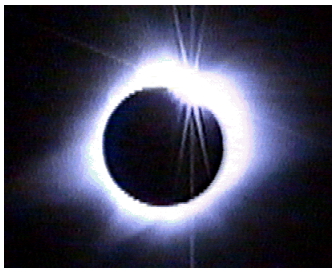

ECLIPS: An Extended CLIPS For Backward Chaining and Goal-Directed Reasoning

Peter V. Homeier and Thach C. Le

Information Technology Department
The Aerospace Corporation

Abstract. Realistic production systems require an integrated combination of forward and backward reasoning to reflect appropriately the processes of natural human expert reasoning. A control mechanism that consists solely of forward reasoning is not an effective way to promptly focus the system's attention as calculation proceeds. Very often expert system programmers will attempt to compensate for this lack by using data to enforce the desired goal-directed control structure. This approach is inherently flawed in that it is attempting to use data to fulfil the role of control. This paper will describe our implementation of backward chaining in CLIPS, and show how this has shortened and simplified various CLIPS programs. This work was done at the Aerospace Corporation, and has general applicability.

1. DESIGN CONSIDERATIONS

The Aerospace Corporation has been using expert system technology since the mid-1980s, beginning with a system to diagnose anomalies in the attitude control system of the DSCS III satellite. These experiments showed the value of expert system technology in Air Force programs, and identified key special requirements.

The Portable Inference Engine (PIE) project (Le and Homeier 1988) was intended to produce a single language and environment for Air Force expert systems to be written and run across a wide variety of hardware bases. CLIPS was identified as meeting most of these requirements.

Singular among these is real-time response (Laffey et al. 1988), which we interpret in the context of expert systems as time efficiency. The Rete net algorithm is known to possess optimal efficiency for matching many patterns to many objects (Forgy 1982). It is based on the assumption of a slowly changing state. Air Force requirements involve high rates of data to be processed in real time. Thus the state is changing rapidly, perhaps completely in a short time. This condition does not satisfy the stated assumptions of the Rete net, and thus adaptations of the algorithm are needed.

Typically, 90% of the execution of an expert system is spent in matching (Gupta 1985). Our approach to improving the speed of matching is to reduce the number of rules being considered for matching at any one time. Most real expert systems do not have a flat structure, where all rules are expected to be ready to fire at any point (Winston 1984). Rather, in many cases, there is effectively a current focus of attention, where a few rules are doing the work for the moment, and the rest of the expert system is essentially waiting around for its turn to contribute to the task. Although that waiting sounds passive, it is truly active, since the left-hand-sides of those rules are

participating in the Rete net matching process, and all facts that apply are being pushed as far as possible into the net.

These effects are exacerbated in an environment where the fact database is changing rapidly. Here the inflow of new facts create a large number of mostly unimportant rule activations and deactivations, when relevant facts are removed in favor of more recent data. Limiting the Rete net activity to those rules that are appropriate to the current focus of attention brings significant savings in avoiding unneeded matching.

Human experts often employ a combination of forward and backward reasoning (Geogeff and Bonollo 1983). A control mechanism that consists solely of forward reasoning does not effectively model this process. In response to this, in many cases expert system programmers have built a goal-directed structure into their programs by adding a clause at the beginning of each rule to select the context of the goal for which the rule applies. These clauses then match facts that the programmer asserts manually to invoke the goal. This reduces the set of applicable rules and effectively provides backward reasoning. However, this approach is clumsy, and adds extra overhead for the programmer, who has to perform housekeeping to ensure the timely removal of goal facts, and settle auxiliary issues like conflicts between two concurrent goals. Also, this practice is only a convention, not supported or checked by the expert system language.

We saw these problems arising from the attempt to use data to fulfil the role of control. What is really needed is a new control structure, that manages the goals cleanly and properly with a minimum of effort.

