A Lisp-Ada Connection

Allan Jaworski, David Lavallee, David Zoch

Ford Aerospace & Communications Corporation
College Park, MD 20740

OVERVIEW

The Ford Lisp-Ada Connection (FLAC) is an expert system generation tool designed to support direct entry of knowledge by experts in a Lisp machine environment and downloading to an inference engine which has been implemented in the Ada programming language. FLAC consists of two subsystems, the Knowledge Editor Graphics System (KEGS) and the Ford Ada Inference Engine (FAIE).

Knowledge is entered through KEGS, an easily learned knowledge base CAD system which provides integrated features for rule development and knowledge base testing. An expert can use a set of menu- and mouse- driven resources to develop a knowledge base which is graphically represented. Tools are provided for the expert to rapidly enter, test, and debug knowledge base logic paths. The user interface is similar to those found in CAD systems for electrical circuit design.

The knowledge base can then be downloaded to FAIE, an extremely fast portable Ada-based inference engine which is capable of firing up to 700 rules per second on an MC68000 or 1500 rules per second on a VAX 11/780. The inference engine is written in the Ada programming language and supports both forward and backward chaining modes of inference. The FAIE run-time environment has been previously used in a prototype of the Space Station Operations Management System.

FLAC currently runs on a Symbolics 3640 and a VAX 11/780 VMS processor connected via DECnet/Ethernet. KEGS has been implemented in Symbolics ZetaLisp and FAIE was originally implemented in Telesoft Ada on an Intellimac MC68000-based system and then ported to DEC VAX/VMS Ada.

EXPERT SYSTEMS IN THE SPACE STATION INFORMATION SYSTEM

Future space information systems must be automated to the fullest extent possible both to support crew and operations staff efficiency and to allow us to maintain spacecraft and instrument safety in the face of increasingly complex systems and operational requirements. The Space Station Information System is a prime example of these enhanced requirements. One of the key engineering guidelines for the

Space Station is that it should be able to carry out normal operations for some finite period of time without contact with the ground. As pointed out in a NASA Technical Memorandum on Automation Technology For The Space Station,

> "Expert systems are needed to perform many monitoring and control functions requiring complex status analysis and automated decision making so that the Station is less dependent on ground support in these areas."

Also in the same document:

> "In emergency situations, automated systems which respond very rapidly to a crisis can bring the system to a fail-safe condition before extensive damage occurs... Without automation, humans may be placed more often in pressure-prone situations such as EVA and emergency maintenance in which there is an increased chance of error."

Expert systems could incorporate fault diagnosis, isolation, and recovery to enhance crew safety. Alarms could be triggered automatically to warn crew members of hazardous situations. In addition, many faults could be corrected before they pose any danger to the crew or spacecraft.

Lisp versus Ada

Ada has been baselined as the programming language for the configuration-managed software associated with the SSIS. A lively debate in the software community has centered on the respective strengths of Lisp and Ada as languages for the implementation of expert systems. Lisp is generally regarded as a "hacker's" language. Lisp and its development environments support highly interactive modes of software implementation where requirements specification, design, implementation, and testing are mixed in a process typically characterized as "prototyping". Ada stems from a heritage of disciplined development and configuration management practices of the sort usually practiced by both NASA and the Department of Defense. Neither mode is a totally satisfactory approach to the development of quality products which fully meet user needs. We cannot tolerate undisciplined development of mission critical systems, yet many of our systems are developed with an inflexible approach that does not fully meet user requirements.

At a more fundamental level Lisp and Ada differ as programming languages. Lisp is a weakly typed language with characteristics that even obscure the distinction between data and programs (both are trees of Lisp atoms). Ada on the other hand requires compile-time checking of type compatibility and does not allow the passing of procedures as arguments. Lisp is commonly used as an interpreted language while the design of Ada (particularly the large volume of compile-time checking) strongly restricts it to compiled implementations.

Lisp and Ada do share certain common characteristics. Object-oriented design is a common theme and some of the best implementation work done in both languages uses object-oriented methodologies. Both languages support structures which facilitate the direct implementation of object oriented design. The Rational Ada development environment borrows many Lisp machine concepts to radically improve productivity of Ada programmers. The Common Lisp notion of a package provides Lisp programmers with modular capabilities similar to those found in the Ada language.

An Integrated Approach

Broadly speaking, Lisp environments are ideal for the prototyping and development of user interfaces. Ada environments are ideal for the development of large software systems with critical reliability and maintenance requirements. An approach currently being evolved at Ford Aerospace is the integrated use of both languages in a networked environment. Lisp is used to construct an extremely friendly knowledge editor which supports the development and testing of knowledge bases which are downloaded to an Ada inference engine that operates in real-time fully independently of the knowledge editor. This paper describes the features of the Ford Lisp-Ada Connection (FLAC), a prototype which combines both systems into a coherent real-time expert systems development environment.

