CHAPTER 3. AN EMBEDDED TEMPORAL EXPERT SYSTEM

3.1 Objectives

This chapter describes the development of an expert system suitable for real-time process control. (Laffey et al. 1988) and (Coyle 1990) have identified several objectives for such a system:

1. High speed.
2. The use of compiled, rather than interpreted, code.
3. Continuous operation without system-introduced delays (such as garbage collection).
4. A predictable response time, or a guaranteed maximum response time.
5. Focused attention -- the ability to select one of several working contexts.
6. Integration with "conventional" (procedural) software.
7. Direct interface to sensors and actuators.
8. Use of embedded hardware, such as single-chip microcontrollers with limited resources.
9. Handling of asynchronous events (interrupts).
10. Time scheduling of events.
11. Expiration or invalidation of input data.
12. Expiration or invalidation of knowledge.
13. Maintenance of a data history.
14. Ability to reason about temporal relationships.

In addition, the trend towards the use of networks of microcontrollers suggests these additional goals for the embedded expert system:

15. Ability to share data (input data or deduced facts) with other processors.
16. Ability to share knowledge (rules) with other processors.

No expert system to date has achieved all of these goals.[1] The expert system created during this research achieves all but goals 4, 5, and 12; and the potential exists to accomplish these as well.[2]

3.2 Efficient Inference Engines

A "production system" is an expert system whose knowledge base is represented in the form of IF-THEN rules (Townsend and Feucht 1986). For instance, an expert system to classify animals may have as one of its rules

```
IF (ANIMAL HAS-HAIR)
   OR (ANIMAL GIVES-MILK)
THEN (ANIMAL IS-MAMMAL)
```

The "IF" clause (*predicate*), contains the *antecedents* of the rule. The "THEN" clause (*conclusion*) contains the *consequents* of the rule, i.e., the new facts that are known if the rule is satisfied. The expert system matches known facts to rules until some ultimate *goal* is reached (in this example, identification of the animal.)

Traditionally the matching of facts to rules has been viewed as a pattern matching problem. In LISP, the patterns may be represented as lists of symbols; in other implementations, text strings have been used. Exhaustively searching the knowledge base for a given pattern is a slow process, and is the principal factor limiting the performance of expert systems. Two different techniques have been used to accelerate this.

Much research has been devoted to accelerating the pattern-match. For example, (Sedgewick 1983) describes how pattern matching can be performed by a state machine; (Siddall 1990) suggests organizing rules as a network. The prevailing algorithm, Rete, constructs a network from the knowledge base which maps new facts directly to the affected rules (Forgy 1982).

The second technique is to represent facts not as symbolic patterns, but as Boolean variables, as

1. Although the recent "Super" expert system (Morizet-Mahoudeaux 1996) comes close.

2. Predictable response time might be achieved by continuous performance monitoring. Guaranteed response time (using progressive reasoning) and multiple working contexts have been demonstrated in Forth by (Broeders et. al. 1989). Forth's "vocabulary" mechanism is another possible approach to context control. The technique described here for data expiration can also be applied to knowledge (rules); but this capability was not required by the accelerator control system.

in (Matheus 1986). For example, the previous rule could be written in C language:

```
int mammal() {
    return( hashair() | givesmilk() )
}
```

and in the postfix language Forth[3]:

```
: MAMMAL   HASHAIR GIVESMILK OR ;
```

This converts the evaluation of predicates from a pattern matching problem, to the evaluation of a logical expression. Since Boolean logic is a fundamental operation on conventional CPUs[4], logical expressions can be quickly evaluated.

In this research a third and different approach is developed, offering the advantages of both of these techniques, and more. Predicates are represented as logical expressions, and a graph is constructed to efficiently test the rules. What is novel is that the edges of the graph (paths through the network) are represented as subroutine calls, and this graph is traversed by the instruction-execution mechanism of the CPU[5]. Thus the rule network is converted to a directly-executable computer program, and the CPU to an optimal matching engine.