2. APPROACH

We propose the *module* as a collection of rules that participate together in a "focus of attention." These rules are strongly linked, in that they can be considered a small expert system dedicated to solving a single subgoal of the original total problem. There are strong arguments to be made in favor of modules from the points of view of generality, security, software engineering, and simple clarity.

Backward chaining has been generally recognized as an important inferencing capability. While a system may be constructed using only forward or backward chaining, an integration of the two provides an increase in effective computational power, allowing more natural and direct reasoning, and may reduce the length of inference chains. The module system establishes the module as the unit of control for backward chaining, while establishing forward chaining within the module (see Figure 1). The two work symbiotically; it is forward chaining that invokes new goals or returns from them, while the backward chaining controls which rules can be used for forward chaining. This synergy produces expressive and deductive power.

Modules give protection to the expert system. While a module is active, only rules within the module can fire; no rule contained within another module can fire, even if its left-hand-side is satisfied. This helps to prevent an error in expert system programs, where due to an unforeseen combination of interactions between rules during execution an unexpected rule becomes satisfied and fires, which had no relevance to the focus of attention at the time. This problem occurs while maintaining or enlarging a rule base, because of the difficulty in foreseeing all possible interactions between hundreds or thousands of rules. The module concept provides "bulkheads" to contain the flow of control within a module, similar to the independently sealable compartments aboard a submarine.

Modules also support reliability and good software engineering. It is important to construct large systems in pieces, where each piece has a distinct and well-described objective or function, and where the different pieces fit together with simple and clear interfaces. The flat structure of traditional rulebases of hundreds or thousands of independent rules, all at the same level and all interacting, is a software engineering nightmare. The concept of either exhaustively testing or actually forming a mathematical proof of correctness of such a system is clearly beyond question, due to its size and complexity. However, with a set of rules broken up into modules, conceivably each module could be verified independently, since it would contain only a handful of