THE FLAC ENVIRONMENT

Figure 1 shows the overall structure of FLAC. In its current implementation an expert uses a mouse-menu interface to enter knowledge through a graphics-oriented editor. He can exercise test cases and directly observe the behavior of the knowledge base which is graphically represented. He can trigger downloading of knowledge bases to a remote computer system and directly observe the execution of applications programs which use the real-time inference engine through remote terminal services. Windowing services on the Offline system allow simultaneous knowledge entry, debugging, and observation of remote applications. The current implementation environment of FLAC is a a Symbolics Lisp machine (Offline System) linked by DECnet/Ethernet to a VAX 11/780 (Online System). We describe the two major components of FLAC, the Ford Ada Inference Engine (FAIE) and the Knowledge Editor Graphics System (KEGS).

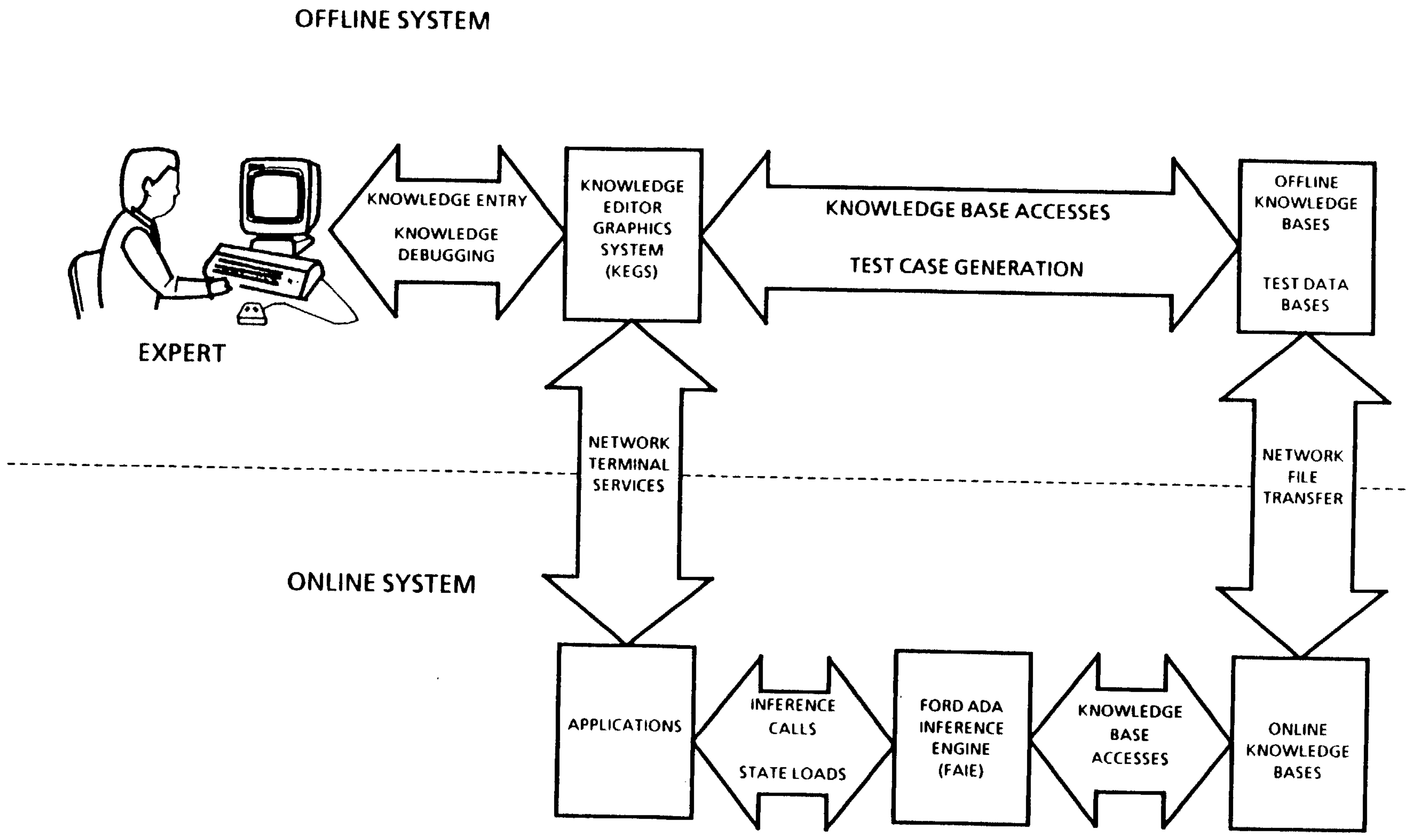


Figure 1. Ford Lisp - Ada Connection (FLAC)

GRA[illegible]ICS-SS031

The Ada Inference Engine

The Ford Ada Inference Engine (FAIE) is a prototype expert system inference engine designed to execute as an Ada task embedded in an expert system which could in turn be embedded in a larger program. The sample application discussed here involves using FAIE for fault diagnosis. A typical rule in this type of system might be:

"IF temperature is above normal and
heater output is above normal,
THEN power off heater."

