AN ADA INFERENCE ENGINE FOR EXPERT SYSTEMS

David B. LaVallee
Ford Aerospace and Communications Corp.
College Park, Maryland

1 INTRODUCTION

The purpose of this research project is to investigate the feasibility of using Ada for rule-based expert systems with real-time performance requirements. This includes exploring the Ada features which give improved performance to expert systems as well as optimizing the tradeoffs or workarounds that the use of Ada may require. A prototype inference engine for general purpose expert system use was built using Ada, and rule firing rates in excess of 500 per second were demonstrated on a single MC68000 processor.

The knowledge base uses a directed acyclic graph to represent production rules. The graph allows the use of AND, OR, and NOT logical operators. The inference engine uses a combination of both forward and backward chaining in order to reach goals as quickly as possible. Future efforts will include additional investigation of multiprocessing to improve performance and creating a user interface allowing rule input in an Ada-like syntax.

Some of the issues discussed concerning Ada's use in expert systems include: How should a knowledge base be structured in Ada? How should the knowledge base be searched, especially in the context of a dynamic problem space with new data constantly entering the system? Can real-time performance be achieved?

A critical issue involves the use of Ada's multitasking to implement parallel algorithms in expert systems. Clearly the inference engine can be implemented as a single task which can be integrated into a larger system and execute only when necessary. However, the execution of the inference mechanism in a parallel manner should increase performance. Using segmented knowledge bases, backward chaining in parallel on all goals at once, and forward chaining in parallel on individual rules are some of the different strategies to be considered. These strategies use different levels of granularity. Using an algorithm with a low level of granularity, fewer parallel computations will be performed and intertask communication will be less frequent. Using a high level of granularity, much computation is done in parallel, however it involves considerable intertask communication. The overhead involved

in creating tasks and in communicating between them, must be weighed against the benefits of the parallel performance.

2 EXPERT SYSTEM USE IN THE SPACE STATION

The Space Station will be a tremendously complex system. The automation of many of the Space Station activities and related monitoring functions in a safe and reliable manner will help to increase the efficiency and cost effectiveness of the system. In addition, 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 [1],

> "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 [1],

> "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.

3 FORD ADA INFERENCE ENGINE

3.1 Description

The Ford Ada Inference Engine (FAIE) is a research 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 1 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 2 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 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.