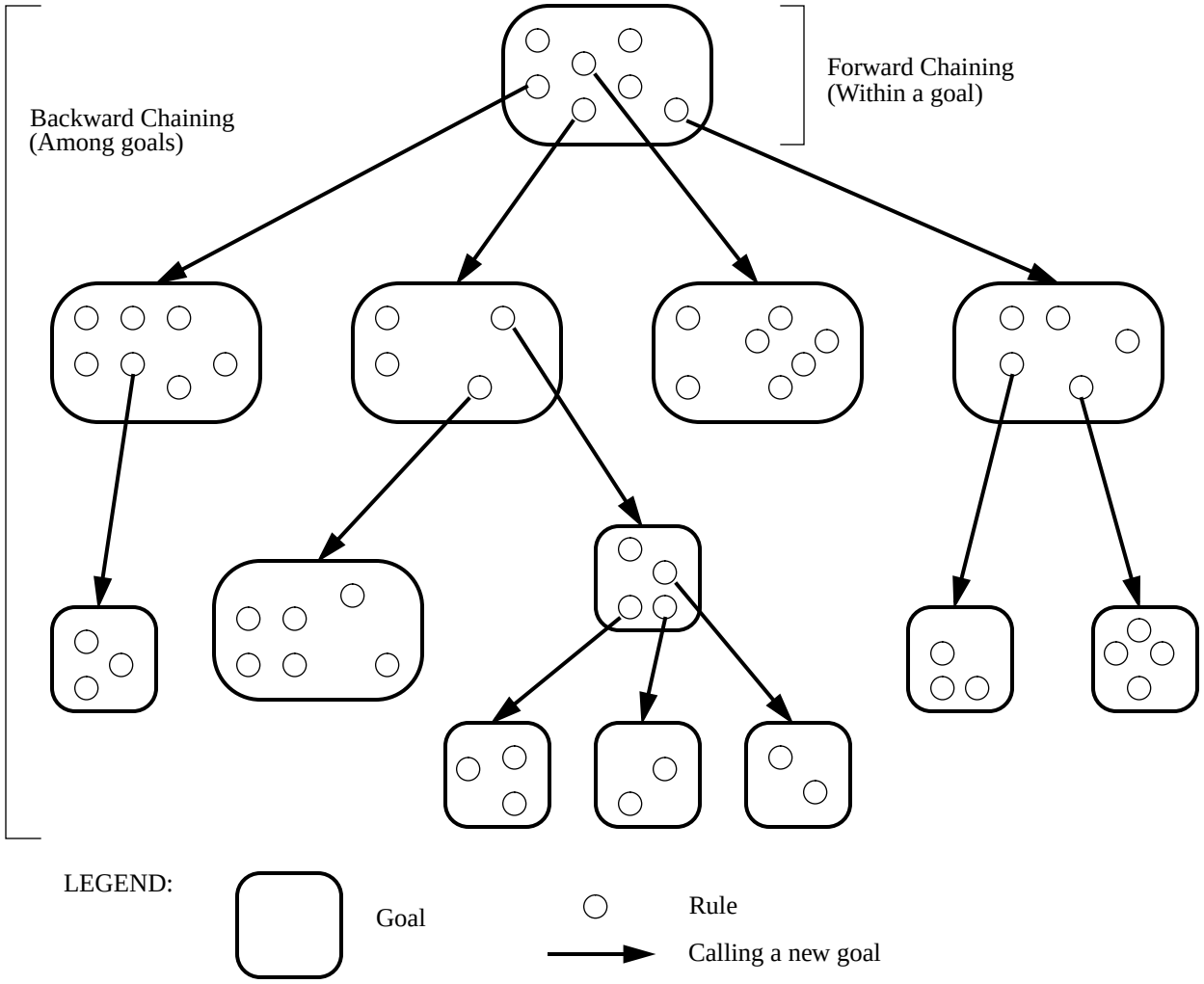

Figure 1. Forward and Backward Chaining in ECLIPS

rules by comparison, and then the results could be combined for a verification of the entire expert system.

Finally, modules clarify expert systems. Replacing the “goal” clauses in the front of rules by a single module header joining all such related rules made the rules shorter and the rule base shorter. Each rule now simply lists the conditions under which it should fire, given the context that the task at hand is the particular subgoal of this module. Data is no longer being used for control, rather, a simple control mechanism provides that function. Having a module construct to group a set of related rules that together accomplish a single purpose clarifies the whole intent of the expert system, and standardizes the meaning of “the current context.” The creation of subgoals is nested like subroutine calls; the module header clearly describes the interface for each “call.” Thinking of a module as a subtask that is accomplished out of view and then returns clarifies the thinking of the expert system writer; he can work on rules to solve one goal at a time, without being concerned with the implementation of how other subgoals are achieved.

3. INFORMAL SYNTAX & SEMANTICS

A module is a collection of rules. There is at most one module active at any one time. Only rules in the currently active module, or rules which are global to all modules, may fire. At the beginning of execution, after a reset, no module is active. A rule can activate a module by executing a "goal" statement. When a module is activated, the formerly active module is suspended until the newly activated module returns. Activations of modules are nested, similar to subroutine calls, and may be recursive. A module activation may return by executing a "return" statement. It will also return if there are no rules which may currently be fired. There is no return value; all results must be transmitted through the fact database. A module declaration may also include formal parameters. Each invocation of the module must present a corresponding list of actual parameters, which become accessible within the rules of the module via the formal parameter names. The lifetime of these bindings is the lifetime of the module activation. Activations of modules are also called "goals"; this terminology is intuitive, but introduces possible ambiguity between these activations and the goal statements that invoke them; the ambiguity is resolved by context.

ECLIPS is presently implemented using CLIPS version 4.1. The following discussion illustrates the module syntax and defines the extensions to CLIPS implemented in ECLIPS. We assume a familiarity with expert systems in general and with CLIPS in particular (CLIPS 1987).