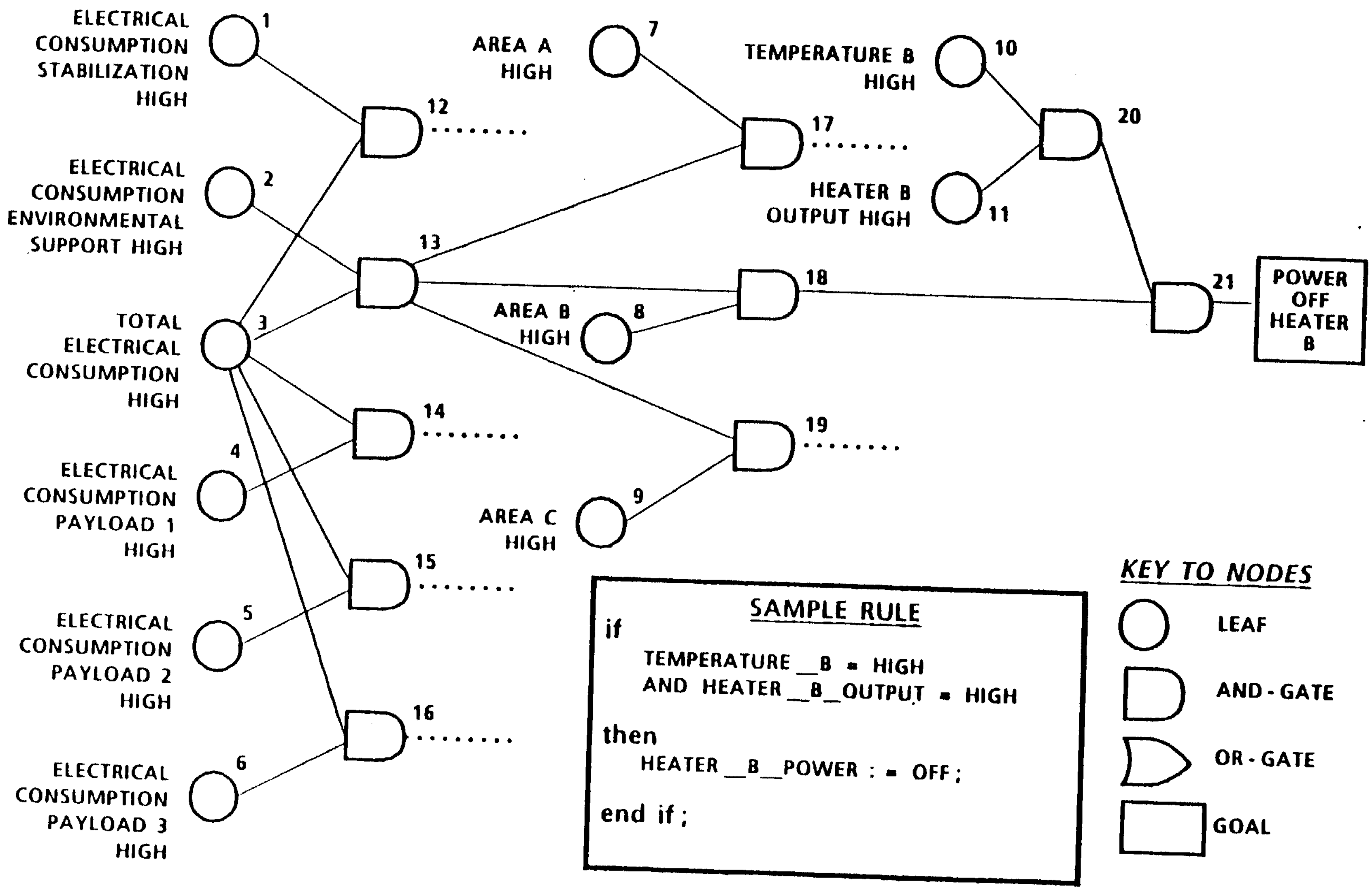
The knowledge base is structured as a directed acyclic graph. This can be thought of as a network of nodes with the links all pointing in the same direction. For the diagnostic system, the leaf nodes on one side of the graph represent the various sensor data measurements. Commands for corrective action are the goal nodes on the other side of the graph. The relationships between erroneous measurements are the intermediate nodes leading to a goal. Figure 2 shows a portion of a sample graph. Note: the dotted lines represent additional portions of the graph that are not shown.

