

A Rule-Based Inference Engine which is Optimal and VLSI Implementable

N. L. Griffin and F. D. Lewis

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506

ABSTRACT

An inference engine for rule based expert systems which forms part of the EXPRES system is developed and presented. It is shown to be universal, correct, and optimal with respect to time. In addition, a VLSI implementation of the system is proposed which allows automatic design of universal as well as special purpose expert systems on a chip.

Introduction.

Expert systems have become a popular method for representing large bodies of knowledge for a given field of expertise and solving problems by use of this knowledge. An *expert system* often consists of three parts, namely: a knowledge base, an inference engine, and a user interface

A dialogue is conducted by the *user interface* between the user and the system. The user provides information about the problem to be solved and the system then attempts to provide insights derived (or inferred) from the knowledge base. These insights are provided by the *inference engine* after examining the knowledge base. This interaction is illustrated by the picture in figure 1.

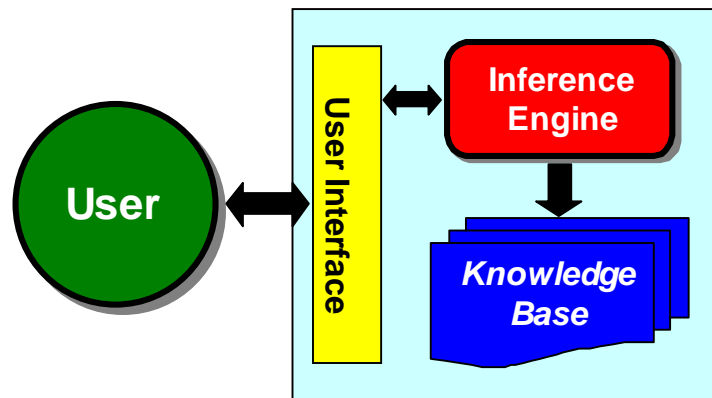


Figure 1 - An Expert System

Knowledge bases consist of some encoding of the domain of expertise for the system. This can be in the form of semantic nets [Qu68], procedural representations [WiT75], production rules [DBS77], or frames [BW77]. We shall consider only production rules for our knowledge base. These *rules* occur in sequences and are expressions of the form:

if *<conditions>* then *<actions>*

where if the *conditions* are true then the *actions* are executed.

When rules are examined by the inference engine, actions are executed if the information supplied by the user satisfies the conditions in the rules. Two methods of inference often are used, *forward* and *backward chaining*. Forward chaining is a top-down method which takes facts as they become available and attempts to draw conclusions (from satisfied conditions in rules) which lead to actions being executed. Backward chaining is the reverse. It is a bottom-up procedure which starts with *goals* (or actions) and queries the user about information which may satisfy the conditions contained in the rules. It is a verification process rather than an exploration process. An example of backward chaining is MYCIN [vMS81], and an example of forward chaining is Expert [WK81]. A system which uses both is Prospector [DGH79].

In this paper we present an inference engine which operates by the method of forward chaining. It is shown to be correct and optimal. In addition we indicate how universal and special purpose VLSI chips can be constructed through silicon compilation to implement rule-based expert systems.

System Overview.

Although the major thrust of this paper is the development and presentation of an optimal inference engine, we will provide a brief overview of *EXPRES*, which is a prototype expert system tool developed at the University of Kentucky. It consists of a parser and a shell (which includes the inference engine) implemented in the C programming language [Gr87]. Features of the system include a Pascal-like rule representation language, a parser/code generator, support of external libraries and subroutines, a menu driven user interface, and an executable system written in portable C code.

The parser translates an expert system written in the *EXPRES* knowledge representation language into a file that contains table definitions, condition evaluation routines, and action execution procedures. This file is then linked with the shell to produce a run-time system written entirely in C.

The knowledge representation language is a high-level block structured Pascal flavored language which allows a user to construct a knowledge base. It is based upon production rules of the form:

if *<conditions>* then *<action list>*

Conditions are expressions involving *attributes* and the logical connective *and*. Several examples are:

temperature > 110
 distance < velocity*(time1 - time2)
 cost > budget and demand = low

Attributes are of course like programming language variables and have *types* which must be *numerical* or *string*. (A string type variable can possess a value from a set of strings, for example: {true, false} or {red, yellow, green}.)

An *action list* consists of one or more of the actions described in the table below.