3.3 Backward Chaining

Given a set of inputs, the matching engine (or "inference engine") traverses the rule network and attempts to reach one or more conclusions. A "backward chaining" inference engine takes each conclusion in turn, and works backwards through the network to discover the required input conditions. If all of the required inputs are true, the conclusion is true. In this mode, each rule is a subroutine that returns a truth value by evaluating a logical expression.

---

3. Other than the niceties of syntax -- such as using : and ; to enclose a function declaration in Forth -- the principal difference in these two program fragments is that the OR operator is *infix* (between the operands) in C, and *postfix* (after the operands) in Forth.

4. Processors do exist which are designed for the LISP language; these are significantly faster at pattern matching.

5. A similar technique has been described for combinator graph reduction (Koopman 1990), but this is its first application to expert systems.

Code Generation

The postfix Forth syntax has been chosen for its ease of compilation (Kogge 1982). For example, the previous Forth fragment can be translated directly to the machine code sequence:

```
CALL HASHAIR
CALL GIVESMILK
CALL OR
RET
```

This code can be compressed 33 to 50 percent[6] through the use of threaded code (Bell 1973; Dewar 1975), which dispenses with the CALL instructions, and compiles only the addresses of the subroutines, thus:[7]

```
CALL INTERPRETER
DW HASHAIR
DW GIVESMILK
DW OR
DW EXIT
```

Since the interpreter need only execute a list of addresses, its speed penalty is tolerable (typically on the order of 100%, i.e., half the speed of the equivalent machine code). This penalty is more than offset by the advantage of portable code representation.[8] A further refinement is to use "tokenized" references to object-oriented "methods."

Applying a conventional language translator to the logical expressions of the rules will yield a subroutine network which implements a depth-first, backward-chaining search of the knowledge base. Calling any rule will cause its subordinate rules to be called, recursively, until supporting facts are found.

Inputs and Outputs

Inputs are simply subroutines which return a truth value from the external world (by reading a sensor or asking an operator, for example). These subroutines may be used freely as terms in the logical expressions of rules. The subroutine of a rule may also perform some output action, such as driving an actuator, depending upon the outcome of its logical expression.

6. 50% on the 68HC16 processor; 33% on the 8086 processor.

7. To be specific, this example is *direct-threaded* code. For the differences between direct-threaded and indirect-threaded code, refer to (Bell 1973; Dewar 1975; Kogge 1982). On most CPUs, direct-threaded code is faster.

8. For example, there are no machine instructions in the address list.

Procedural Interface

Similarly, *any* procedural code may be called from within a rule. For example, a subroutine could be called which performs numerical computation, or which accesses a database, e.g. to determine a desired goal. Indeed, numeric and logical expressions may be freely mixed within a rule. Procedural subroutines may also be called as output functions.

Since each rule is itself an executable subroutine, it is possible to call individual rules from "outside" the expert system, in order to determine their current status -- even partially activating the inference engine if needed. This provides much finer control than other expert systems, which at best may allow starting the entire inference engine, returning only when a conclusion is reached and the engine halted. Selectively accessing rules and facts, while the inference engine continues to run, is invaluable for process control.

Dynamic Memory

Pattern-matching expert systems, such as those written in LISP, make heavy use of dynamically allocated "heap" memory, as patterns (lists) are constantly created and destroyed during the inferencing process. This can lead to unpredictable delays for garbage collection or defragmenting. On small systems, the inferencing process may halt due to lack of memory. In contrast, this inference engine requires only a modestly-sized subroutine stack, which is always reclaimed perfectly after a rule is evaluated. This allows the inference engine to run continuously, with no system-introduced delays, on small microcontrollers with only a few kilobytes of RAM.