The leaf nodes represent initial data points that must be provided to the inference engine. The nodes on the other side of the graph represent goal states that are sought when executing the inference engine. The nodes in between represent hypotheses or subgoals that will be tested. The links between the nodes are the "production rules" that the inference engine uses to traverse the graph.

Since we have a compiled, static knowledge base, all elements are present in the graph. Each node has a status which we will refer to as "flagged", "unflagged", or unknown. A "flagged" node is one that satisfies its associated IF-THEN rule. We must distinguish between an untested node (status equals unknown), and a node that was tested and does not satisfy the associated IF-THEN rule (status equals "unflagged"). A "flagged" node is one that will be used to traverse the graph. The path to a goal must be continuous through "flagged" nodes. An "unflagged" node represents a "dead end".

Status for all the leaf nodes is passed to the inference engine when a problem exists. Figure 3 shows the sample knowledge base with all the leaves (nodes 1-11) given an initial status. Nodes 2,3,10 and 11 are "flagged".

In an attempt to find a goal as quickly as possible, the successors of the first "flagged" leaf node are examined and the first one in the list is visited using Ada procedure FORWARD_CHAIN. Since the status of the successor node is initialized to unknown, its predecessors are examined along


ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
4
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
6
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
12
13
14
15
16
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
17
18
19
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND - GATE
OR - GATE
GOAL
SAMPLE RULE

```
if
    TEMPERATURE__B = HIGH
    AND HEATER__B__OUTPUT = HIGH
then
    HEATER__B__POWER : = OFF;
end if;
```