3.1. Module Syntax

An ECLIPS program in a file may contain module definitions as well as rules and initial fact definitions. Syntax:

```
(defmodule <module name> ( <list of formal parameters> )[ "comment" ]
   <rule definition> ... )
```

Each module contains one or more rule definitions. The formal parameters are variable names, as in this example:

```
(defmodule move (?obj ?place) "Move the object ?obj to be at ?place." ... )
```

These variables are defined throughout the text of the module, and may be used on the left or right-hand-sides of rules, for example,

```
(defrule move-object-to-place ""
  (monkey ~?place ? ?obj)
  =>
  (goal walk-to ?place))
```

3.2. New Right-Hand-Side Statements

Modules are invoked as goals by using the "goal" statement on the right-hand side of a rule:

```
(goal <module name> <parameter value> ... )
```

for example,

```
(goal at ladder a5-7)
```

The actual parameter values are listed successively after the name of the module. A goal is ended and closed when a "return" statement is executed on the right-hand side of a rule:

```
(return)
```

A goal is also ended and closed when the agenda becomes empty; that is, when no rules of that activation of that module or global rules are ready to fire. When a goal is ended, the suspended goal most recently invoked is resumed.

3.3. New User Commands

The user can display the set of modules that are currently defined with the “modules” command:

```
(modules)
```

The user can display a particular module with the “ppmodule” command:

```
(ppmodule <module name>)
```

During a run, the user can display the current stack of goals, with their actual parameter values, with the “goal-stack” command:

```
(goal-stack)
```

Also during a run, the user can trace the activation and deactivation of goals with the “watch goals” command:

```
(watch goals)
```

“Watch all” now turns on “watch goals” as well as rules, facts, and activations. “Unwatch” also handles the “goals” option.

3.4. New Debugging Commands

The user can trace the development of the Rete join net with the “watch drives” command. Every node in the join net that has a binding driven into it is displayed. This adds considerably to the volume of output.

```
(watch drives)
```

“Unwatch” also handles the “drives” option. To print out the entire Rete join net at a time, the user can give the “show-jn” command:

```
(show-jn)
```

This also generates a considerable amount of output.

```
;;;*************************
;;;* chest unlocking rules *
;;;*************************

(defmodule unlock (?chest) "To unlock ?chest."

  (defrule hold-chest-to-put-on-floor ""
    (object ?chest ? light ~floor ? ?)
    (monkey ? ? ~?chest)
    =>
    (goal holds ?chest))

  (defrule put-chest-on-floor ""
    ?f1 <- (monkey ?place ?on ?chest)
    ?f2 <- (object ?chest held light held ?contains ?key)
    =>
    (printout "Monkey throws " ?chest " off " ?on " onto floor." crlf)
    (retract ?f1 ?f2)
    (assert (monkey ?place ?on blank))
    (assert (object ?chest ?place light floor ?contains ?key)))

  (defrule get-key-to-unlock ""
    (object ?chest ?place ? floor ? ?key)
    (monkey ? ? ~?key)
    =>
    (goal holds ?key))

  (defrule move-to-chest-with-key ""
    (monkey ?mplace ? ?key)
    (object ?chest ?cplace&~?mplace ? floor ? ?key)
    =>
    (goal walk-to ?cplace))

  (defrule unlock-chest-with-key ""
    ?f1 <- (object ?chest ?place ?weight ?on ?obj-in ?key)
    (monkey ?place ?on ?key)
    =>
    (printout "Monkey opens chest with " ?key " revealing " ?obj-in crlf)
    (retract ?f1)
    (assert (object ?chest ?place ?weight ?on nil ?key))
    (assert (object ?obj-in ?place light ?chest nil nil))
    (return))

)
```

Figure 2. An example module

4. EXAMPLE

The example in Figure 2, taken from the monkey-and-bananas problem, shows a module called "unlock", whose purpose is to accomplish the goal of unlocking a chest; the particular chest to unlock is indicated by the formal parameter ?chest. This module represents the rules that accomplish this subgoal of the overall goal of the monkey to eat the bananas. There are five rules in this module, three of which invoke further subgoals as part of solving this one, and two which are able to take immediate action. Only one rule has an explicit return statement. This is an example of a well-coded module, with rules which cooperate in solving a single, well-defined task.