Pruning

Evaluation of a rule network can be accelerated by "pruning" irrelevant branches of the logic tree. For example, it is known that the animal has hair, it must be a mammal -- there is no need to ask (or sense) whether it gives milk. When Boolean expressions are compiled, this pruning may be achieved by the use of "lazy evaluation" logic operators. These are operators which terminate evaluation of an expression as soon as a result is known. In C, the operator || is the "lazy" OR:

```
int mammal() {
    return( hashair() || givesmilk() )    /* note the change */
}
```

and in Forth, similar operators have been devised (Bartel 1990; Rodriguez 1990):

```
: MAMMAL   HASHAIR || GIVESMILK ;    ( || is an infix operator )
```

Rule Memory

The backwards traversal of the network may follow convergent paths, causing an intermediate rule to be evaluated more than once. These superfluous evaluations may be eliminated by storing the result of the logic expression the first time the subroutine is executed (the *evaluate* phase), and returning this stored result on subsequent calls (the *invoke* phase). This requires an auxiliary flag to indicate whether the rule is "known" or "unknown" (not yet evaluated).

Redefinition of Rules

Since this technique uses the language compiler to translate the knowledge base to executable code, there is no need for a separate "rule compiler." This implies that the knowledge base is fixed at compile time, a reasonable restriction for many applications. A further advantage of the Forth language is that its integral compiler allows new code to be added at run time. New rules may thus be added, or, if the subroutines are vectored through pointer variables[9], old rules may be modified.

Uncertainty

It has been assumed that each rule returns a Boolean (true or false) result. Rules may as easily return a numeric result representing a range of certainty from false to true. The logic operators AND, OR, and NOT can be redefined to support strict probability (Siddall 1990), MYCIN uncertainty (Townsend and Feucht 1986), or other algorithms.

Fuzzy Logic

A numeric result may as easily represent a degree of membership in a fuzzy set (Mercaldo 1992; Kosko 1997). Fuzzy logic may be represented with the model just described if two conditions are met.

9. Most Forth compilers support this through the defining word DEFER.

First, each membership function is made a separate rule. Thus, instead of a single rule TEMPERATURE which has fuzzy "classes" VERY-HOT, HOT, COLD, and VERY-COLD, four rules must be defined: TEMPERATURE-VERY-HOT, TEMPERATURE-HOT, etc. Second, a new logic operator "defuzzify" must be created that combines the outputs of these rules, e.g. by computing a centroid.

3.4 Forward Chaining

For process control applications a "forward chaining" inference engine is more suitable. Given a set of inputs, this engine attempts to establish the truth or falsehood of intermediate conclusions, working forward through the tree until the conclusions have all been reached.

Many forward chaining engines (Johnson and Bonissone 1983, Matheus 1986, Sanderson and Shackleford 1986) operate by repeatedly re-evaluating the entire rule base. A significant acceleration can be achieved by maintaining, for each node in the graph, a list of other nodes which are immediately dependent on its outcome (Lewis 1986). When a node changes, only its immediate dependents need be revisited. Applied recursively, this yields a "depth first" forward traversal.

This research further accelerates the matching process by representing the forward-chain links as subroutine calls. Thus the CPU becomes the inference engine, in the same manner as the backward chaining technique just described. In this mode, the function of each subroutine is to re-evaluate the logic expression of the rule, store the result, and then call each of the dependent subroutines in turn (the forward links). It is convenient to separate this action into an *evaluate* phase[10] and a *propagate* phase.

Object Orientation

If both backward and forward chaining are to be supported, each rule requires two subroutines, or a subroutine with two entry points (Rosen, 1982). Object-oriented languages facilitate this by allowing multiple "methods" for a given object. A rule can be represented as an object, for which four distinct methods are useful:

10. If enough information about the logic expression is known, it may not be necessary to recompute it -- for example, if it is known to be an AND conjunction, and an input has just gone false. However, it is simpler, usually faster, and always more general to reevaluate the entire expression.

invoke - return the current truth value of this rule.

evaluate - recompute the logic expression of this rule.

assert - store the new truth value of this rule.

propagate - visit and re-evaluate the dependents of this rule.

Ordinarily these steps will occur in this order, as shown in Figure 3-1.