Action	Description
print("<string>")	Output the <i>string</i>
call <subsystem>	Transfer to <i>subsystem</i>
return	Transfer to calling (sub)system
halt	Terminate processing
<attribute> := <expr>	Assignment to <i>attribute</i>

Thus, a full example of a *rule* might resemble the following.

```

if temperature > 100 and complexion = pale then
    cost := cost + 35, print("Patient has the flu."),
    call Specialist, halt;

```

This has been a very brief introduction to the *EXPRES* system and the formulation of rule-based expert systems which can be designed using the system. There are more options which will be described elsewhere [Gr87], but this should be an adequate amount of material for the presentation of the inference engine in the next section.

The Inference Engine.

In order to execute a rule-based expert system using the method of *forward chaining* we merely need to *fire* (or execute) actions whenever they appear on the action list of a rule whose conditions are true. This involves assigning values to attributes, evaluating conditions, and checking to see if all of the conditions in a rule are satisfied. A general algorithm for this might be:

```

while values for attributes remain to be input
    read value and assign to attribute
    evaluate conditions
    fire rules whose conditions are satisfied

```

Several points about this require consideration. First, some *conflict resolution strategy* needs to be employed in order to decide which rules are fired first. Our method is to fire the rule which the system designer defined first. Also, we wish to cut down on computational time. To do this we must not do anything which does not absolutely need to be done. This means that conditions are only evaluated at the time they might change and that rules are checked (to see if all of their conditions are satisfied) only when they might be ready to be fired, not before. We shall do this as attributes are assigned values and shall only consider rules and conditions affected by the new attribute assignment.

Let us develop an inference engine for a rule-based system whose basic components are:

Attributes:	X_1, X_2, \dots, X_{n1}
Conditions:	C_1, C_2, \dots, C_{n2}
Rules:	R_1, R_2, \dots, R_{n3}
Actions:	A_1, A_2, \dots, A_{n4}

We only need to execute an action when a rule containing it is fired. We fire a rule only when all of its conditions are satisfied. To detect this we shall assign a *counter* to each rule and use it to keep track of exactly how many of the

conditions in the rule are currently satisfied. Thus, we only check to see if a rule is ready to fire when one of its conditions has become true. In turn, a condition need be evaluated only when all of its attributes have been defined and one has changed. This is kept track of with a *counter* assigned to that condition. In addition, an attribute is flagged as *defined* or *undefined*.

Going the other way, we can determine which conditions need be checked and maybe evaluated with the aid of a *condition list* assigned to each attribute. Then, the rules which need checking and possibly firing appear on a *rule list* allocated to each condition. And, each rule possesses an *action list* which enumerates the actions to be executed when the rule is fired.

For rules such as:

R₁: if $X_1 > 100$ then A₁;
 R₂: if $X_1 * 0.6 < X_2 + 50$ and $X_2 < 43$ then A₂, A₃;

we preprocess with the parser and form the conditions:

C₁: $X_1 > 100$
 C₂: $X_1 * 0.6 < X_2 + 50$
 C₃: $X_2 < 43$

Then the various lists are set up and the rules and the relationships between the attributes, conditions, rules, and actions may be presented as the graph in figure 2.

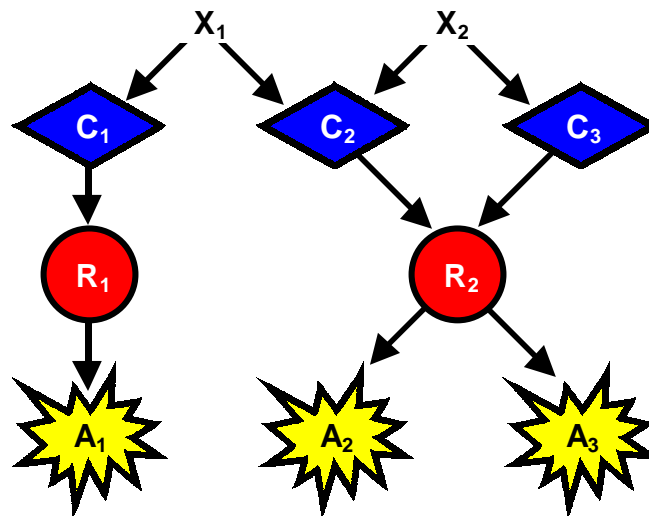


Figure 2 - Expert System Data Structure

This graph is also in some sense an illustration of the inference engine for a system containing the above two rules since the engine operates by doing a depth-first search of the graph, beginning at the attribute being changed and continuing down the graph whenever the counter assigned to a condition or a rule indicates that all of the information required is present. (Note that this is very much a finite state machine process. And as such it is related to others such as Petri nets and pattern matchers [BM77, Fo82].)