Compare these rules with the corresponding normal CLIPS counterparts:

```
(defrule hold-chest-to-put-on-floor ""
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

versus

```
(defrule hold-chest-to-put-on-floor ""
  (goal-is-to active unlock ?chest)
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  (not (goal-is-to active holds ?chest))
  =>
  (assert (goal-is-to active holds ?chest)))
```

The ECLIPS code for the "unlock" module is shorter by 5 lines, over 12%, and the lines are shorter and less complex. In particular, note the CLIPS need for special code to prevent goal duplication. This section is typical of the entire monkey-and-bananas example.

5. IMPLEMENTATION

The fundamental idea in the implementation is inspired by the example above, but different. The example shows an attempt to implement backward chaining in normal CLIPS using data for control. In ECLIPS, every rule within a module is compiled into the Rete net with an additional clause at its front, describing the module for which this rule is active. For example, the first rule in the example above would be compiled not as

```
(defrule hold-chest-to-put-on-floor ""
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

but as

```
(defrule hold-chest-to-put-on-floor ""
  (goal unlock ?chest)
  (object ?chest ? light ~floor ? ?)
  (monkey ? ? ~?chest)
  =>
  (goal holds ?chest))
```

The new clause consists of the standard word "goal", then the name of the module, then the formal argument names. (The word "goal" is not reserved in ECLIPS, but should not be used by the ECLIPS programmer as the first word in facts, to avoid confusion with the module implementation.) This clause is prepared while parsing the module header line; it is stored in the internal CLIPS structures for a clause. Then while parsing each rule within the module, the goal clause structure is copied and prepended to the structure being prepared to represent the list of clauses on the left-hand-side of the rule. The goal clause will match a fact of the correct form, for example

```
(goal unlock red-chest)
```

The "goal" statement, which appears on the right-hand-side of a rule, has the semantics of activating a module. This is implemented by creating a "goal" fact of the form shown above and adding it to the fact database. The new goal fact is automatically driven into the Rete net, and is combined in the usual method with facts that satisfy other clauses of the rules in the module to create activations which are put on the agenda in the normal way. These "goal" facts are removed from the fact database by returning from a module, either by an explicit return statement or by having no further rules to fire.

This method of implementation means that the Rete net does most of the work of managing the applicability of rules. While applicable facts do get accumulated at the upper leaves of the Rete net, none of them can merge with superior clauses until the first clause, the "goal" clause, is matched. When that happens, then all the potential subordinate matches and activations are free to occur.

A data structure is maintained in ECLIPS, called the "goal-stack," which is a stack of the goal facts that have been introduced and not yet removed; it is thus a statement of the modules that have been entered, each with their actual parameter values. The fact on top of the goal-stack describes the currently active module.

When a new module is activated and the prior one is suspended, that suspension is not accomplished by removing the associated goal fact, as might be imagined, but rather through a modification of the conflict resolution strategy employed by CLIPS. CLIPS maintains an agenda which is a list of activations of rules that are ready to fire. The agenda is kept ordered by priority, and within each level of priority, the agenda is ordered by recency, with newest first. As new rule activations are generated, they are added to the appropriate place in this agenda, so that the desired order is preserved. Every expert system must have some policy for deciding which activation of the available set will be chosen to fire. This is called "conflict resolution." The conflict resolution policy implemented in CLIPS simply chooses the first activation in the agenda, i.e. the most recent of the highest priority. However, in ECLIPS, the conflict resolution is modified to choose the first activation in the agenda from the current module, or which is from a global rule, not contained within any module. Thus there may be more recent or higher priority activations which are passed over if they belong to a module which is not the currently active one. Activations therefore are continuing to occur for rules in suspended modules; however, none can fire until those modules become the current module.