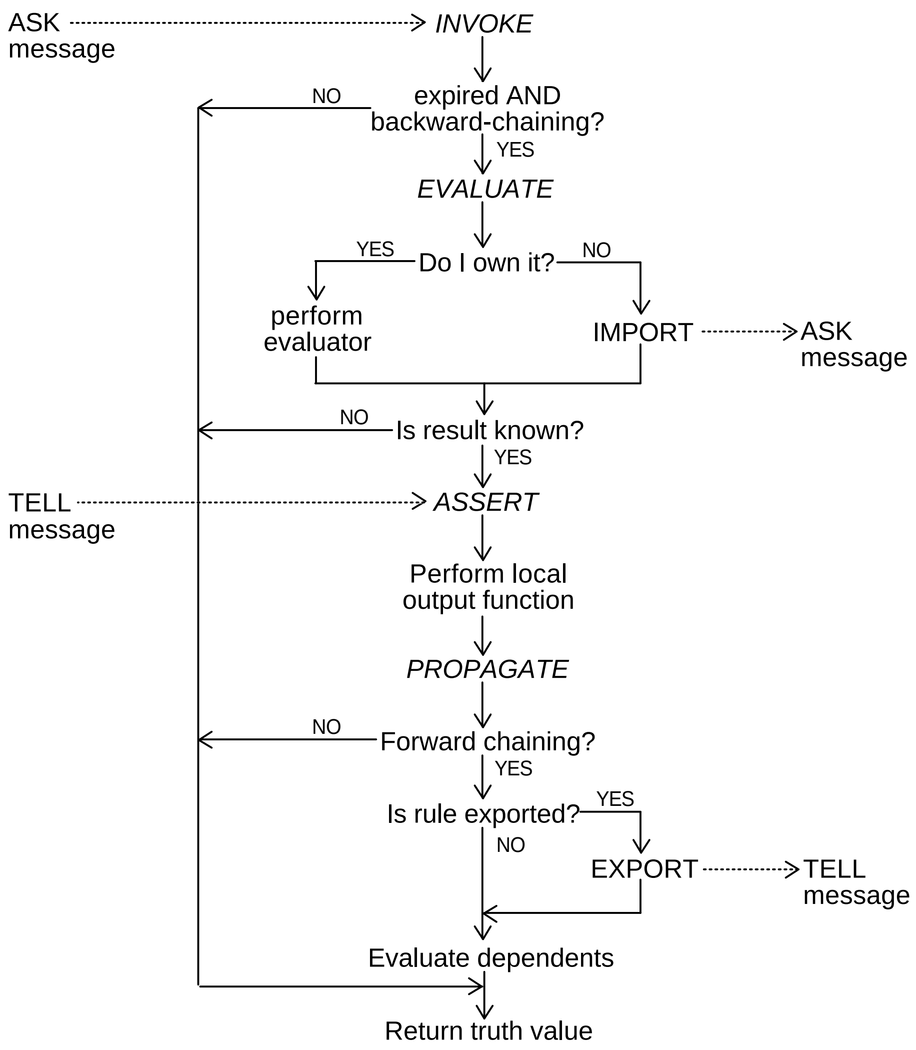

Figure 3-1. Flowchart of the Inference Engine

The object-oriented feature of "polymorphism" allows these four methods to be defined differently for different classes of rules.

Asynchronous Events

If forward-chaining is enabled, a chain of inferences is initiated whenever a new truth value is *assert*ed for a rule. This provides a means for the system to respond to asynchronous events. For example, a new data value arriving from the external world (the system being controlled) can be asserted as a new value; then all rules dependent upon that value will be re-evaluated. Since *assert* can be used as a procedural subroutine, it is even possible to initiate inferencing in response to a processor interrupt.

Time Scheduling

Most real-time operating systems have the ability to schedule a routine for execution at a given time.[11] This may be used with the *assert* operation to schedule inferencing events. Events can also be linked to a real-time clock interrupt; for example, the accelerator control system (Chapter 4) contains a fact which is asserted 18.2 times per second. Inferences which are dependent on this fact are thus regularly re-evaluated.

3.5 Time-Valued Data

An essential adjunct to forward-chaining is persistence of data (rule memory). No benefit can be expected from forward chaining if every input change requires all inputs and intermediate conclusions to be reevaluated.

For process control, a reasonable assumption is that each input is valid for some finite time. Each rule returns a 2-tuple: a truth value, and an associated "expiration time." When the current clock time exceeds the expiration time, the rule is deemed to be unknown, and will be re-evaluated.

