C++ is a language which provides such capability in a familiar setting. A programmer familiar with C will have little difficulty improving or adding code. This reduces the cost of development of new projects, and their maintenance once they are in operation.

Intelligent controllers are a natural extension of Artificial Intelligence into the fields of conventional programming and control. Embedded systems may one day have the ability to control and learn from previous conditions and actions. Research into such systems will prove to be profitable and stimulating. CLIPS is an excellent tool with which to conduct such research as it is written in C, which combined with C++, allows for programmer involvement in the development of the rules and structure of the application.

Figure 1 - Set Classes

```
class set_element {
friend set;
private:
        set_element * next; // pointer to the next element in the set.


public:
        set_element();
        // Effects: Creates a set element.


        virtual print();
        // Effects: Prints the set element's contents.
};


class set {
private:
        int size;               // number of elements in the set.
        set_element * elements; // The elements in the set.


public:
        set();
        // Effects: Creates a set.


        add(set_element* a);
        // Requires: A set element to add to the set.


        print();
        // Effects: Prints the contents of the set.
};
```

Figure 2 - Derived Set Classes

```
class card : public set_element {
private:
        int value;
        int suit;


public:
        card(int v,int s);
        // Requires: A value and a suit.
        // Effects: Creates a card with value of suit.


        print();
        // Effects: Prints the value and suit of the card.
};


class toy : public set_element {
private:
        char* name; // Name of the toy.
        char* color; // Color of the toy.


public:
        toy(char* n,char* c);
        // Requires: Name and color of the toy.
        // Effects: Creates a toy.


        print();
        // Effects: Prints the toy.
};
```

Figure 3 - The use of the Set class and its output

```
#define DIAMONDS 1
#define HEARTS 2


main()
{
        set a;

        card d10(10,DIAMONDS);
        card h1(1,HEARTS);


        toy doll("doll","blue");
        toy ball("ball","green");


        a.add(d10);
        a.add(doll):
        a.add(h1);
        a.add(ball);


        a.print();
}
```

```
OUTPUT

green ball
ace of hearts
blue doll
10 of diamonds
```

Figure 4 - Comsumer/Producer Rules

```
(defrule consumer
    ?f<-(consume $?a)
=>
      .
      .  misl processing
      .
    (retract ?f)
    (assert (produce a)) )


(defrule producer
   ?f<-(produce a)
     .
     . Specific producer info
     .
=>
   (retract ?f)
     .
     .  misl processing
     .
   (assert (consume $?a)) )
```

Figure 5 - Interrupt Rule

```
(defrule fire-rule
        (declare (salience 10000))
        (fire ?room)
        (sprinklers ?room $?sprks)
=>
        (sound-alarm)
        (bind ?i 1)
        (while (< ?i (length $?sprks))
               (do_command ?room (member ?i $?sparks) ON)
               (bind ?i (+ ?i 1))
        )
)
```

Interrupt asserts (fire room1).

Figure 6 - Input/Output Port Definitions

INPUTS

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
NULL	X10	House	Room	Device Type	Device	Command	Dim Value

OUTPUTS

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
NULL	X10	NULL	NULL	NULL	NULL	NULL	NULL

Figure 7 - Application Independent Fact Structures

The following information consists of the structure of the facts that are used by the controller. These facts are generic to all applications that run on the controller. In the house rule, data and alarm files, their use is further illustrated.

Application Facts: These rules deal with the contents of application specific information. The format of the rule does not change only the contents of the $?info field.

```
(action ?action-type $?info ?state ?time)
(sensor $?info ?state ?time)
(state $?info ?state)
(status $?info ?state ?time)
```

Where:

?action-type - Action description: Usually user defined based on sensor information sensor reset is signified by break in ?action-type field.

$?info - Application specific information

?state - State location is in (i.e., on, off, 0, 1, etc.)

?time - Time status was returned from controlled object.

Figure 8 - Alarm Types

ALARM EVENT TYPES

TYPE	DESCRIPTION
one-time	Fires on specified date and time and is removed from the system.
daily	Fires every day.
week-day	Fires Monday through Friday.
week-end	Fires on Saturday and Sunday.
weekly	Fires each week on the same day.
biweekly	Fires on the first week day and then 3 days later.
monthly	Fires each month on the same day.
every-day	Fires every specified number of days.
every	Fires every specified number of seconds.