Figure 2. Sample Compiled Knowledge Base

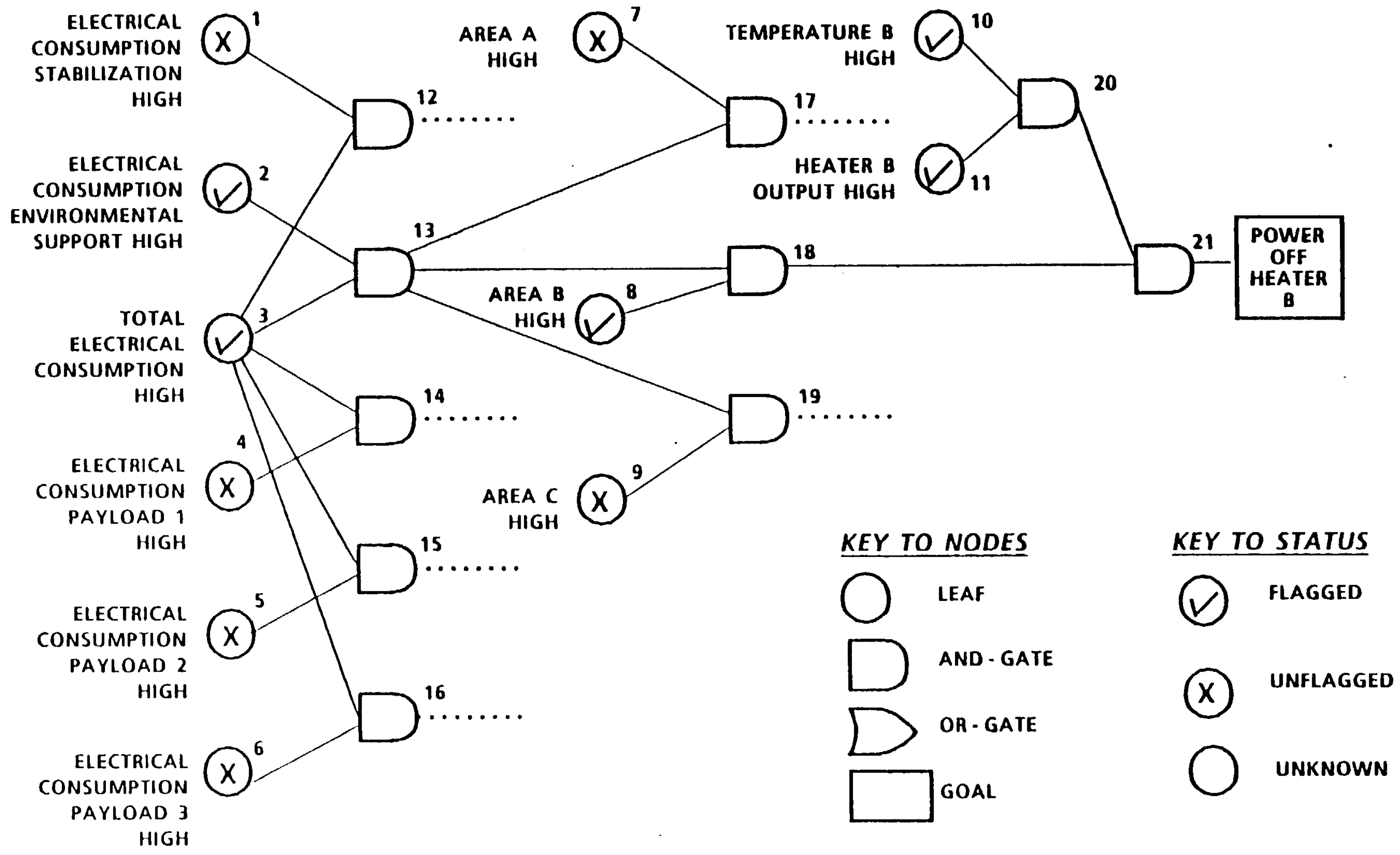

ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
4
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
6
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
12
13
14
15
16
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
17
18
19
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND - GATE
OR - GATE
GOAL
KEY TO STATUS
FLAGGED
UNFLAGGED
UNKNOWN

Figure 3. Knowledge Base - Initial Problem State

with its AND/OR flag to determine its status. If the status of this first successor to the first leaf node is found to be "flagged", then its first successor in its list is visited, and so on until a goal is found or a dead end is reached. If the status of this first successor is found to be "unflagged", then the next successor in the first leaf node's list is visited.

If the status of a predecessor node is unknown, then Ada function BACK_TRACK is invoked to return the status. Both subprograms FORWARD_CHAIN and BACK_TRACK are recursive.

Figure 4 shows the resulting status after running the inference engine. To get to Figure 4 from Figure 3 the following steps were taken:

1. Node 2's successor list is examined, and node 13 is passed in a call to FORWARD_CHAIN.
2. Since node 13 is an "and gate" and both its predecessors (2 and 3) are "flagged", node 13 becomes "flagged".
3. Node 13's successor list is examined, and node 17 is passed in a recursive call to FORWARD_CHAIN.
4. Since node 17 is an "and gate" and node 7 is "unflagged" node 17 becomes "unflagged".
5. FORWARD_CHAIN returns to visiting node 13, where the successor list is examined, and node 18 is passed in another recursive call to FORWARD_CHAIN.
6. Since node 18 is an "and gate" and both its predecessors (8 and 13) are "flagged", node 18 becomes "flagged".
7. Node 18's successor list is examined, and node 21 is passed in another recursive call to FORWARD_CHAIN.
8. Since the status of node 20 is unknown, node 20 is passed in a call to BACK_TRACK.
9. Since node 20 is an "and gate" and both its predecessors (10 and 11) are "flagged", node 20 is "flagged" and BACK_TRACK returns.
10. Since node 21 is an "and gate" and both its predecessors (18 and 20) are "flagged", node 21 is "flagged" and a goal has been found.
11. The recursive calls return and visit other successor nodes for additional goals.

In practice FAIE is capable of exceedingly fast performance. As implemented on a nonvalidated version of the Telesoft Ada compiler FAIE supported a rule firing rate of 700 rules per second on a Motorola MC68000 processor. On its current VAX 11/780 implementation rule-firing occurs at the rate of 1500 rules per second. For small expert systems this rate is more than adequate to achieve real-time performance. Moreover, the maximum search time for a goal can easily be computed from the characteristics of the knowledge tree. Future versions of FAIE which take advantage of multiprocessing implementations of Ada run-time systems will be even more powerful.