Temporal Algebra

This expert system is unique in that expiration times are calculated automatically when the rule's Boolean expression is evaluated. This is done by "overloading" the integer arithmetic and Boolean logic

11. Either an absolute clock time, or a given number of milliseconds in the future.

operators,[12] that is, defining a new set of operators which act on time-valued data. These operators, invented during this research, comprise a new mathematics -- a "temporal Boolean algebra."

Unary operators carry the time unchanged; e.g., if X is known until time t, NOT X is also known until time t. For most binary operators, the resultant expiration time is the lesser of the two inputs; if X is known until time t1, and Y until an earlier time t2, the sum X+Y is only valid until t2. Logical operators may have a more complex behavior: if X is known to be false until t1, and Y is known until an earlier time t2, then X AND Y will be false until t1, regardless of any possible change in Y at t2. Figure 3-2 illustrates the temporal relationships for AND and OR.

AND	unknown	FALSE,t2	TRUE,t2
unknown	unknown	FALSE,t2	unknown
FALSE,t1	FALSE,t1	FALSE, max(t1,t2)	FALSE,t1
TRUE,t1	unknown	FALSE,t2	TRUE, min(t1,t2)

OR	unknown	FALSE,t2	TRUE,t2
unknown	unknown	unknown	TRUE,t2
FALSE,t1	unknown	FALSE, min(t1,t2)	TRUE,t2
TRUE,t1	TRUE,t1	TRUE,t1	TRUE, max(t1,t2)

Figure 3-2. Truth Tables for Temporal Boolean Algebra.

Pruning

When expiration time is significant, the logic tree can no longer be pruned. For example, an early term in an AND expression may be false (rendering the expression false). But a further term may be false with a greater expiration time, increasing the expiration time of the result.

Data History

It may be desirable to maintain a "history" of a rule or an input value. This may be accomplished by redefining the *assert* method to record old values when they are replaced. For example, one suitable technique records "decades" of data in progressively larger intervals, storing a year's worth of data in 81

12. Overloading refers to the programmer's ability to define multiple actions for a language operator; for example, adding new behaviors to the operator "+" to add vectors, and OR Boolean values.

samples (Garland et. al. 1989). Object-oriented programming allows different "classes" of rules to be defined, each using a different method for history storage. Methods can be added as required to these classes, to retrieve or operate on this historical data.

3.6 Temporal Logic

Temporal logic is an extension of mathematical logic which allows descriptions of temporal relationships. (Gabbay, Hodkinson, and Reynolds 1994) define eight temporal relations which may pertain to statements A and B:

GA	"A will always be true (*henceforth*)"
HA	"A was always true (*heretofore*)"
FA	"A will sometimes be true (in the *future*)"
PA	"A was sometimes true (in the *past*)"
U(A,B**)**	"B will continuously be true *until* A is true"
S(A,B**)**	"B was continuously true *since* A was true"
TA	"A will be true *tomorrow*"
YA	"A was true *yesterday*"

The last two operators imply a discrete sampling period. They might better be called "previous" and "next" to emphasize that the period need not be daily.

A process control system needs to reason about current and prior observations, thus requiring the four "past" relations. (Ostroff 1989) shows that the fundamental future operators are *tomorrow*, *henceforth*, and *until*. Similarly, all past relations may be derived from *yesterday*, *heretofore*, and *since*.[13]

Implementation

No expert system to date has implemented the operations of mathematical temporal logic. When temporal operators are provided, they are typically "ad hoc" devices to address the needs of a particular

13. Strictly speaking, *since* is not fundamental, since it may defined in terms of *heretofore*, *and*, and *not*: **S**(A,B) = **H**(¬(¬B ^ ¬**H**(¬A))). In loose English translation, the statement "B has been continuously true since A was first true" is restated "there has never been a time when B was false, but A was at some previous time true."

problem. This research has yielded a simple and efficient implementation of the relations listed above.