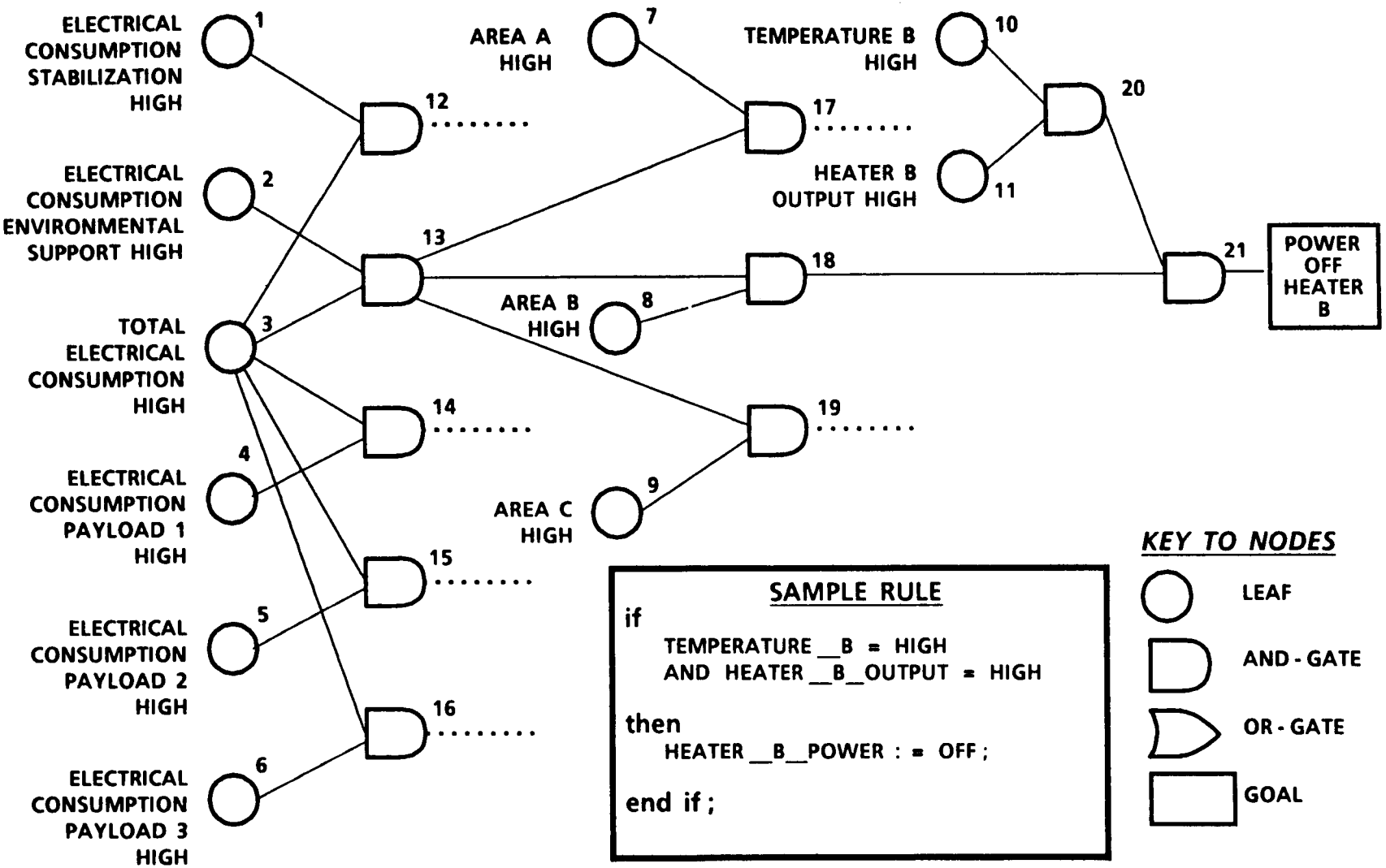

Figure 1. Sample Compiled Knowledge Base

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 3 shows the resulting status after running the inference engine. To get to Figure 3 from Figure 2 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.

3.2 Performance

The search speed is dependent upon the depth of the graph from leaf to goal but is independent of the number of leaves or goals in the graph. The only rules that are attempted to be matched already have at least one element of its left-hand-side "flagged". When a goal node is "flagged", the inference engine will issue a procedure call or task rendezvous to invoke logic associated with the goal state (e.g. turn a circuit on or off).

Neither heuristic pruning nor optimal search techniques are employed. Some control over program execution can be accomplished by ordering the leaf nodes and/or ordering the list of successors and predecessors. Factors such as severity of problem or frequency of occurrence can be used to prioritize these lists.