It is important to allow these activations from suspended modules, and to leave these activations on the agenda, even though they are "inert" as long as the current module is active. Otherwise, we would lose the property of reflexivity, which assures that if a rule fires, then it will not fire again on the same data that matched its left-hand-side. Reflexivity is accomplished in the CLIPS implementation by simply putting activations onto the agenda when they are generated, and removing them when the activation is actually fired, or when any of their fact support is removed. As long as the facts do not change, the rule is only activated once, and once it is fired, it is off the agenda. This information, whether the rule has fired, would be lost if the goal fact for the rule's module were retracted and later re-asserted when the module ended its period of suspension; we would not know which rules had already fired.

Unlike the attempted CLIPS implementation of backward chaining, these "goal" facts are not visible during normal "(facts)" queries. This is accomplished by giving them negative ID numbers, similar to the "not" facts that are generated to help implement negative clauses, such as

```
(not (object ladder ?place ? ? ? ?))
```

Facts with negative ID numbers are not printed by the "(facts)" command; therefore these goal facts do not clutter up the user's view of the fact database with control-related constructs.

Modules may invoke themselves recursively, either directly or through intermediate modules; this is accomplished automatically in the implementation described above. New goal facts do not interfere with the presence of older ones, even for the same module and with the same actual parameters. The goal stack enables the recursion by keeping track of which goal is current. The Rete net keeps track of which rules are ready to fire and for which module activations. Every rule activation on the agenda keeps a list of the facts which satisfied its left-hand-side; for rules within modules, this includes the goal fact which satisfied the rule's goal clause.

When the search of the agenda cannot find an activation to fire, normal CLIPS will end; ECLIPS however will pop the goal-stack, removing the goal fact associated with the last goal, and re-search the agenda. Removing a goal fact will automatically cause all rule activations from that activation of a module to be removed from the agenda. The goal stack will continue to be popped

until either an appropriate activation is found or the goal-stack is empty, at which time ECLIPS will end.

The activation and return from goals are particularly interesting events to an ECLIPS expert system programmer, and so the "(trace goals)" utility was added to keep track of this changing context. In addition, at any point the user can give the "(goal-stack)" command to print out the entire stack of module activations, with the current module on top of the stack.

During the study of CLIPS, it was necessary to understand its data structures in detail, particularly the Rete net. Some utilities, "(show-jn)" and "(watch drives)" were built to provide visibility to the Rete net and its changes during computation, and these have been retained as generally useful learning tools for those who wish to study a real-life implementation of the Rete net algorithm.

6. COMPARISONS BETWEEN CLIPS AND ECLIPS

Here we compare CLIPS and ECLIPS in size and time. Two of the original examples distributed with CLIPS were recoded in ECLIPS. "mab" is a version of the traditional monkey-and-bananas problem, and "wine" is an expert system to choose an appropriate wine for dinner.

6.1. Size Comparisons

	CLIPS lines	ECLIPS lines	% reduction
mab	235	211	10 %
wine	419	346	17 %

Table 1. Size Comparisons

Some of the savings in the wine example were the result of eliminating 6 rules which only controlled the sequencing between phases of the wine selection process; the module structure allowed a less cumbersome coding.

6.2. Time Comparisons

Here are the results, in seconds, of 100 iterations on a SPARCstation 1, with i/o dependencies reduced by directing output to /dev/null:

	CLIPS	ECLIPS	% reduction
mab	28.1	26.8	4.6 %
wine	12.13	11.06	9.0 %

Table 2. Time Comparisons

The question arises, what is the time efficiency cost of the new features that ECLIPS provides? ECLIPS is completely backwards compatible with CLIPS, so a normal CLIPS program will run exactly the same under ECLIPS; but how much of a performance penalty will it suffer for the additional parsing, new statements and commands, goal stack maintenance, and additional agenda processing? The answer is that it is very difficult to measure any appreciable difference at all! It appears to be about 0.25%; in any case, it is well within the normal variance between runs.