A simple modification to the *assert* method allows the expert system to evaluate "past time" temporal expressions. The *yesterday* (or *previous*) operator requires storing the "old" truth value of a rule, whenever it is updated. *Heretofore* requires a flag which is set if ever the rule evaluates false; if this flag is clear, the rule has always evaluted true. Similarly, a *past* flag may be set if ever the rule evaluates true.

A more general solution stores the previous value, the maximum integer value, and the minimum integer value for each rule. Boolean rules represent true and false as integers -1 and 0, respectively, so the minimum and maximum yield *past* and the inverse of *heretofore*. For integer-valued rules, the previous, maximum, and minimum values are useful in their own right, and allow computation of first difference and range.

3.7 Distributed Facts

When rules are implemented as subroutines which return truth values, they may be distributed across a network of processors through the use of Remote Procedure Call (Tanenbaum 1989). However, RPC requires blocking the caller until the reply is received, an undesirable trait in a real-time program. Also, a rule which is used frequently may cause excessive network traffic.

A better solution is to cache the truth value of each rule in each processor. The expiration time indicates when the cache is no longer valid. Only one processor, the *owner*, possesses the subroutine to evaluate any given rule. When its truth value expires in an different processor, that processor will request the new value from the owner; meanwhile it may either return "unknown," or block until the owner replies. This communication involves two network messages,

```
ASK  rule#
TELL rule#, truth value, expiration time
```

Rules have global identifiers; that is, all processors refer to the same rule by the same rule number.[14] A subset of rule numbers is reserved for "private" rules, that is, rules which are known to only one processor

14. When rules are defined as objects, the object identifier is an integer. The internal object format is common across all processors. See Appendix C for a description of the object implementation.

(and are thus never the object of an ASK or TELL message).

When chaining backward through the logic tree, an ASK message is generated during the *evaluate* phase for every unknown, "external" rule. Each ASK message will cause a TELL message in reply, requiring a total of two messages to transfer a new fact. When the TELL message is received, the *assert* method stores the fact.

Forward chaining requires the *propagate* phase to notify other processors of a new truth value, by sending a TELL message. Processors receiving the TELL message *assert* the new fact. If the network supports broadcast or multicast messages (Tanenbaum 1989), a single message suffices to update all dependent processors. Thus forward chaining may require less than half as much network traffic as backward chaining.

3.8 Distributed Rules

It is also desirable to move rules about the network. For example, a heavily loaded processor may wish to shift some of its inferencing burden to other processors.

When rules are compiled to machine code, they cannot be easily moved. The destination may be a different CPU, or may have a different load address for the routine. To solve this, the rule subroutines are compiled to a tokenized intermediate language (Apple 1979; Pelc 1992; IEEE 1994). This language, ITL (Inferencing Token Language) is a zero-operand, postfix virtual machine, and can be efficiently interpreted.[15] Tokens are represented in either 8 or 16 bits. Appendix D lists the current ITL primitives.

In addition to the previous classification of private vs. public, rules are divided into *bound* vs. *unbound*:

Private rules are known to only one CPU, and can be used only for that CPU's inferences.

Public rules are known to all CPUs; their truth values are cached in every CPU.

Bound rules must execute on a specific CPU; for example, a rule which uses a physical input.

15. Tests with the expert system have shown "token-threaded" code to be approximately half as fast as direct-threaded Forth code; however, optimization of the rule methods has regained this lost speed. See Chapter 5.

Unbound rules may execute on any CPU.

For a rule to be unbound, it must use only public facts, and it must be written entirely in ITL. Bound rules may use private facts and incorporate machine-language subroutines.

Four network messages support rule transfer across the network:

```
DEFINE-RULE rule#, ITL statement
I-OWN       rule#
WHO-OWNS    rule#
```

The owner of a rule may send it to another CPU with the DEFINE-RULE message. That CPU replies with a broadcast I-OWN message, which both acknowledges receipt and notifies other CPUs of the change of ownership. A WHO-OWNS broadcast allows any CPU to determine the current owner of a rule; the owner replies with an I-OWN message.