Figure 3 represents the algorithm for assigning a value to an attribute and performing all other appropriate tasks that this assignment triggers.

Assign(X_i, v)

PRECONDITION: $X_i \neq v$ and the knowledge base is correct

*POSTCONDITION: all appropriate actions have been executed
and the knowledge base has been updated correctly*

```

 $X_i = \text{value } v$ 
  for each  $C_j$  on the condition list of  $X_i$  do
    if  $X_i$  is undefined then increment the counter for  $C_j$ 
    if all attributes needed for  $C_j$  are defined then
      evaluate  $C_j$ 
      if the value of  $C_j$  changed then
        for each  $R_k$  on the rule list of  $C_j$  do
          if  $C_j$  became true then
            increment the counter for  $R_k$ 
            if all conditions in  $R_k$  are true then Fire( $R_k$ )
          otherwise decrement the counter for  $R_k$ 
  mark  $X_i$  as defined
  
```

Figure 3 - Central Inference Engine Algorithm

Correctness will be carefully examined in the next section, but several points need to be emphasized now. We note that whenever an attribute changes in value: .(l .(c a) All conditions which might change are checked, and b) All rules which might fire are examined. .)c .)l This will form the basis for the proof of correctness. We also note that nothing is examined *unless there is a need to examine it*. This is the basis for our optimality proof in the next section.

Some additional routines need to be developed to complete the algorithm. The *Fire(R_k \ /f)* procedure is very simple. It merely places all of the actions on the action list of R_k onto an *execution stack* so that they will be removed in the order that they appeared on the action list of R_k . A procedure named *ExecuteAction(stack)* removes actions from the stack and executes them until either the stack is empty or a *halt* is encountered. The main procedure calling the *Assign(X, v)* procedure is presented as figure 4.

```
Set all attributes to undefined
Set all counters to zero

while input exists and no halt is encountered do
  read(i, v)
  if  $X_i \neq v$  and v is proper type then Assign( $X_i$ , v)
  ExecuteActions(stack)
```

Figure 4 - Main Inference Engine Algorithm

An essential observation which must be made at this point is that whenever the *Assign*(X_i , v) procedure is called, its preconditions are satisfied. This means that values are assigned to attributes only when they change. This also is central to the algorithm's correctness.

Power, Correctness, and Optimality.

In this section we shall present theorems which show that the knowledge representation and inference engine of the *EXPRES* system possess great computational power and are correct and optimal. Proofs will be briefly discussed and appear in the full paper.

Let us examine computational power first. Computation can be carried out by actions which assign values to attributes. Since arithmetic is allowed in these assignments, the same kind of assignment statements performed by ordinary programming languages are possible in an expert system written in the *EXPRES* knowledge representation language. And, since subsystems can be called recursively, recursion is part of the language. Thus computer programs or Turing machines can be simulated.

Theorem 1. *Every computable function can be computed by an expert system written in the *EXPRES* knowledge representation language.*

This means that we have a *universal expert system* capability and that every expert system can be designed with this knowledge representation language. A possibly negative aspect of this computational power is that unsolvability rears its head. Expert systems can be designed which contain undetectable infinite loops - just like computer programs. Thus a designer must take great care when developing a system.

Next, correctness. Formulation of correctness is not simple due to the very properties which make the system universal. Let us proceed in small increments. Input to a system is a sequence of values to be assigned to attributes. And output consists of a sequence of actions which are executed.

Correctness demands that for *every* sequence of attribute assignments, the actions *be executed in the proper order*.

If no attribute assignments or subsystem calls are caused by the execution of actions then correctness is straightforward. The sequence of inputs causes a sequence of rules to be fired and these in turn lead naturally to the proper sequence of actions. If one of the actions is an attribute assignment then one merely inserts this assignment into the sequence of inputs (as the next input) and correctness is still well defined. (Note that this may lead to an infinite sequence of attribute assignments, but if the system was designed that way then this is correct.)

Subsystem calls are handled in the following manner. If such a call is encountered after a sequence of inputs has been processed then there is a sequence of actions which have been executed and a sequence of actions on the stack awaiting execution. The actions triggered by the subsystem are then placed between these two sequences.

The entire proof of correctness for the inference engine consists of a very detailed examination of the above intuitive discussion.