Figure 9 - Fact Structures and Time Constants

```
     Alarm Facts:
        (alarm ?id ?event-type ?event-repetition ?year ?month
                ?day ?time $?info)
        (alarm-mark ?id ?event-type ?event-repetition ?year
?month
                ?day ?time $?info)
        (date ?year ?month ?day ?day-of-week ?julian-date)
        (duration ?id ?from ?to)
        (new-date ?year ?month ?day ?time ?day-of-week
                ?julian-date)
        (time ?secs)

Where:
        ?id                 - Alarm id - either number or character
                              string.
        ?event-type         - Determines how and when alarm is
fired.
                              See above table for event types.
        ?event-repetition - Determines frequency of event.
                             Used by
                                weekly    - Day of week to
                                                activate alarm.
                                biweekly - First day of week
                                                to activate
                                                alarm on.
                                every-day - Number of days till
                                                next alarm.
                                every      - Number of seconds
                                                till next alarm.
        ?year               - Last two digits of year.
        ?month              - Month id based from zero.
        ?day                - Day of month.
        ?time               - Time of day in seconds.
        $?info              - Application specific information.
        ?secs               - Number of seconds since midnight.
        ?from               - Time in seconds to start alarm.
        ?to                 - Number of seconds to allow alarm to
run.
        ?day-of-week        - The day of the week with Sunaay as 0.
        ?julian-date        - Days from beginning of year to
present.
```

Constant Facts: These facts are constant through out the life of an application and from application to application.

```
        (biweekly-map 1 2 3 4 5 6 0 1 2 3)
        (month ?month-id ?month-name ?days-in-month)
        (week-days 1 2 3 4 5)
        (week-end-days 0 6)
        (year-length 365)
```

```
Where:
       ?month-id       - Id of month (January - 0)
       ?month-name     - Jan, Feb, etc.
       ?days-in-month  - length of month in days
```

Figure 10 - House Controller Fact Structures

The following consists of the structure of the facts that are unique to the house controller application.

```
(door ?house ?door ?room1 ?room2)
(door-sensor ?house ?room ?sensor ?door-type ?door)
(outside-sensor ?house ?room ?sensor)
(people-in-house ?house ?number)
(people-in-room ?house ?room ?number)
(window-sensor ?house ?room ?sensor ?window)
```

Where:

?house	- House id door is in
?door	- Door id
?room1	- Room 1 id
?room2	- Room 2 id
?sensor	- Id of sensor
?door-type	- Door type: door, outside-door
?number	- Number of people
?window	- Window id

S48-76

N96- 12961

359011

p. 16

Applying CLIPS to Control of Molecular Beam Epitaxy Processing

Arthur A. Rabeau[1], Abdelhak Bensaoula[2], Keith D. Jamison[2], Charles Horton[2], Alex Ignatiev[2], John R. Glover[3], University of Houston, Houston, Texas 77004.

[1] Department of Electrical Engineering and Space Vacuum Epitaxy Center.

[2] Department of Physics and Space Vacuum Epitaxy Center.

[3] Department of Electrical Engineering.

1. Introduction

A key element of U.S. industrial competitiveness in the 1990's will be the exploitation of advanced technologies which involve low-volume, high-profit manufacturing. The demands of such manufacture limit participation to a few major entities in the U.S. and elsewhere, and offset the lower manufacturing costs of other countries which have, for example, captured much of the consumer electronics market.

One such technology is thin-film epitaxy, a technology which encompasses several techniques such as Molecular Beam Epitaxy (MBE), Chemical Beam Epitaxy (CBE), and Vapor-Phase Epitaxy (VPE). Molecular Beam Epitaxy (MBE) is a technology for creating a variety of electronic and electro-optical materials. Compared to standard microelectronic production techniques (including gaseous diffusion, ion implantation, and chemical vapor deposition), MBE is much more exact, though much slower. Although newer than the standard technologies, MBE is the technology of choice for fabrication of ultraprecise materials for cutting-edge microelectronic devices and for research into the properties of new materials.

Investigation of MBE processing science and technology is one of the foremost goals of the Space Vacuum Epitaxy Center (SVEC) at the University of Houston. SVEC, a NASA-sponsored Center for the Commercial Development of Space, is a consortium which includes a number of industrial, academic and government members. Research at the Center includes both study of MBE science at the basic level and investigation into advanced MBE techniques and applications. SVEC's centerpiece project is the Wake Shield Facility (WSF), an orbital MBE laboratory which holds promise for unparallelled quality and volume of MBE processing. The first flight of the WSF is scheduled for April 1992, at which time it will be held at the end of the Shuttle's manipulator arm for an experimental run lasting about two days.

As will be seen below, each individual MBE experiment is a relatively slow process, with a mixture of many straightforward features and some requiring careful attention by an experimenter. Without computer automation, MBE is manpower-intensive to the extent of absorbing a large amount of researchers' time. Fortunately, it is relatively simple to apply automatic control to a typical MBE production system with a PC-class microcomputer. This has been done with the laboratory MBE system at SVEC, using a PC-AT computer to control the sequencing of basic experiment actions. However, the conventional program used to control the experiment is relatively inflexible in any unusual or contingency situation. To remedy this situation and take the place of the experimenter as much as possible, an expert system addition is being developed at SVEC using the CLIPS (C Language Integrated Production System) expert system tool. The applications and implementation of this CLIPS application are described below.