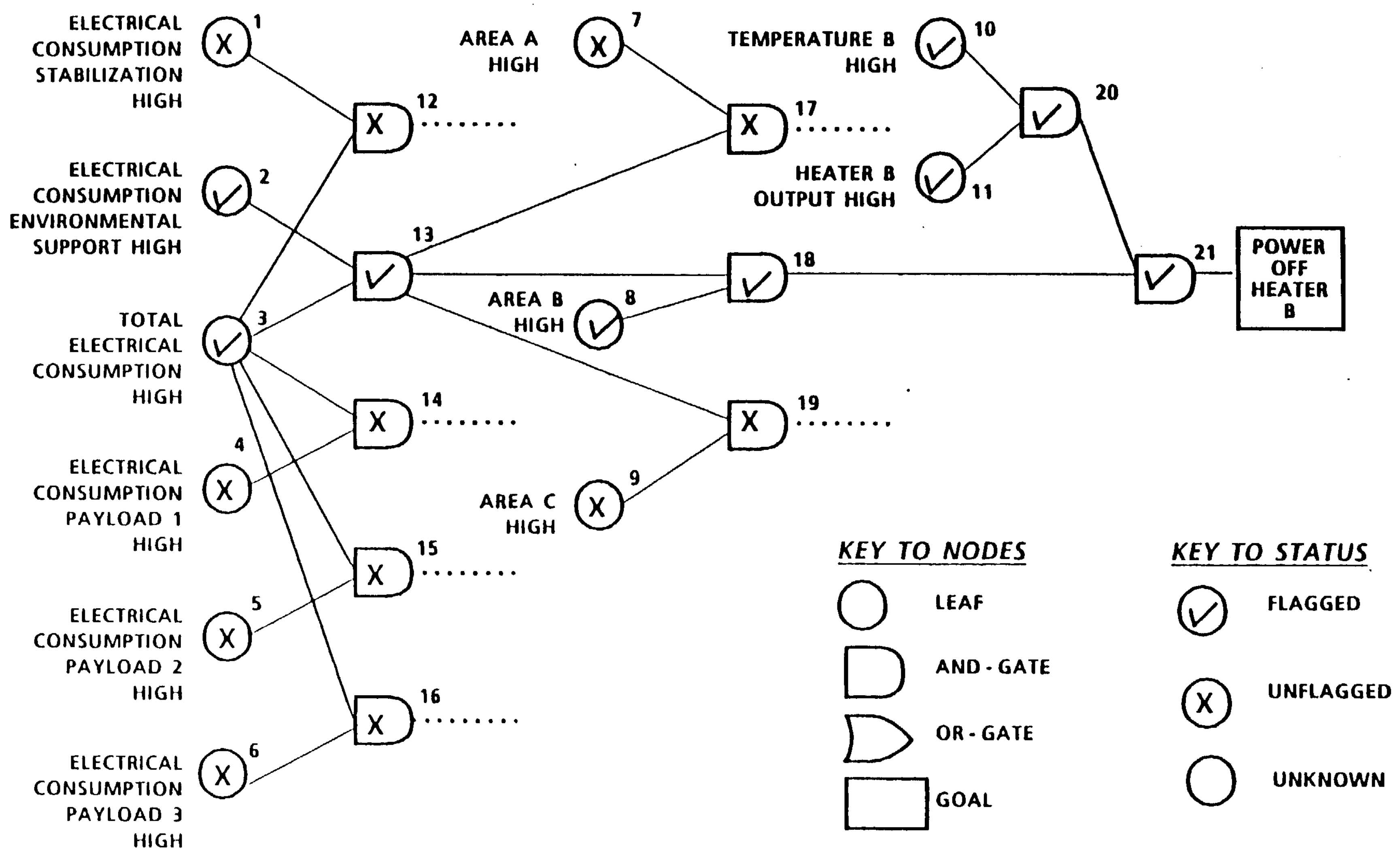

ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
4
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
6
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
12
13
14
15
16
17
18
19
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND-GATE
OR-GATE
GOAL
KEY TO STATUS
FLAGGED
UNFLAGGED
UNKNOWN