3.9 Implementation

The inference engine has been implemented in F-PC Forth for the IBM PC, and MPE-Forth forthe 68HC16, using the object-oriented extensions described in Appendix C. The basic rule object is defined as shown in Figure 3-3. Other rules may inherit this definition and add instance data, e.g. for temporal logic or data history.

The methods `:build`, `:dependent`, and `:evaluator` are used by the rule compiler.[16] The other methods have been previously described, except `:scrub` whose purpose is to reset the rule to its initial state (normally "unknown").

Both the evaluator and the dependency list (the backward and forward chains) are stored and executed as ITL statements. An ITL "escape" token calls machine-language or Forth subroutines. An optional "proponent" routine may be defined to perform an output action whenever the rule is asserted with a new value. The use of a proponent function or the "escape" token automatically flags the rule as

16. The rule compiler is available at all times, even while the application is running.

```
0 METHOD :build       \ initialize new rule object
1 METHOD :scrub       \ reset rule to "unknown"
2 METHOD :invoke      \ get rule's truth value
3 METHOD :assert      \ set new truth value
4 METHOD :evaluate    \ recalculate truth value
5 METHOD :propagate   \ recalculate dependent rules
6 METHOD :dependent   \ add rule#n as dependent
7 METHOD :evaluator   \ store ITL thread as evaluator
INHERIT COMMON
    CELL ITEM .Expiration   \ expiration time
    CELL ITEM .Value        \ current value
    60   ITEM .EvThread     \ ITL thread for the evaluator
    CELL ITEM .DepThread    \ pointer to dependency list
    CELL ITEM .Proponent    \ pointer to output routine
    CELL ITEM .Pending      \ flag: awaiting TELL message
    CELL ITEM .Owner        \ id of this rule's owner
    CELL ITEM .Export       \ export to this CPU (0=all)
    \ Performance monitoring data
    CELL ITEM .#Evals       \ # times Evaluated.
    CELL ITEM .#Asserts     \ # times Asserted.
    2 CELLS ITEM .EvalTime  \ accumulated evaluate time
    2 CELLS ITEM .PropTime  \ accumulated propagate time
CONSTANT FactSize
```

Figure 3-3. Definition of a Rule Object.

"bound" (owner=0). Newly asserted values may be exported to a single CPU, all CPUs (export=0), or no CPU (export=-1).

3.10 Future Work

Three desirable objectives have not yet been achieved.

Focused attention allows the knowledge base to change, depending on external conditions. This may involve storing multiple definitions for a rule, and dynamically using the definition appropriate to the current "milieu." One possible technique for dynamically changing subroutines has been described in (Broeders, Bruijn, and Verbruggen 1994).

Predictable or guaranteed response time has been addressed in many ways. The current system logs average evaluation and propagation times for each rule; this is intended to provide a rough prediction of response time. With the addition of focused attention, the "progressive reasoning" technique described by Broeders et. al. could be implemented, wherein progressively more complex rules are applied until the available time is consumed.

Expiration of knowledge would involve marking rules "invalid." This may perhaps be desirable if a rule has not been used for some time. A simple solution would employ an *invalid* ITL primitive, which would return an unknown result and generate an error alert if the expired rule was invoked. A more complex solution would reclaim the memory used by the expired rule; this would probably require expiration of its dependent rules as well.

Other promising areas for future research include:

"Just in time compilation" of ITL threads. Currently the ITL evaluator for a rule is interpreted, with a 100% speed penalty over compiled code. It would be possible, when a rule is transferred to a new CPU, to translate the ITL statement to a machine language subroutine (retaining the ITL definition so that the rule could be moved again). This would provide the speed of machine code with the portability of ITL, with a compilation cost only when a rule is moved.

Solicited rule transfer. At present only the owner of a rule can cause it to be transferred to a new CPU. Another network message, SEND-ME, could allow another processor to request ownership of a rule. This may be useful if that processor is lightly loaded, or if it uses the rule frequently.

Automatic load balancing. Rule transfer is performed manually in the existing system. A mechanism should be devised to allow the network to automatically determine the best location for each rule in the system, ideally in response to changing conditions. This is an ambitious optimization problem, and will doubtless require much more performance monitoring data for each rule and each processor.