2. Overview of Molecular Beam Epitaxy

The term epitaxy refers to the accumulation of atoms on a surface in an orderly fashion. This means that, if atoms accumulate epitaxially on a crystalline surface, the new atoms will form a crystalline structure that duplicates and extends the lattice of the original crystal[1]. In MBE parlance, the original crystal surface is known as a "substrate" and the deposition-accumulation process is simply called "growth." In the ideal case of epitaxial growth ("two-dimensional" or "layered" growth), hot atoms falling on a hot crystal will have enough kinetic energy when they hit the substrate to migrate to an unoccupied, energetically-favorable spot on the surface where it bonds with neighbor atoms to form flat surface "islands." Thus, the material being deposited will form in ordered layers a single atom thick.

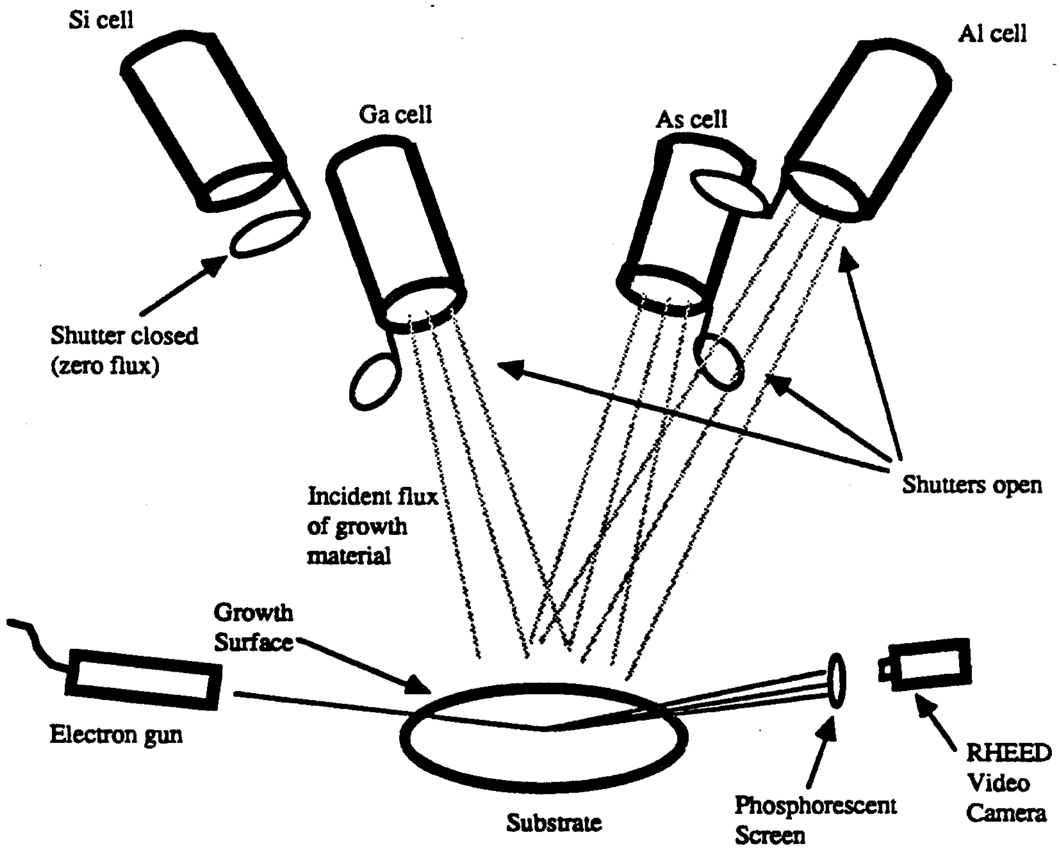


Figure 1. MBE Processing (Growth of AlGaAs film)

MBE growth is achieved by directing a flux of the desired growth materials onto a substrate, which must be in an ultrahigh vacuum (UHV) on the order of 10^{-11} torr to avoid contamination of the growth surface (1 atmosphere = 760 torr). The deposition flux is provided by beams of atoms evaporated from solid ingots heated in cylindrical crucibles ("cells"). A typical MBE growth process, in which layers of aluminum gallium arsenide (AlGaAs) are deposited on a GaAs substrate, is illustrated in Figure 1. The process and apparatus shown are enclosed, in the laboratory, in a stainless steel vacuum chamber pumped down, baked out at about 200° C for about two days (to drive out contaminants from the chamber walls) and pumped down further to its final operating pressure using ion and turbomolecular pumps.