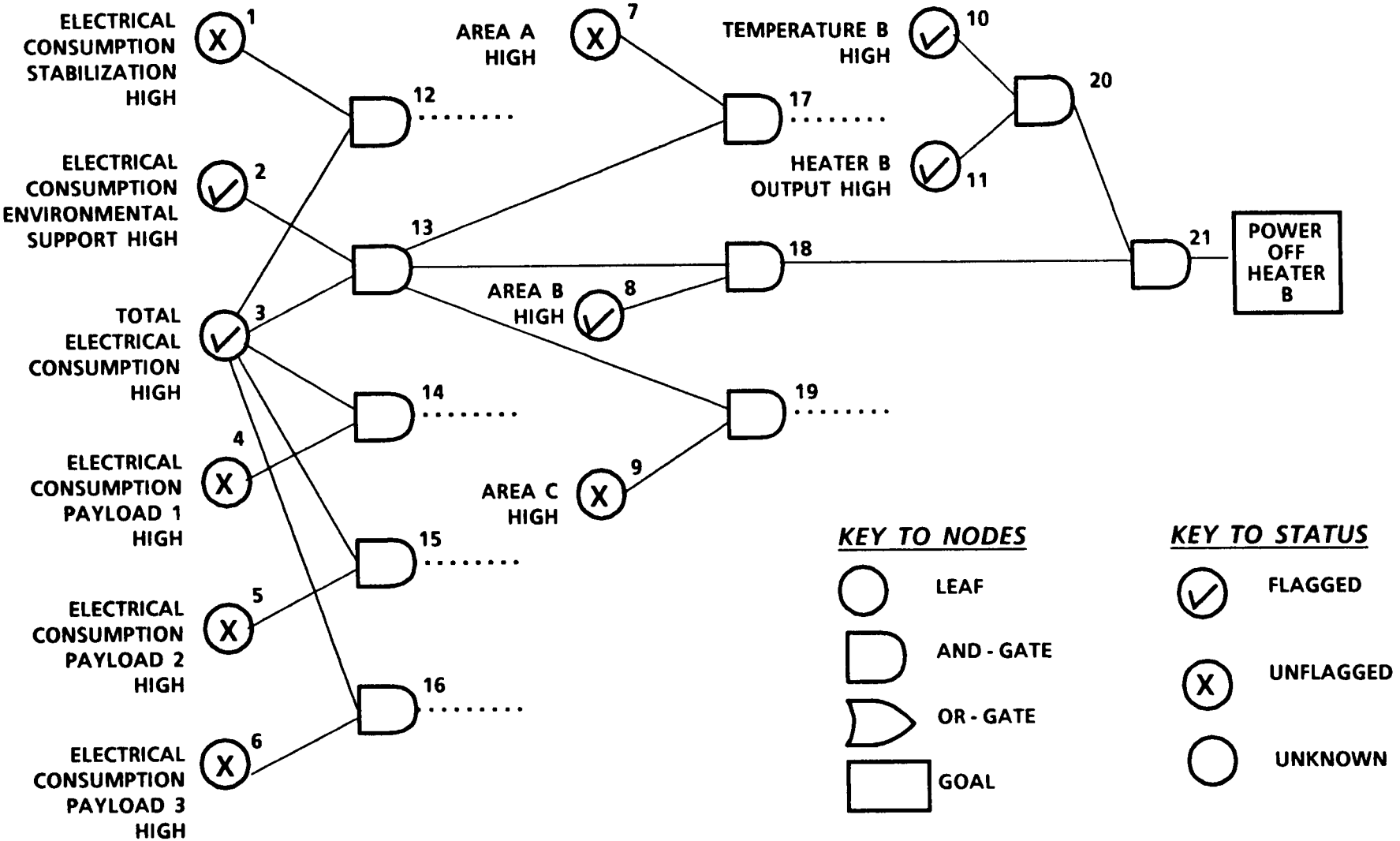

Figure 2. Knowledge Base – Initial Problem State

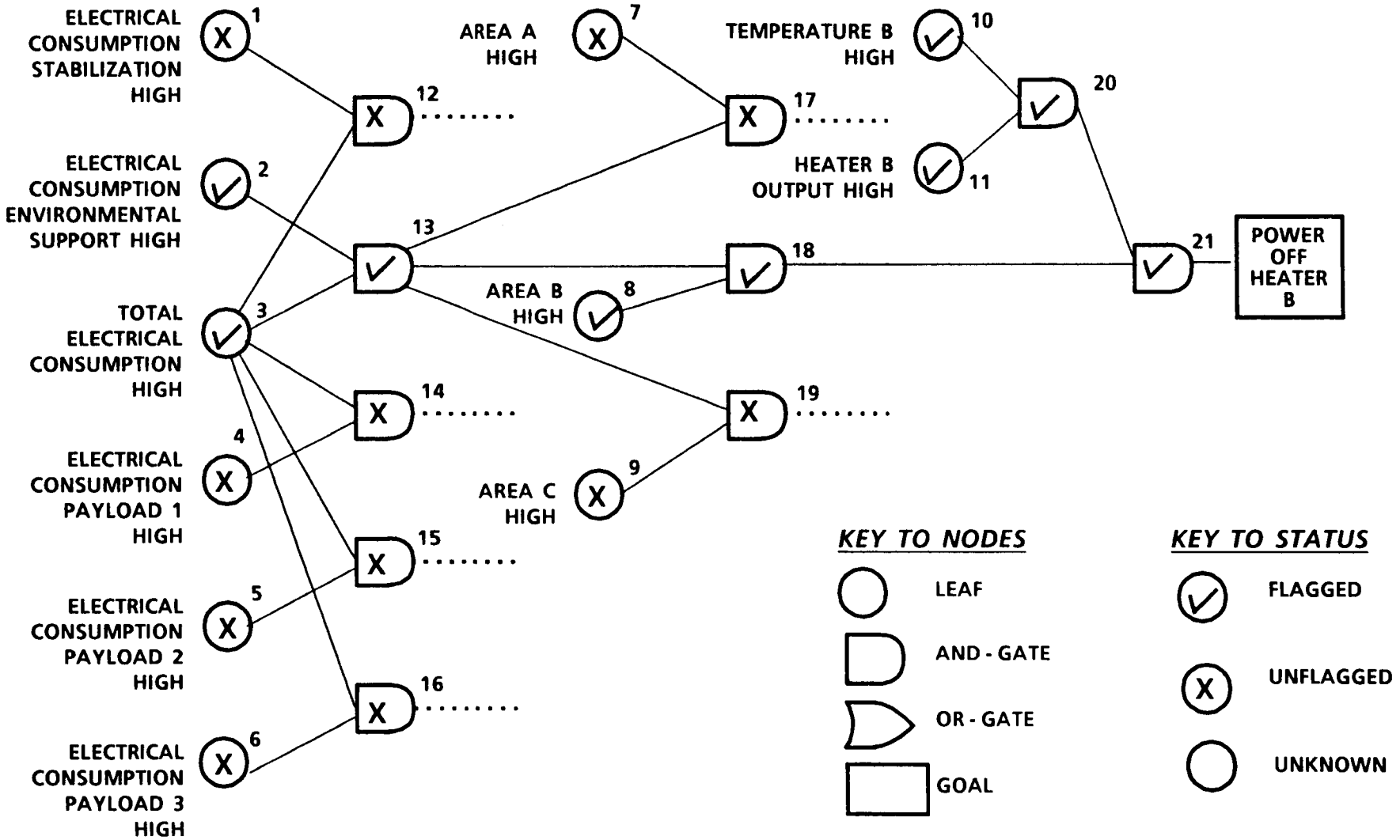

Figure 3. Knowledge Base – Problem Solution

This design assumes that all calculations on the data are performed up front, prior to invoking the inference engine. Speeds in excess of 500 rule firings per second were executed on a single processor. A rule firing is defined to be "flagging" a node, increasing working memory. This is similar to results obtained by other non-LISP inference engines (e.g. OPS83 or the BLISS version of OPS5). These results indicate that real-time performance is achievable.

4 USE OF ADA FEATURES

The knowledge base is an array of records. Each record is a node with the following information:

```
STATUS - UNKNOWN, FLAGGED or UNFLAGGED
FORM - LEAF, SUBGOAL or GOAL
AND_OR_FLAG - AND or OR
POINTER TO PREDECESSOR LIST
POINTER TO SUCCESSOR LIST
TEXT STRING IDENTIFIER
```

The Ada package describing the data types in the knowledge base is given in Figure 4. A description of Ada constructs used to transform LISP research prototype expert systems into Ada production systems was given by Rude [2]. Unlike Rude, I have implemented the predecessor and successor lists as linked lists of records using access types rather than arrays of records. This allows flexibility in dynamically altering the knowledge base at runtime, e.g. if a sensor is determined to be faulty and you wish to ignore its input. In addition, the minimum amount of storage space is used. Using arrays would require that all nodes allocate space for the largest list of predecessors or successors and would also require re-compilation to adjust the maximum sizes.

Ada tasking was used to embed the expert system in a larger Ada program. It can stand idle while other monitoring and limit checking functions are performed and then spring into action when an anomaly is detected. A more extensive use of tasking can be made to perform various functions of the expert system in parallel. This will be discussed in the next section.

Although Ada provided adequate constructs to build this inference engine there are a couple of features of other languages (notably LISP languages) that would be very useful for expert systems if supported in Ada. The main feature desired is the ability to pass Ada functions as parameters in subprogram calls. An alternative would be the ability to embed a function in a data structure, such as the field of a record, to be executed when accessed. This could be used to

perform calculations when needed. As mentioned earlier, in this version, all calculations needed to execute the inference engine must be performed up front.