7. FUTURE WORK

As mentioned, this implementation was built on CLIPS version 4.1. CLIPS has undergone many subsequent changes (version 5.0 is being released) but still does not support true backward chaining. We hope to port the ECLIPS modifications to CLIPS version 5.0, and make this capability available to interested parties.

There were several ideas presented in the prior paper (Le and Homeier 1988) that were not included in the implementation, for reasons of time. We described a capability for the user to specify the conflict resolution strategy, enabling the use of dynamic prioritization or other application-specific control strategies. These may support the creation of rule bases more directly matching the expert's control knowledge of how to apply his rules. We also intended originally to allow a *set* of modules to be activated at one time, instead of just one. This would allow, for example, the dynamic "widening" or "narrowing" of the search for the explanation of an anomaly to include more or fewer subsystems of a satellite.

We also considered having the activation of a module immediately cause a suspension of the execution of the RHS of the rule being fired, and then after the module activation returned, the RHS would be resumed at the next statement. Instead, the current implementation merely changes the currently active module, which has its effect upon the next selection of a rule to fire. In addition, we considered extensions where modules returned either a flag indicating success or failure, or an arbitrary value computed as the result of the subgoal.

8. SUMMARY AND CONCLUSION

ECLIPS provides backward chaining in the CLIPS environment in a clean and simple manner, yet it shortens program code, generalizes the inferencing, increases clarity, provides security, supports good software engineering, and runs faster. There is no significant penalty for using ECLIPS in place of CLIPS.

We believe that the module concept is applicable to the majority of all domains, whenever the size of the rule base grows beyond a certain size. We hope that the ideas in ECLIPS contribute to the practical work of the CLIPS community and beyond, enabling the creation of more effective expert systems in all domains.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Russ Abbott, Dr. Jim Hamilton, Dr. Stephen Hsieh, and Charles Simmons who made many valuable comments on this paper. The authors would also like to express their appreciation to Rick Cowan, who provided the initial vision and impetus to the PIE project.

BIBLIOGRAPHY

Brownston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5, Addison Wesley Publishing Company, 1985.

CLIPS Reference Manual, Version 4.0, March 1987, Mission Support Directorate, Mission Planning and Analysis Division, NASA, 87-FM-9, JSC-22552.

Forgy, C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, vol. 19, 1982, p. 17-37.

Forgy, C. and Shepard, S., "Rete: a Fast Match Algorithm," AI Expert, Jan. 1987, p. 34-40.

Geogeff, M. and Bonollo, U., "Procedural Production Systems," Proceeding of the Eighth International Joint Conference on Artificial Intelligence, vol. 1, 8-12 Aug. 1983, Karlsruhe, West Germany, p. 151-157.

Gupta, A., “Parallelism in Production Systems: The Sources and the Expected Speed Up in Expert Systems and Their Applications,” Fifth International Workshop Agence de l’Informatique, Avignon, France, 1985, p. 26-57.
Laffey, T., Cox, P., Schmidt, J., Kao, S., and Read, J., “Real-Time Knowledge-Based Systems,” AI Magazine, Spring 1988, p. 27-45.
Le, Thach, and Homeier, Peter, “PORTABLE INFERENCE ENGINE: An Extended CLIPS for Real-Time Production Systems”, Proceedings of the Second Annual Workshop on Space Operations Automation and Robotics (SOAR ‘88), July 20-23, 1988, NASA Conference Publication 3019, p.187-192.
Mettry, W., “An Assessment of Tools to Build Large Knowledge-Based Systems,” AI Magazine, Winter 1987, p. 81-89.
Michie, D., “Expert Systems,” The Computer Journal, vol. 23, 1980, p. 369-376.
Nilsson, N., Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California, 1980
Winston, P., Artificial Intelligence, Second Edition, Addison Wesley Publishing Company, July 1984.