3. MBE Processing

The basic method of MBE growth is fairly straightforward. As shown in Figure 1, the substrate is placed in front of the deposition sources (effusion cells) which contain ingots of the material to be deposited. The substrate is heated to drive off surface oxides and other impurities and then is adjusted to the proper temperature for favorable surface growth conditions. The cells which are to be used are also heated to drive out impurities, and are then adjusted to the proper growth temperatures, i.e. the temperature for each cell which yields the proper evaporated flux of its deposition material. Care must be taken during this step to avoid thermally stressing the ingots as well as the crucibles themselves. When the proper temperatures have been attained, flat shutters covering the aperture of the appropriate source cells are opened, permitting evaporated atoms from the cells to reach the substrate "target". (It should be noted that even with the sources active, the entire growth chamber is still in a hard vacuum by most standards.) Atoms from the active cells (in this example, aluminum, gallium and arsenic) spray onto the substrate and collect in an ordered manner, forming a lattice on the substrate in a layer-by-layer manner (if the growth parameters are correct and impurities are minimized). A typical growth rate is about one monolayer (single atomic layer) per second, or about a micron per hour. Typical temperatures involved are approximately 150° C for the substrate, 200° for the As cell, 1050° for the Al cell and 950° for the Ga cell.

The principle means for determining the rate and characteristics of the growth is electron diffraction monitoring, also as shown in Figure 1. In this technique, called RHEED (Reflection High Energy Electron Diffraction), 10 keV electrons are fired at a grazing angle onto the substrate as growth occurs. The electrons are diffracted by the top few layers of atoms on the growth surface, and the constructive and destructive interference forms a diffraction pattern on a phosphorescent screen opposite the electron gun. A video camera is used to monitor the pattern, which can indicate whether two-dimensional growth is occurring or not, and what the surface crystal characteristics are. A trained MBE physicist can determine whether or not the growth process is occurring satisfactorily by looking at the screen, and adjust the parameters accordingly. Also, since layered growth produces regular cycles from maximum constructive to maximum destructive interference in the diffracted beams, the physicist can tell how many monolayers have been deposited by simply counting the number of cycles of intensity in the diffraction pattern.

4. Control of MBE - Hardware

There are a variety of devices in an MBE system with a mixture of instrumentation and control interfaces. These are summarized in Table I below. The most important control devices are those which operate the cells, which are viewed from a control standpoint as the effusion sources and associated shutters taken together. Under optimum circumstances, a particular cell will yield a known flux of its material when its temperature reaches a certain setpoint and its shutter is opened. If all conditions were known and constant, it would be possible to obtain highly reproducible results from run to run without any monitoring.

Table I. MBE Instrumentation and Control Interfaces

Device	Function	Interface
Source Cell	Source effusion	Heater driven by programmable power supply Power supply driven by controller voltage signal Controller driven by serial command link: Setpoint from computer Power signal from controller Power (voltage, current) from power supply
	Temperature sensing	Thermocouple voltage signal to controller Controller reports voltage via serial data link
Shutter	Flux modulation	Shutter motor driven by digital control board Control board driven by computer digital output
Ion Gauge	Pressure sensing	Sensor generates analog reading of pressure Computer A/D reads sensor signal
Mass Spectrometer	Composition analysis	Mass spec generates numeric readings Computer starts/reads via serial comm link
RHEED	Electron gun	Electron gun controlled by voltage signals Computer D/A generates voltage signals

Of course, the conditions of neither the effusion cells nor the other parts of the growth chamber remain the same. A variety of sensors are used to provide feedback from the source cells themselves (controller signal level, thermocouple reading, power supply levels) and from other devices which monitor the flux of the beam and chamber environment (ion gauges, mass spectrometer). Information from these sensors is used not only to monitor the proper progression of an experiment and watch for fault conditions but also to confirm settings of previous growth runs and to calibrate settings against each other when system modifications are made.

The usage of the devices discussed above is illustrated by analyzing the growth process shown in Figure 1 with referral to Table 1. We consider the growth of aluminum gallium arsenide (AlGaAs) on a typical substrate, e.g. gallium arsenide (GaAs). Initially, the substrate and sources are all at standby temperatures (Al: 600° C, Ga: 500°C, As: 100° C, substrate: 100°C) with all shutters closed. The first step is to warm up the sources and substrate to growth temperature (Al: 1050°C; Ga: 950°C; As: 200°C, substrate: 160°C). This is done by the computer issuing a serial command to the temperature controllers to hold a certain setpoint. In each case, the source or sample must be ramped or "staircased" up in temperature through a certain range in which it is especially vulnerable to undue thermal stress. (The aluminum, for example, is actually molten at growth temperature and must be eased through a phase change.) Also, before reaching their final values, each source/sample is heated above its growth temperature by a small amount to drive off surface contaminants and oxidation. The sequence of warming up the system can take up to about two hours.

