N89-19843

PORTABLE INFERENCE ENGINE:
An Extended CLIPS for Real-Time Production Systems

Thach Le & Peter Homeier
The Aerospace Corporation

Abstract

The present CLIPS (C-Language Integrated Production System) architecture has not been optimized to deal with the constraints of real-time production systems. Matching in CLIPS is based on the Rete Net algorithm, whose assumption of working memory stability might fail to be satisfied in a system subject to real-time dataflow. Further, the CLIPS forward-chaining control mechanism with a predefined conflict resolution strategy may not effectively focus the system's attention on situation-dependent current priorities, or appropriately address different kinds of knowledge which might appear in a given application. Portable Inference Engine (PIE) is a production system architecture based on CLIPS which attempts to create a more general tool while addressing the problems of real-time expert systems. Features of the PIE design include a modular knowledge base, a modified Rete Net algorithm, a bi-directional control strategy, and multiple user-defined conflict resolution strategies. This paper will analyze the problems associated with real-time applications, and explain how the PIE architecture addresses these problems.

1. Introduction

Expert system technology has been successfully applied to a number of practical applications [1]. As this technology becomes more widely used and starts to address more complicated real world problems, issues of portability and accommodation of real-time data will become more significant.

The ability to handle real-time data will automate processes and remove the human dependency for some cognitive preprocessing of inputs. For example, a real-time satellite diagnosis expert system, instead of asking the operator to analyze satellite telemetry data, will take in the raw telemetry data directly and extract the needed information. This capability is important, especially in a situation where the amount of real-time data precludes exhaustive human preprocessing. This situation exists in many aerospace applications.

CLIPS, an OPS5-like (Official Production System) production system written in the C language by NASA, is an excellent delivery vehicle because of its portability. However, it was not designed for real-time systems. PIE (Portable Inference Engine) is a production system based on CLIPS which attempts to avoid degradation in performance when the system is subject to real-time dataflow and also supports a more explicit control mechanism.

In part two and three of this paper, we will briefly review a typical production architecture as exemplified by CLIPS and discuss the problems which arise in real-time systems when employing this type of architecture. In part four and five, we describe the PIE architecture, design issues, and how this architecture accommodates these problems. PIE is an on-going project, and its architecture is currently being implemented for embedded applications.

2. CLIPS & Production Systems Architecture

A production system is a typical rule-based system. It consists of a knowledge base and an inference engine. The problem state, contained in a data structure called working memory, is represented by facts. These facts may be created during the problem-solving process, either through rule execution or via the external environment. The knowledge base, which resides in a data structure called production memory, contains rules of the form "IF antecedent THEN consequent". Usually the antecedent is a set of patterns representing the rule's conditions, and the consequent represents conclusions or actions.

Production systems may be data-driven, goal-driven, or some combination of the two. CLIPS' behavior rests on the frequent re-evaluation of the problem state (represented by the current memory elements in the working memory), rather than on any static control structure of the program. Therefore, CLIPS falls into the class of data-driven production systems.

A production system is usually described in terms of recognize/act cycles, which may be divided into three separate processes:

1. Matching: Match a set of existing facts in working memory, which represents the current problem state, against all available rules. Rules whose conditions are satisfied are called instantiated. The set of instantiated rules at any cycle is called the conflict set. The matching process updates the conflict set at each cycle.

2. Conflict resolution: Select a single rule from the conflict set based on some criteria, which could be (as in the CLIPS case) user-predefined priority of rules, the recency of working memory elements, the number and complexity in rules' patterns.

3. Execution: Carry out the actions specified on the Right Hand Side of the selected rule. This could affect the content of the working memory (change the problem state).

The production system performs this recognize/act cycle

repeatedly until encountering an empty conflict set or a halt action. The production system as a model of computation provides a powerful context within which large, ill-structured problems may be described [2].

3. Real-time Issues

The term "real-time" is not easily defined. It is usually associated with fast response. A more precise definition is a system which has guaranteed response time for a defined class of events. In general, the design of a real-time system involves an integrated hardware/software approach, with careful prioritization of competing service requests. A more limited definition for real-time is adopted here. For production systems, we define a real-time system as one where efficiency is a primary design concern, which allows the generation of working memory elements from on-line inputs, and which has the capability for conflict resolution based on dynamic prioritization.

In the following sections, the real-time issues that are related to the performance of CLIPS are discussed. When the system is subject to real-time dataflow, two main sources of deficiency are identified: the control mechanism and the matching process.