Theorem 2. *The EXPRES system inference engine is correct.*

Showing optimal execution time follows from a careful look at exactly what is checked by the inference engine. The basis of the proof is that no conditions or rules are checked unless absolutely necessary. The proof proceeds by induction on the number of attribute assignments processed. And, at the core of this induction is an advocacy argument which states that after a sequence of assignments has been made, then any evaluation of conditions or checking of conditions and rules which is omitted might cause a necessary action not to be executed.

Theorem 3. *The EXPRES system inference engine is optimal with respect to execution time.*

VLSI Implementation.

At the heart of a VLSI implementation of our inference engine is a device called a *programmable logic array* or PLA. This is a piece of array logic which is often used to construct multi-output logic functions [FM75]. Because of their regular structure, PLA's provide a rather straightforward realization for these logic functions.

Ordinarily, two planes or grids comprise a PLA. An example is pictured in figure 5.

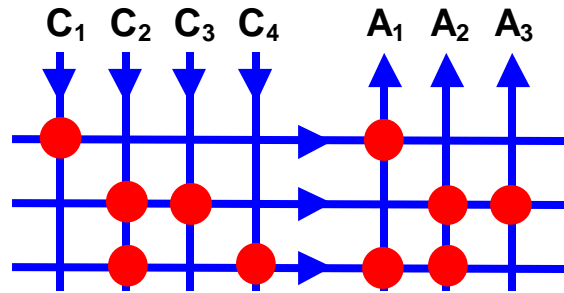


Figure 5 - Programmable Logic Array

The *AND* plane (on the left) receives inputs on its columns and produces outputs on its rows which are logical products of its inputs. The logical product output on a row consists of those input columns which contain a *personality* or connection (represented by a dot in our example) on that row. The first three rows of the above PLA's *AND* plane correspond to the products:

C_1 C_2 and C_3 C_2 and C_4

These rows or *product lines* are shared by the *OR* plane and form the inputs to that plane. The outputs of the *OR* plane occur on the columns and are sums of the products. The output at A_1 is true when:

C_1 or $(C_2$ and $C_4)$.

So, whenever there is an input signal on column C_1 there will be an output signal on column A_1 . And if there are input signals on both C_2 and C_4 then there will also be an output signal on column A_1 .

We now have a direct correspondence between PLA's and expert system rules. Consider the following relationships:

<i>AND</i> plane columns	Conditions
<i>OR</i> plane columns	Actions
Rows	Rules

Our three PLA rows now can be seen to represent the following rules.

R_1 : if C_1 then A_1 ;
 R_2 : if C_2 and C_3 then A_2, A_3 ;
 R_3 : if C_2 and C_4 then A_1, A_2 ;


```

LOAD X, 1
ADD Y, 1
LOAD Z, 2
MULT Z, 2
EQU 1, 2

```

All of the design of the ROM's and the PLA can be done by a slightly modified version of the *EXPRES* system compiler. Then they can be *mask programmed* for large numbers of chips or *field programmed* for either prototypes or for situations when small batches are desired. (Field programmable ROM's and PLA's do require extra space, so this does cut down on the size of the system which may be realized.)

On to system execution. This corresponds almost exactly to the system algorithm of figure 3. There are three major portions: condition evaluation, rule selection, and action execution. Let us examine them in order.

Condition Evaluation.

1. The user interface sends an attribute value to the I/O controller buffer.
2. This value is recorded in RAM by the *IN box* that then notifies the *condition controller* and waits until this assignment has been fully executed.
3. For every condition containing this attribute, the condition controller extracts:
 - a) attribute values (from RAM through the *OUT box*) needed to evaluate the condition and
 - b) a program from the condition ROM (R_c).
4. For each of these conditions, if all of the attributes have been defined, the condition controller selects a *processor* (P) and transmits the program and values to it.
5. After evaluation, the processor returns a Boolean value to the condition controller.
6. The condition controller conveys this value to the *condition selector* (S_c).

Rule Selection.

1. Values for the conditions are recorded by the condition selector in *flip-flops* (circles) which lead into the *AND plane* of the PLA.
2. If a rule is satisfied and actions are to be executed, the *print selector* (S_p) and the *assignment selector* (S_a), and the *system controller* (C_s) are notified by the *OR plane* of the PLA.
3. The selectors pass on appropriate information. For actions which are system calls, the *system controller* (C_s) sets flip-flops on the *AND plane* of the PLA which enable or disable the appropriate systems.