When all temperatures have been reached as indicated by the temperature controller readings (measured via thermocouple), the sources are checked for proper flux. This is done by opening the shutter for each cell (by generating a discrete digital signal to the shutter motor controller) and checking the value of its pressure reading with an ionization gauge. (These readings should agree from run to run within about 25 percent.) The desired fluxes are obtained by adjusting the cell temperatures up or down. With all cells properly set, the shutters for (in this case) the aluminum, gallium and arsenic cells are opened and growth begins. At this point, growth is now monitored by using the ionization gauges and mass spectrometer to check the deposition fluxes and the RHEED pattern to verify that epitaxial growth is occurring as planned. When the experiment is finished, the shutters are closed and all temperatures are taken down in reverse sequence to standby temperatures.

5. Control of MBE - Software

Epitaxy process control, as seen above, does not generally require much rapidity of response or analysis on the part of the controlling system, unlike most "real-time" process applications. This fact has enabled us to develop the MBE control software for the SVEC laboratory to satisfy other important requirements, namely: (1) the need to isolate software development from the hardware as much as possible to accommodate changes and transfers to other systems; (2) the need for ease of software development and maintenance in an academic environment with regular personnel changes; and (3) the need for an open architecture to allow additions and other upgrades (such as integration of CLIPS into the software).

Based on these needs, the primary MBE control software at SVEC has been designed to be modular and functionally layered. Modularity, i.e. separation of different software functions into individual units, allows for rapid development of the code by relatively uncoordinated individuals and groups of programmers - again, a desirable feature in an academic setting where regular schedules are difficult to set. Layering allows for a clean separation of the details of the system hardware from the purpose and form of the control software itself. This eases design of the code to make it user-friendly and useful for experimenters who are concentrating on science aspects rather than on esoteric details of programming. In effect, it enhances contact between the highest level of the experiment - the user - with the basic level - the physical processes going on in the MBE growth itself.

The layering begins at the lowest level, that of hardware. Although most MBE chambers and supporting equipment are essentially similar, the control and data-acquisition interfaces vary widely from manufacturer to manufacturer, so the "look" of the devices to the controlling system can be very different. At the hardware level, then, nothing is assumed other than the basic kind of information the devices collect and accept. The types of parameters which are measured/controlled (e.g., flux of gallium atoms, temperature of the substrate) are known but the manner in which they are changed or monitored is a detail which varies as the laboratory equipment is maintained or upgraded. Thus, these details should be encapsulated as much as possible.

This encapsulation or isolation of system hardware details is achieved by the next level of control, the lowest level of software: the hardware-specific front-end code. This code is composed of drivers and linkages which use and manipulate the machine-specific data on the "bottom" side, i.e. that which couples to the hardware. However, on the "top" side, that which deals with the rest of the software, the view is of process parameters such as those mentioned above. The front-end software thus separates the experimenter's model (physical variables) of the MBE process from the programmer's model (e.g., writing a string to a serial port, reading a D/A signal). The modules which perform this function can be changed relatively easily to accommodate different types of equipment, in a manner similar to changing printer drivers on a word-processing program. Unlike those drivers, however, the front-end modules separate conceptual data levels rather than perform a direct translation.

Above this level is the software that deals with the experiment control itself, which is termed the supervisor level. The supervisor oversees the process by dealing with the process variables on one hand and the commands issued by the experimenter on the other. It performs the timing functions for the experiment, setting temperatures, waiting for setpoints to be reached, opening and closing shutters at predetermined events or intervals, and checking the system for fault conditions. The supervisor software is responsible for suspending operations (closing the shutters, possibly bringing temperatures down to standby) and notifying the experimenter in event of a fault. A fault could be anything from a temperature controller time-out reported by the front-end software to an out-of-range condition on a cell (e.g., measured temperature above high operating limit). Such software, implemented in Turbo Pascal version 5.0, has been in operation at SVEC on a trial basis for about four months and shown acceptable performance running on an AT-class computer. Current capability involves cell temperature and shutter control, with temperature range-checking implemented. Monitoring of flux (pressure) gauge and mass spectrometer data will be added during the summer of 1990. Feedback on the shutter status, requiring some modifications to the MBE hardware, should also become available during this period.

6. Integration of CLIPS into MBE Control Software

Using the MBE process control software described above frees up time for experimenters to a certain extent. However, it is limited to operating by preset parameters alone. If the process does not fit these parameters as it moves along, the supervisor program can only either continue or suspend the process while signalling for operator intervention. This means that there is still a need for an experimenter to be immediately available for responding to computer-generated events. To compensate for this, we seek to add a layer of higher "understanding" above those described above - a layer of knowledge and guidelines for dealing with the exigencies of MBE growth that does not need a human operator present. The layer we are describing, of course, is an expert system. The system to be applied to the MBE software is being developed and tested, and ultimately integrated, at SVEC using CLIPS version 4.3.

The modular-layered structure of the conventional MBE process control software makes it easy for CLIPS to be added to the system. The block architecture of the epitaxy control system is shown in Figure 2. Again, the lowest level is the instrumentation and control hardware itself, topped by the front-end software. The front end takes in data in raw form using machine-specific codes and converts them to process-variable information. For control, the data flows and conversion occur in the opposite direction as commands from the supervisor are converted into the appropriate groups of control signals. Above this level, the supervisor code stores and monitors the process data, comparing it to prestored configuration data and "scripts" of process commands entered by the experimenter.