3.1 Control

CLIPS provides an implicit control mechanism, built-in to the production system to govern the direction of the inference engine. This control algorithm is specified in the conflict resolution strategy. The CLIPS conflict resolution approach is to select a rule from the conflict set according to its priority, the recency of the working memory elements that match the rule conditions, and the specificity of the conditions (measured by the number of tests performed). In addition, rules that have previously fired will not be fired again on the same facts or working memory elements.

To exploit the full power of the CLIPS language, the application programs should be data-driven. The course of execution, or the sequence of rule firings, should be sensitive to the characteristics of the data. Such systems, where the direction of problem-solving is from facts toward goals, are characterized as having a forward-chaining control strategy. Other systems might use a backward-chaining, or goal-driven, control strategy, where the direction is from goals toward facts.

For a given search space, the best direction of reasoning is to move in the direction of less alternatives to minimize backtracking, a process of returning to the parent search node to explore other alternatives if the current search node is not satisfied [4]. Hence, the decision to use either forward-chaining or backward-chaining is dependent on the structure of the search space [5]. However, many realistic problems do not have a simply structured search space. For such problems, an efficient reasoning strategy will be bi-directional (i.e., combining both backward- and forward-chaining).

A bi-directional control strategy, beside enhancing the searching process performance, is also useful in expressing the way of human experts do problem solving [3]. He or she often alters the line of reasoning and sets up different hypotheses, or goals, if a particular fact is observed.

A problem with a predefined conflict resolution strategy is that its algorithm is not appropriate for all applications. Different applications or different parts of the same application might require different control knowledge specific to their domains. For example, many applications select rules based on confidence factors, or perhaps some rule-of-thumb provided by an expert. Such knowledge has to be embedded within rules, whereas the appropriate place is the conflict resolution. The code as a result will be harder to understand. There are also performance penalties. The system, instead of taking one recognize/act cycle to select the right rule, has to perform a few recognize/act cycles to come up with the same result. In time-critical situations, explicit control may be needed to quickly resolve conflict.

In building a large expert system, one may encounter many problems that require various control techniques to keep the system's performance efficient. As a real-time production system language, PIE must fulfil this requirement.

3.2 Matching

Matching in production systems is the process of collecting a set of rules, the so called conflict set, that have their conditions satisfied by the current problem state. This process takes about 90% of a recognize/act cycle [6]; therefore, the performance of the matching algorithm is crucial to the overall performance of expert systems.

CLIPS uses the Rete algorithm for matching. Rete is known to be the most efficient matching algorithm for many patterns to many objects [7]. It is implemented in several popular expert system shells [8]. One major assumption that contributes to Rete's efficiency is the assumption that the problem state changes slowly. This assumption is valid for many systems where the problem state is changed only by the execution of rules. Since each rule is usually small, its effects on the problem state should also be small. The Rete match algorithm capitalizes on this observation by saving the previous cycle's matching information, and only updating the matches that have changed. However, in a real-time situation, the problem state could be changed, possibly massively, by the external environment. This violates the Rete Net's assumption, and Rete should not be used in such cases [9].

As mentioned before, some degree of backward chaining is often incorporated into a forward-chaining system. In this case, the resulting system behaves as if it only focuses its attention on the most recent goal; but in reality, at the underlying level, the system still spends its computational resources matching all rules, which includes many that are not relevant to the current goal. For example, if the current task of an automobile diagnosis system is to check out a problem with the battery, it should not devote much of its computing resources to match rules related to incoming data from the engine.

4. PIE

PIE is a production system based on CLIPS with architectural modifications to increase applicability to real-time systems and to provide enhanced production system capabilities.

4.1 Design Issues

As seen above, the potential unsuitability of CLIPS as a language for building real-time expert systems results from the built-in control mechanism, and from assumptions of the Rete matching algorithm. In order to overcome these

shortcomings, PIE has two requirements. First, the matching algorithm should be sensitive to the changed data that are relevant to the current task being solved, but not to the total amount of changed data. Second, the control mechanism has to be more explicit and flexible enough to accommodate different kinds of knowledge which might appear in a single application.

The primary architectural features of PIE which differ from CLIPS are a modular knowledge base, a modified Rete match algorithm, a bi-directional control strategy, and multiple user-defined conflict resolution strategies. The following section will detail these features.

4.2 Architecture

Beside rules, PIE has two other data structures, called modules and goals. A module is a set of rules that are grouped together based on their functionality or any other convenient criteria that the programmer defines. A goal is a set of modules that defines a particular task to be achieved. A module could belong to more than one goal; in this way, multiple definition of a set of rules that belong to more than one goal can be avoided.