Figure 4. Knowledge Base - Problem Solution

The Knowledge Editor

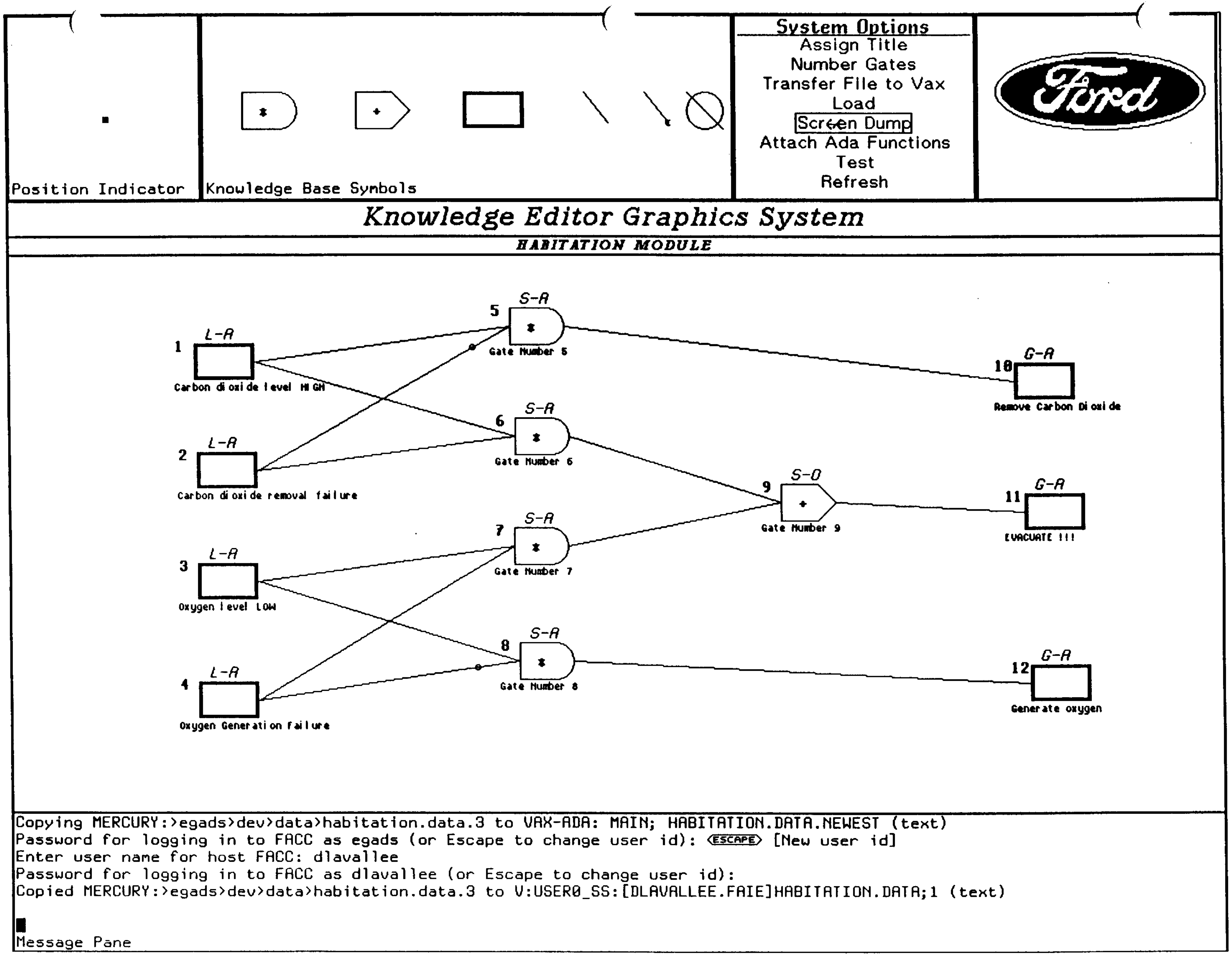
The Knowledge Editor Graphics System (KEGS) is a simple CAD-oriented system for the direct entry of knowledge by experts. Knowledge entry is accomplished by using a mouse to draw logic gates, leaves, and goals. Figure 5 is a sample screen from a KEGS session. It can be seen that the screen representation of the knowledge base is nearly identical to the internal representation of the knowledge base. This greatly simplifies design and eliminates need for time-consuming conversions of knowledge base formats. Moreover, since the representation is identical to that used in electrical circuit design it is nonintimidating to a large class of experts, the hardware engineers who design spacecraft systems.

Operation of KEGS is intuitive. The expert simply touches the cursor to the symbol to be drawn and places as many as required on appropriate screen locations. Connections are drawn by touching the line icon and then touching pairs of symbols in sequence. To delete an icon or connection the expert touches the cursor to the circle with the diagonal line through it and then touches the symbol to be deleted. Connections to that symbol are then automatically deleted. Ada functions to test leaf conditions or execute goal conditions are attached through a simple fill-in-the-blanks menu. These functions can be either user-written functions or selected from a library of system-provided functions (e.g. test a variable for range, send a command). Current capabilities to test the data base are limited to manually marking the leaf nodes and observing goal firing. Future plans call for inclusion of automatic test and tracing functions. Once the data base is complete and tested, the expert system knowledge base can be downloaded through the network to be executed by the runtime environment.

RESEARCH DIRECTIONS

The primary focus of our work is the use of multitasking to improve performance. This will also solve the problem of reading dynamic data which is constantly being updated as inferencing is in progress. It seems reasonable to use Ada tasking to enhance the real-time performance of inference engines. Although true production-quality multiprocessing Ada compilers do not yet exist, it is now feasible to write tasking implementations of inference engines which will exhibit order-of-magnitude improvements in rule-firing rates when ported to true multiprocessing Ada environments.

Douglass lists five levels of potential parallelism in rule-based expert systems. They are: subrule level, rule level, search level, language level, and system level. These levels include different types within them. Douglass concentrates on rule level and various types of search level parallelism. He gives a range of quantitative results for



```
Copying MERCURY:>egads>dev>data>habitation.data.3 to VAX-ADA: MAIN; HABITATION.DATA.NEWEST (text)
Password for logging in to FACC as egads (or Escape to change user id): ESCAPE [New user id]
Enter user name for host FACC: dlavallee
Password for logging in to FACC as dlavallee (or Escape to change user id):
Copied MERCURY:>egads>dev>data>habitation.data.3 to V:USER0_SS:[DLAVALLEE.FAIE]HABITATION.DATA;1 (text)
```

Figure 5. Sample KEGS Screen

these levels using mathematical models and concludes that combinations of subrule, rule and search level parallelism will yield better results than any single level when the characteristics of the specific system are taken into consideration. He also mentions that very little work has been implemented and tested on parallel computers.