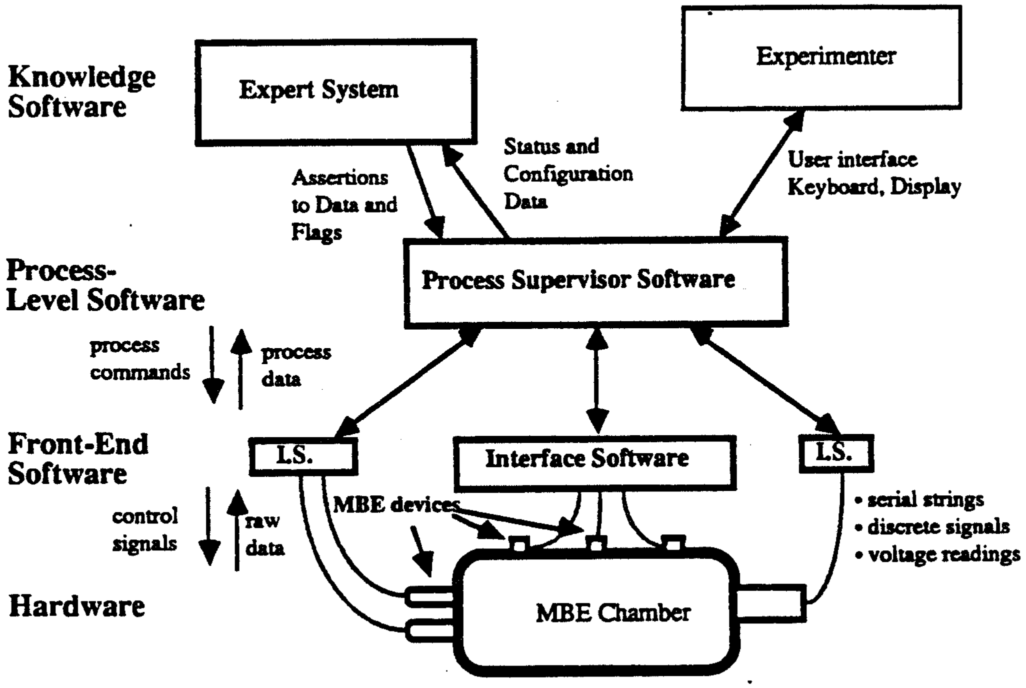


Figure 2. MBE Control Layers

The expert system, as seen in Figure 2 above, fits conceptually into this schematic above the supervisor and "halfway" below the human experimenter. The experimenter, of course, is the final authority on any facet of MBE processing, but when operating unattended, the expert system will have enough of the experimenter's knowledge and experience loaded into it that it will be able to make the same adjustments and decisions an experienced human researcher would make.

The mechanism for implementing this is scheme is fairly straightforward. Since the control software discussed in section 5 is written in Turbo Pascal, the code is being rewritten in Turbo C to take advantage of the direct interfacing methods between C and CLIPS[2]. This allows data to be transferred back and forth between CLIPS and the rest of the program. In this program, then, CLIPS is used to "advise" (actually order) the supervisor what to do in a given situation, based on data passed to it from the supervisor, its own rulebase, and data gathered directly by CLIPS.

As an example, consider a typical sequence, wherein the supervisor program obtains a flux reading for the aluminum cell, stores it in a global data area and finds it 50% lower than it should be. The supervisor then pauses the growth (closes the shutters), and then uses the **assert**(string) function to add the facts (flux Al low) and (growth status paused) into the CLIPS knowledge base. The supervisor now uses the **run**(iters) function to call CLIPS and allow forward chaining to proceed. Appearance of the new facts causes rules to fire which retract any previous items about (growth status) and (flux Al). The expert system can now invoke C functions which return data about the cell directly from global storage, such as power and temperature readings. CLIPS then forward-chains with all the data to come to a conclusion about what to do about the misbehaving cell. After reaching a conclusion, CLIPS uses C functions to set flags which tell the supervisor to raise the temperature, notify an operator or any other appropriate action; then control is returned to the supervisor.

Typically, the supervisor would invoke CLIPS after each polling cycle of the MBE devices, i.e. after all the process variables have been refreshed. The supervisor performs the initial checking on the variables as given in the above example; the boundary checks can be performed much faster this way, instead of the expert system individually retrieving and testing each piece of data. When called, CLIPS can be allowed to run to completion if a contingency condition exists, or otherwise can be restricted to run through a small number of rules at a time. Another consideration on invoking the expert is the mode of experiment at the time. For example, during experiments in Atomic Layer Epitaxy (ALE), the experimenter attempts to grow single monolayers of atom, which requires rapid (<1 sec) cycling of the shutters. During this type of experiment, the supervisor will not invoke CLIPS because it is too busy; in fact, all device polling might be suspended during such an experiment. During intermediate-speed runs, CLIPS would definitely be called with a firing limit of just a few rules.