At any time, there will only be one active goal. Only the active goal's rules are to be considered in the current recognize/act cycle. Corresponding to this set of active rules is a set of active working memory elements which are defined in the active rules' conditions. Matching has to be done between the active rules and the active working memory elements only; other rules and working memory elements will be ignored as long as they remain passive.

A rule, once selected to fire, could activate a new goal or deactivate the current goal. Initially, a top-level goal is activated. If a new goal X is activated by a rule in the top-level, then X is said to be a child of the top-level goal, and the top-level goal is said to be a parent of X. When activated, a goal's rules become active and its parent's rules become passive. When deactivated, a goal's parent rules will become active.

Associated with each module is a conflict resolution strategy that can be defined either by the programmer or by a default strategy. Modules included in the same goal must share the same conflict resolution strategy. The conflict resolution is defined in a procedural language (C), and can access system information such as number of instantiations, rule priority, recency factor, number of tests, condition patterns of rules, etc.

The system at the highest level is a goal tree that behaves like a goal-driven/backward-chaining system. At the goal level, it behaves like a purely data-driven/forward-chaining system with a user-defined conflict resolution strategy. With this architecture, it is possible to build any level of integration of forward-chaining and backward-chaining (See figure 1). For example, a purely forward-chaining system, such as CLIPS, is a PIE with a single goal. A purely backward-chaining system is a goal tree in which each goal contains rules to invoke subgoals, except for the leaf goals of the goal tree which contains rules that match to facts.

The Rete Match algorithm compiles all rules in production memory into a dataflow graph called the Rete Net [7]. The matching information is saved in each node of the graph. For

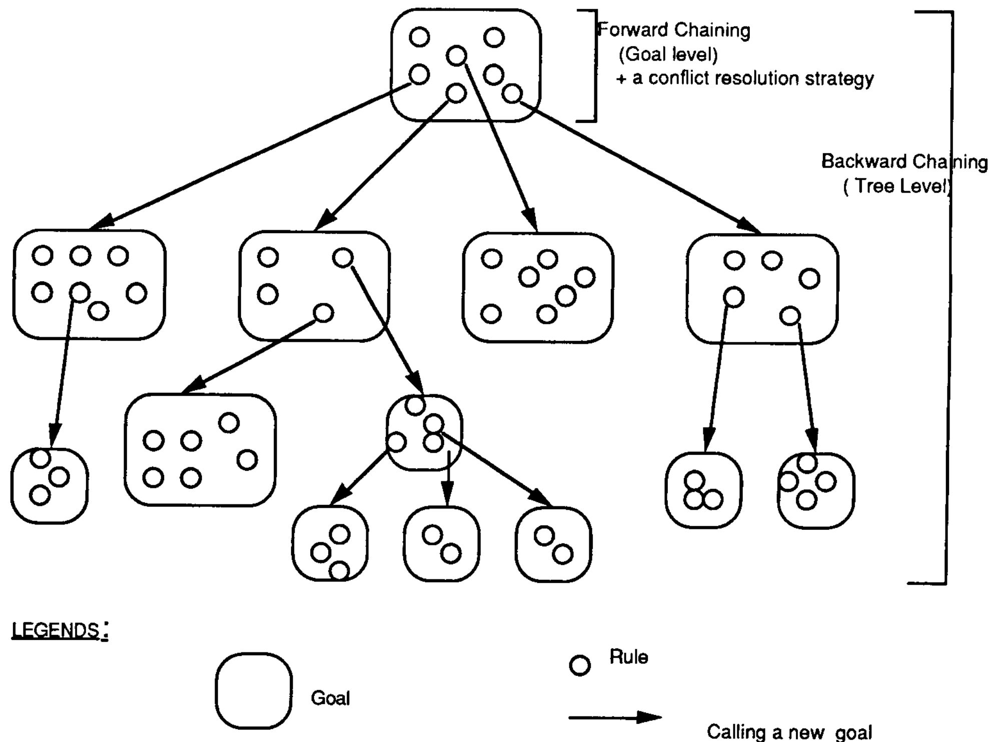

Figure 1: Forward Chaining and Backward Chaining in PIE

ORIGINAL PAGE IS
OF POOR QUALITY

PIE, Rete is modified so that at each recognize/act cycle, it only updates a partial net related to rules belonging to the current active goal, instead of updating the whole net. A controller is added to the Rete algorithm to keep track of the relationship among rules, modules, and goals. During the course of execution there usually are many goals to be activated and deactivated. The controller's function is to turn on the appropriate nodes to be involved in the matching for the current active goal and turn off those of the deactivated goal (see figure 2).