Communication between processes is an important factor in the efficiency of parallel algorithms. Generally speaking, the more frequently that information is exchanged, the slower the computation is performed since processes spend a larger portion of their time communicating rather than computing. Researchers working on the DADO machine have developed some unique methods of communicating between parallel processors (e.g. a binary tree structure of processors with communication rules controlled by hierarchy).

In Ada, the task is the natural construct for parallel processing. However, multitasking involves considerable overhead in creating/activating tasks, communicating between them, and terminating them. This overhead must be compared with the amount of computation performed in parallel in order to determine the relative efficiency gained by various strategies of parallel processing. Gehani concurs, and goes on to say that in designing concurrent programs in Ada, one must avoid the polling bias in the communication mechanism. He also points out that multiprocessing programs will be more efficient if the underlying hardware offers genuine concurrency.

Deering also emphasizes that hardware considerations, especially processor speeds versus memory speeds, must be examined when designing the architecture of expert systems. He says one should "study hardware technology to determine at what grain sizes parallelism is feasible and then figure out how to make [the] compilers decompose programs into the appropriate-size pieces."

Granularity is the average amount of work done by a process between communication with other processes. It is inversely proportional to the frequency of communication. The five levels of parallelism mentioned by Douglass range from very finely grained to roughly grained. A fine grained approach was taken by Rude where each rule was itself declared as an Ada task with rendezvous for links to predecessors and successors. This concept has merit but is questionable for real-time applications. In the implementation of the PICON expert system for real-time process control, a roughly grained algorithm was chosen by segmenting parts of the knowledge base and applying priorities to searching the different portions. Our future investigations will include analyzing various strategies, including forward and backward chaining on individual rules in parallel, dividing the knowledge base, and combinations of the different strategies.

For the knowledge editor we expect to investigate modes of enhancing the overall debugging interface, including the development of tools for automated testing which allow the expert to explore the more critical logic paths of the system, particularly those logic paths which might lead to actions or recommendations which affect overall spacecraft safety.

A significant problem which we have encountered in the KEGS implementation is the binding of Ada procedures and functions to leaves and goals. Since Ada does not provide the capability to dynamically define new procedures or functions and pass them as procedure parameters we have been forced to limit the expert to the use of a previously defined library of procedures and functions (e.g. test to see if a variable is in range, send a command string). A more flexible approach would give the expert access to the Ada program library in order to implement procedures and functions when necessary. We are currently investigating the alternatives and implications for implementing these bindings.

Conclusions

The prototype demonstrates the feasibility of using Ada for expert systems and the implementation of an expert-friendly interface which supports knowledge entry. In the FLAC system Lisp and Ada are used in ways which complement their respective capabilities. Future investigation will concentrate on the enhancement of the expert knowledge entry/debugging interface and on the issues associated with multitasking and real-time expert systems implementation in Ada.

BIBLIOGRAPHY

David C. Brauer, Patrick P. Roach, Michael S. Frank, and Richard P. Knackstedt, "Ada and Knowledge-Based Systems: A Prototype Combining the Best of Both Worlds", Expert Systems in Government Conference, McLean, VA, October 1986

NASA Advanced Technical Advisory Committee, Advancing Automation and Robotics Technology for the NASA Space Station and for the U.S. Economy, NASA Technical Memorandum 87566/, Volume II, March 1985 p. 5.

Deering, M. "Architectures for AI", Byte Magazine, April, 1985.

Douglass, R., "Characterizing the Parallelism in Rule-Based Expert Systems", Proc. Hawaii International Conference on Systems Science, HICSS-18, Jan. 1985.

Douglass, R., "A Qualitative Assessment of Parallelism in Expert Systems", IEEE Software, May 1985, pp. 70-81.

Gehani, N., Ada: Concurrent Programming, Prentice-Hall Inc., 1984.

David B. Lavallee, "An Ada Inference Engine for Expert Systems", Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, TX, June 1986.

Moore, R., L. Hawkinson, C. Knickerbocker, L. Churchman, "A Real-Time Expert System for Process Control", First Conference on AI Applications, IEEE Computer Society Press, Dec. 1984.

Moore, R., "Adding Real-Time Expert System Capabilities to Large Distributed Control Systems", Control Engineering, April 1985.

Rude, A., "Translating a Research LISP Prototype to a Formal Ada Design Prototype", Proceedings of the Washington Ada Symposium, March 1985.

Stolfo, S., and D. Miranker, "DADO: A Parallel Processor for Expert Systems", Proceedings of the 1984 International Conference on Parallel Processing, IEEE Computer Society Press, August, 1984.

Stolfo, S., "Five Parallel Algorithms for Production System Execution on the DADO Machine", Proceedings of the NCAI, Austin, TX, 1984.