7. Application of CLIPS to MBE Processing

As illustrated in the example above, CLIPS has two basic roles in the MBE processing system. The first is the monitoring and adjustment of growth parameters which are not at their desired points; this is required quite often in MBE work even when there is nothing "wrong" with the MBE apparatus. It is a combination of a number of quite normal factors, which MBE experts have learned to work around - results are simply calibrated for the changed parameters. Naturally, there are also times when an errant parameter is the result of a malfunction in the control or process hardware. The second role of CLIPS, then, in the MBE control software is to guard the process and handle such situations while preserving in order (1) safety, (2) chamber function and (3) as much as possible in the way of experimental results.

C-7

As an example, consider the operation of an aluminum (Al) effusion source cell as depicted in Figure 3. During a growth run, we set the cell (actually, the temperature controller driving the power supply driving the heater filament) to a certain temperature setpoint. At that setpoint, there should be a certain flux (±25% from run to run), a certain power signal level reported by the temperature controller to maintain the setpoint, and a certain range of voltage/current readings from the power supply itself as it pushes power through the resistive coil of the heater filament. The temperature of the cell is measured by a thermocouple touching the back of the crucible; the voltage generated across the thermocouple is measured and interpreted by the temperature controller, which is calibrated (presumably) for the correct thermocouple type.

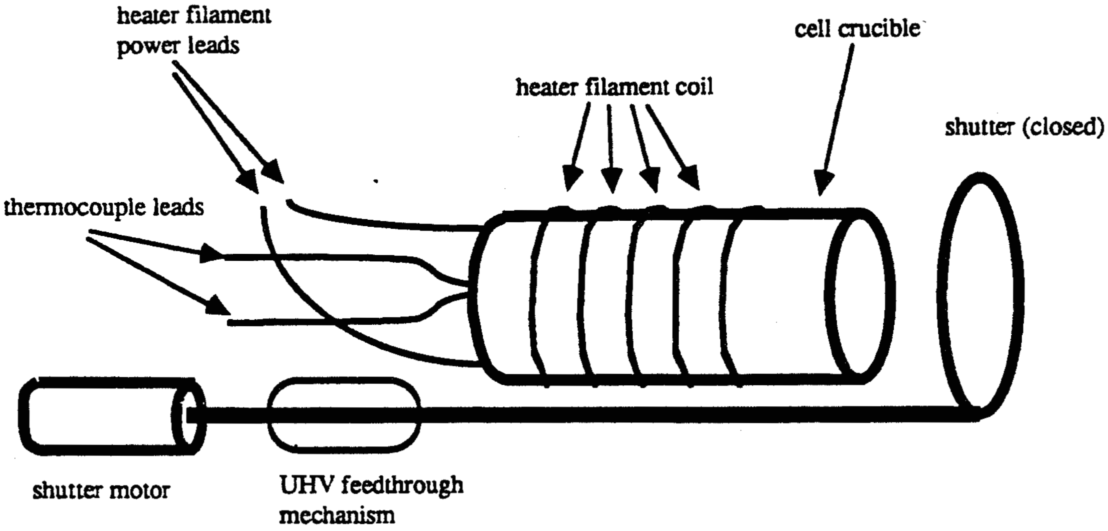


Figure 3. MBE Source Cell

Suppose, for example, that we measured an aluminum flux that was too low - clearly out of the bounds of normal variation - for the current temperature setpoint during a growth of AlGaAs. Can the use of CLIPS help here? It can - especially if the experimenter currently running the machine is relatively inexperienced, and thus not sure of all the system's possible behaviors. This situation is analyzed by an MBE expert in the following manner:

- Is the temperature controller power reading too high for the established setpoint? If so, there is probably a partial break in the filament. This is easily checked by measuring the resistance across the leads to the cell heater filament.

- Is the controller power reading too low? The thermocouple setting on the temperature controller may be wrong. This is also easily checked and corrected by using the front panel keys on the temperature controller. If this is not the problem, then the thermocouple may have shifted position and be touching or close to the filament. This can only be checked by removing the cell, which exposes a UHV chamber coated with arsenic dust to the air. This means full clean-room gowns and masks for all personnel while the inspection is made, and another lengthy bakeout period to restore the chamber to operation. This is the least desirable option. The optimum action is to attempt to change the cell setpoint until the desired flux is measured, ignoring the temperature measurement, and continue the experiment.