The incoming data of an inactive goal will overwrite the older data but will not participate in the matching process until the goal is activated. If older data are to be saved in time order for some later use, the corresponding patterns created have to include the time index defined by programmers.

PIE is an on-going project. The major effort has been to understand the implementation of Rete algorithm in CLIPS and to build a controller for Rete. The user-defined conflict resolution strategies written in the C language will be compiled and called appropriately with the activation of corresponding goals. Other work will be to enhance the parser to recognize modules, goals, activation, deactivation, etc.

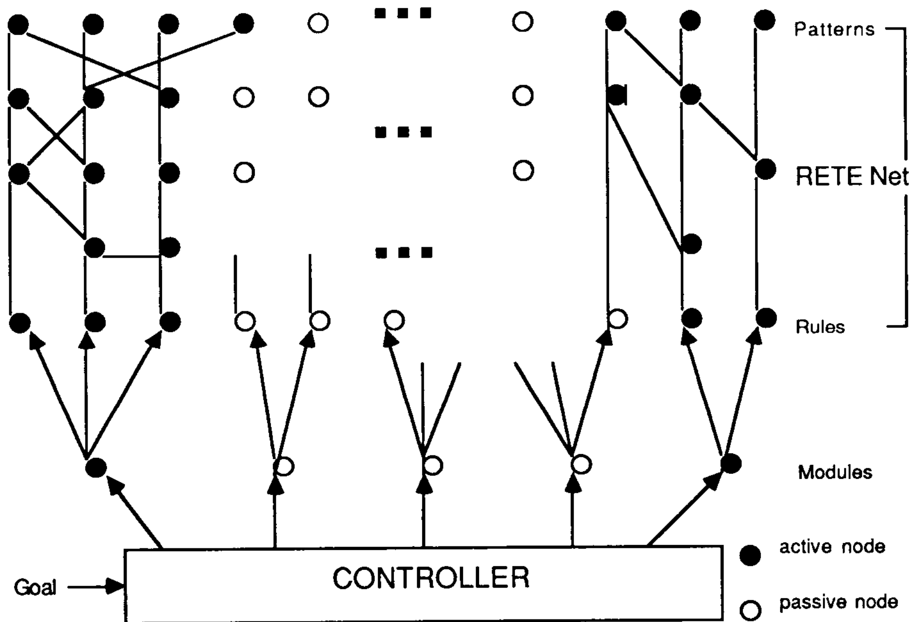

Figure 2: Controller For Rete Match Algorithm

5. Advantages of the PIE Architecture

As stated previously, modularity and the concept of an active rule set decreases the magnitude of the matching problem, permitting use of the Rete match algorithm in a rapidly changing problem state. Each goal of a goal tree might associate with a number of on-line inputs, so the cost of matching for each goal over a recognize/act cycle is proportional to this number of inputs. During the course of execution, the system might explore only part of the goal tree before the solution is found; by focusing the matching process on the currently active goal, the system computing resources are not spent on rule matching for goals never explored.

In addition, supporting modularity at the language level strips away much of the bookkeeping usually needed at the programming level to achieve the same purpose. The result is clearer, easier to understand programs, assisting debugging and maintenance.

Large expert systems also benefit from modularity. Because of the recursive nature of PIE, where each goal can be thought as a PIE system by itself, a problem can be broken down into subproblems or goals, then can be tackled independently. On the other hand, one PIE system can be integrated into another PIE system as a new goal at any appropriate level of the goal tree.

The integrated forward- and backward-chaining control strategy of PIE optimizes the search process of many realistic problems whose search space does not directly support either a simple forward-chaining or a simple backward-chaining strategy. Simulated backward-chaining is no longer required. A programmer can look at the search space structure of his or her problem and decide on the appropriate strategies for different portions of the search space.

The computational cost of a production system is due to two elements: the rule application cost and the control costs. In figure 3, the cost of rule application is high if the level of "informedness" (i.e., encoded knowledge) of the control strategy is low, and vice versa [4]. With the availability of user-defined conflict resolution strategies, programmers determine the right level of informedness for the control strategy. This will optimize the overall computational cost of