(A final note on the PLA. Between the AND and OR planes reside pairs of delay elements and AND gates on each product line. These ensure that after a rule has fired, it will not fire again until it becomes resatisfied.)

Action Execution.

1. The *print controller* (C_p) merely extracts messages from print ROM and sends them to the user interface.
2. If the assignment selector notifies the *assignment processor* (P_a) of an assignment to an attribute, the assignment processor in turn fetches:
 - a) attribute values from RAM through the *OUT box*
 - b) and a program from *assignment ROM* (R_a)
 and evaluates the assignment statement. Then it sends the new value to the I/O controller buffer.
3. When all actions have been executed, the *IN box* is requested to continue processing.

The processors, selectors, and controllers all can be designed with the aid of current CAD tools [Ga88, Mu82]. Much of the design of this system appears elsewhere [IL89] and is of a distinct electrical engineering flavor. So ends this brief overview of the design and operation of the *EXPRESS* system as VLSI circuitry.

Conclusion.

A correct and optimal inference engine for rule based expert systems was developed and presented. Its correctness is due to the following facts.

- a) *The engine is invoked when attributes change.*
- b) *Conditions are checked when their attributes take values.*
- c) *Rules are examined whenever their conditions change.*
- d) *Actions are executed when their rules are fired.*

Optimality of execution time is primarily due to the fact that nothing is done which does not need doing. In particular:

- a) *Nothing happens until attributes change value.*
- b) *Conditions are evaluated only if their attributes are defined.*
- c) *Rules are examined only when all their conditions are true.*

Another reason for the efficiency of the inference engine is that it is a finite state machine that operates on a graph rather than a tree. As it traverses the graph, it only explores paths when it is absolutely necessary to do so. The full

system also owes much of its speed to the fact that it is totally implemented in C code.

It was found that production rules for expert systems correspond naturally to VLSI array logic. This logic is very easy to design and fabricate and this leads to the possibility of straightforward automatic silicon compilation for rule-based expert systems. Thus universal or special purpose chips can be inexpensively designed and used to construct dedicated computing devices using expert system methodology.

References

- BW77** Bobrow, D. G. and Winograd, T., "An Overview of KRL a Knowledge Representation Language," *Cognitive Science* (1977).
- BM77** Boyer, R. S. and Moore, J. S. "A Fast String Searching Algorithm," *Communications of the ACM*, 20 (1977).
- DBS77** Davis, R., Buchanan, B., and Shortliffe, E. H. "Production Rules as a Representation for a Knowledge-Based Consultation Program," *Artificial Intelligence* .8 (1977).
- DGH79** Duda, R. O., Gaschnig, J. G., and Hart, P. E., "Model Design in the Prospector Consultant System for Mineral Exploration," in *Expert Systems in the Microelectronic Age*, ed. D. Michie, Edinburgh University Press, Edinburgh, 1979.
- Fo82** Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* 19 (1982).
- FM75** Fleisher, H. and Maissel, L. I., "An Introduction to Array Logic ," *IBM J. Res. and Dev.* (1975).
- Ga88** Gajski, D. D. ed, *Silicon Compilation* Addison-Wesley, Reading, Massachusetts, 1977.
- Gr87** Griffin, N. "A Fast Architecture for Rule-Based Systems," *MS Thesis, University of Kentucky*, 1987. Also in *Proc. 1988 Southeastern Regional ACM Conf.*
- IL89** Igarashi, Y. and Lewis, F. D. ."Expert Systems in Silicon," in preparation.
- Mu82** Muroga, S., *VLSI System Design*, John Wiley & Sons, New York, 1977.
- Qu68** Quillian, M. R., "Semantic Memory," in *Semantic Information Processing*, ed. M. Minsky, MIT Press, Cambridge, Mass., 1968.
- vMS81** van Melle, W., Shortliffe, E. H., and Buchanan, B. G., "EMYCIN: A Domain-Independent System that Aids in Constructing Knowledge-Based Consultation Programs," *Machine Intelligence* 3 (1981).
- WK81** Weiss, S. M. and Kulikowski, C. A., "EXPERT Consultation Systems: The Expert and Casnet Projects," *Machine Intelligence* 3 (1981).
- WiT75** Winograd, T., "Frame Representation and the Declarative/Procedural Controversy," in *Representation and Understanding: Studies in Cognitive Science*, ed. A. Collins, Academic Press, New York, 1975.