• Is the controller power reading normal? There may be several causes. The shutter may not have moved fully out of the way and is partially blocking the cell aperture. This is easily checked through a chamber inspection port, and is fixed by adjusting the cell motor position. If the shutter position is correct, then the problem may be a cracked cell, caused by thermal stress as discussed previously. When this happens, liquid aluminum flows out of the cell and onto the filament and chamber wall. The determination for this is to look for a filament shorted by the spilled aluminum. This can be detected by looking at the power supply - is the current very high and the voltage correspondingly low? If not, the cell may simply be empty - all the aluminum has been used. Unfortunately, there is no way to tell with the chamber closed. Repairing either of the last two problems, of course, requires opening the chamber, with all the problems mentioned above.

As seen above, the condition we could describe as (flux Al low) can have a number of causes and remedies of widely varying complexity. The value of an expert system here is that this knowledge can be codified quite nicely for entering onto the system, so it can deal with the contingency competently. The system could notify the operator and ask for the results of the non-intrusive checks above, and make a recommendation. If running unattended, the system could halt growth of the AlGaAs sample, cover it with a "buffer layer", and proceed with some other useful material (e.g., GaAs) that did not require use of the aluminum cell.

8. Future Applications of CLIPS to MBE Projects

There are some important uses for CLIPS-using MBE control software waiting in the very near future. One is the use of the expert system to analyze RHEED data. As discussed before, RHEED is the primary analytical "real-time" tool for assuring proper epitaxial growth of a sample. There are two main types of RHEED data: one is the counting of layers deposited during the growth process. This information has been successfully extracted with a computer at SVEC by taking the Fourier transform of the oscillations of diffracted RHEED beam brightness[3]. The other application, use of the actual diffraction-pattern geometry to determine growth modes, will require the integration of pattern recognition and image analysis tools with the expert system to successfully implement on the computer[4].

Successful incorporation of these RHEED techniques into a CLIPS-using epitaxy control system will greatly enhance the effectiveness of a much more ambitious project, the Wake Shield Facility (WSF) described in the Introduction. The Wake Shield Facility, currently under construction in Houston, is a circular platform about four meters in diameter which will be carried in the Shuttle Orbiter payload bay and deployed by the Remote Manipulator Subsystem arm. The platform has a circular shield which faces the direction of orbital motion, pushing aside the incident gas particles which exist at an ambient pressure of about 10^{-8} torr. Since the orbital speed of the platform is greater than the thermal speed of the ambient particles, a low-pressure wake of approximately 10^{-14} torr total pressure is formed behind the shield.

The wake side of the WSF contains the epitaxial growth facility, consisting of a rotating tray ("carousel") of prepared substrate samples, effusion sources (cells) and associated shutters. Monitoring equipment includes, as on the ground-based facilities already discussed, ionization gauges, mass spectrometers, a RHEED system, plus various auxiliary experiments. An 8086-based computer on the Wake Shield will carry out the process sequencing. For the first two flights, all analysis will be done on remote computers via telemetry from the WSF, but the system will then be tested as a free-flying facility which must be able to operate autonomously for days at a time. If successful, this will be the precursor to larger production platforms, operating up to six months at a time while turning out hundreds of ultra-high-quality epitaxial wafers. Such facilities will obviously need a high degree of robust expert control. The use of CLIPS for MBE in the laboratory will provide the development and testing necessary to provide that control.

9. Closing Remarks

We have seen that molecular beam epitaxy is a technology that is well-suited for a control software system using CLIPS as a top-level expert consultant. MBE has a number of well-defined problems which require more expertise than broad knowledge or problem solving to master. Additionally, MBE growth is a slow process which definitely benefits from having a machine take over the task from human researchers, yet has computational loads low enough for CLIPS to be invoked frequently on a 80286-class computer controlling the experiment.

The epitaxial control software at SVEC will integrate CLIPS into a C-language version of a currently-operational Turbo Pascal software package. This will be able to perform standard epitaxial processes in stand-alone mode while dealing flexibly with a fairly broad range of system fluctuations and faults. With the expertise of several MBE researchers at SVEC gradually built up into the system, it will also provide useful training for new personnel at the laboratory, as it has the ability to guide them through the experimental process. The development of CLIPS-using control software at SVEC will eventually lead to use in other facilities, including potentially other MBE research centers as well as the Wake Shield orbital MBE facility.

References

1 Marian Herman and Helmut Sitter. *Molecular Beam Epitaxy: Fundamentals and Current Status*, Springer-Verlag, Berlin: 1989.

2 Chris Culbert. *CLIPS Reference Manual, Version 4.3*, Artificial Intelligence Section, NASA-Lyndon B. Johnson Space Center: July 1989.

3 Jay S. Resh. *The Use of Reflection High-Energy Electron Diffraction for Molecular Beam Epitaxy*, Master's thesis, Department of Physics, University of Houston: August, 1989.

4 Patrick H. Winston. *Artificial Intelligence*, second edition, Addison-Wesley, Reading, Massachusetts: 1984.

AUTHOR INDEX