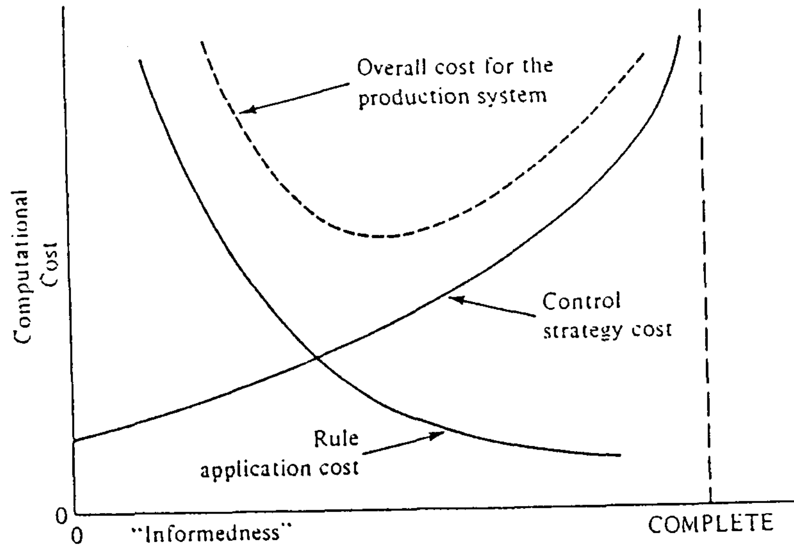

Figure 3: Computational Cost of AI Production Systems [4].

the production system while reducing the obscurity of control knowledge embedded in the rule set. User-defined conflict resolution also helps to preserve the data-driven or the goal-driven nature of rules.

From the design of PIE architecture, the predicted performance of PIE and of CLIPS as functions of the amount of incoming data is shown in the figure 4. If there is no incoming external data, the performance of PIE is likely to be lower than CLIPS due to additional overhead. As the amount of incoming data increases, CLIPS performance will degrade because of the increasingly invalid assumption used in the Rete matching algorithm. PIE performance is expected to degrade more slowly because of its insensitivity to the total amount of incoming data. The degradation of PIE performance depends mainly on the amount of incoming data associated with activated goals. The programmers have the flexibility to improve the performance by designing the system so that this number can be reduced to as much as possible.

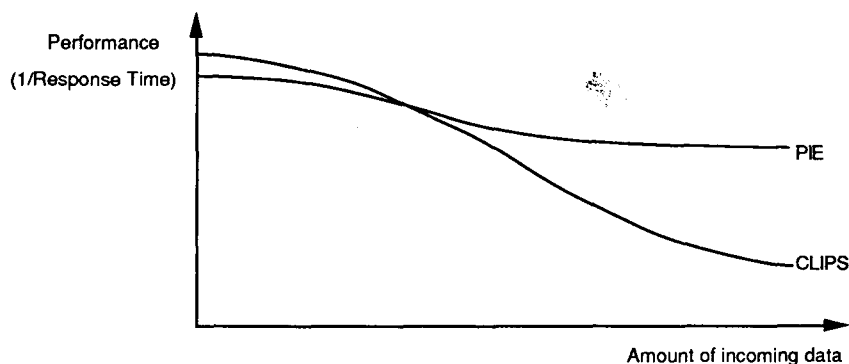

Figure 4: The predicted relative preformance of PIE and CLIPS vs the amount of incoming data

6. Conclusion

A PIE architecture has been defined based on extending CLIPS to accommodate modular, hierarchical data structures. The concept of an active rule set and associated working memory subset was introduced to "focus the production system attention". The resultant architecture provides programmers with more flexibility in defining the control strategy of the inference engine. This structure appears promising for systems with real-time constraints, and provides a clear delineation of the control structure from the rule base. This architecture is currently being implemented for embedded applications.

Some applications might have requirements that exceed the capability of PIE in a sequential processing environment. Performance could be improved by use of parallel computing. PIE modularity and independence of control for each goal seem to lend PIE to a macro-level of parallelism, permitting partitioning and implementation on a parallel architecture.

Other areas of interest include extension to temporal reasoning [10], debugging environments for PIE-like architectures, and integration of the knowledge base with the more conventional database structures.

Acknowledgements

The authors would like to thank Dr. Charlie Crummer, Dr. Russ Abbott, and Dr. Jim Hamilton who made many valuable comments towards this paper.

References

1. Michie, D., "Expert Systems," The Computer Journal, vol. 23, 1980, p. 369-376.

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

3. 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.

4. Nilsson, N., Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California, 1980.

5. Winston, P., Artificial Intelligence, Second Edition, Addison Wesley Publishing Company, Jul 1984.

6. 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.

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

8. Mettry, W., "An Assessment of Tools for Build Large Knowledge-Base Systems", AI Magazine, Winter 1987, p. 81-89.

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

10. Laffey, T., Cox, P., Schmidt, J., Kao, S., and Read, J., "Real-Time Knowledge-Based Systems," AI Magazine, Spring 1988, p. 27-45.