The second desired feature is the ability of an object to inherit values from a parent. For example, when new elements are added to a linked list or tree-like structure, they could inherit values in specified fields of their parents. This would reduce subprogram calls and a number of extra objects for data storage.

5 FUTURE INVESTIGATION

5.1 Further Multitasking Work

One main thrust of our further work will focus on 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 [3,4] 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 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 [5,6] 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 [7] concurs, and goes an 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 [8] 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 [2] 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 [9,10], 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.

5.2 User Interface

Another area for future work involves building a user interface for accurate and efficient knowledge acquisition. The accumulation of the domain knowledge and its insertion into a knowledge base has often been a bottleneck in expert system production. The Ada language IF-THEN-ELSE constructs are readable and English-like. We will build a user interface in an Ada syntax that is hopefully both easy for the knowledge engineer to use, and also easily translates into Ada code.

6 CONCLUSION

The prototype demonstrates the feasibility of using Ada for expert systems on a small scale. Investigation of multitasking and alternate knowledge base representations will help to analyze some of the performance issues as they relate to larger programs.

References:

1. 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.

2. Rude, A., "Translating a Research LISP Prototype to a Formal Ada Design Prototype", Proc. Washington Ada Symposium, March 1985.

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

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

5. Stolfo, S., and D. Miranker, "DADO: A Parallel Processor for Expert Systems", Proc. 1984 Int. Conf. on Parallel Processing, IEEE Computer Society Press, August, 1984.

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

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

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

9. Moore, R., L. Hawkinson, C. Knickerbocker, L. Churchman, "A Real-Time Expert System for Process Control", 1st Conf. on AI Applications, IEEE Computer Society Press, Dec. 1984.

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

```
with DYNAMIC_STRING;
package GRAPHS is
  type NODE_NUM is new INTEGER range 0..INTEGER'LAST;

  type STATUSES is (FLAGGED, UNFLAGGED, UNKNOWN);
  type GATE is (AND_GATE, OR_GATE);
  type NODE_FORM is (GOAL, SUBGOAL, LEAF);

  type PRED_NODE;          -- DATA STRUCTURE FOR LINKED LIST
                           -- OF PREDECESSORS
  type PRED_NODE_PTR is ACCESS PRED_NODE;
  type PRED_NODE is record
    NAME : NODE_NUM;
    NEG_LOGIC_FLAG : BOOLEAN := FALSE;
              -- FALSE = want pred to be flagged.
              -- TRUE = want pred to be unflagged.
    NEXT : PRED_NODE_PTR;
  end record;

  type SUCC_NODE;          -- DATA STRUCTURE FOR LINKED LIST
                           -- OF SUCCESSORS
  type SUCC_NODE_PTR is ACCESS SUCC_NODE;
  type SUCC_NODE is record
    NAME : NODE_NUM;
    NEXT : SUCC_NODE_PTR;
  end record;

  type NODE is record      -- DATA STRUCTURE FOR A NODE OF
                                        --    THE GRAPH
    STATUS    : STATUSES := UNKNOWN;
    AND_OR    : GATE  := AND_GATE;
                      -- AND means all predecessors must
                      --    be satisfied.
                      -- OR means one or more predecessors
                      --    must be satisfied.
                      --  Does not apply to leaf nodes.
    PRED      : PRED_NODE_PTR;
    SUCC      : SUCC_NODE_PTR;
    FORM      : NODE_FORM;
    MESSAGE   : DYNAMIC_STRING.UCSD_STRINGS;
  end record;

  type KNOWLEDGE_BASE is array (NODE_NUM range <>) of NODE;
                       -- ARRAY OF RECORDS
  type FLAGGED_NODES is array (INTEGER range <>)
            of NODE_NUM;                    -- Init. state
  function SIZE return INTEGER; -- ALLOWS SIZE OF GRAPH TO
                                -- BE READ AT RUN TIME.
end GRAPHS;
```

Figure 4. Graphs Package