LESS THAN GENERAL PRODUCTION SYSTEM ARCHITECTURES[1]

Douglas B. Lenat and John McDermott
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Many of the recent expert rule-based systems [Dendral, Mycin, AM, Pecos] have architectures that differ significantly from the simple domain-independent architectures of "pure" production systems. The purpose of this paper is to explore, somewhat more systematically than has been done before, the various ways in which the simplicity constraints can be relaxed, and the benefits of doing so. The most significant benefits arise from three sources: *(i)* the grain size of a typical rule can be increased until it captures a unit of advice which is meaningful in that system's task domain, *(ii)* the interpreter can become accessible to the rules and thus become dynamically modifiable, and *(iii)* meaningful permanent Knowledge can be stored in data memories, not just within productions. Although there are costs associated with relaxing the simplicity constraints, for many task domains the benefits outweigh the costs.

1. Introduction

Most AI programs employ "search" in some form. Our earliest experiences [translation; chess] taught us the futility of unconstrained searching. Since that time, AI researchers have experimented with various methodologies for limiting search. Some of these [Nilsson, 1971] have been *externally imposed* procedures: small programs that help to direct an otherwise-unconstrained search (e.g., heuristic evaluation functions that guide the expansion of nodes in a search tree; alpha/beta cutoff detectors in an and/or game tree). Another approach has been to scrap the search tree paradigm entirely and to program in such a way that search is *inherently* constrained. One such technique is to use a network of semantically related nodes (e.g., Woods' ATN). Another is to encode the program as a production system (PS).

One of the earliest uses of production systems in AI was by Newell [1973]. The architecture that he proposed and that has since been emulated by several researchers has the following familiar components: *(i)* a single production memory containing an indefinitely large number of productions (condition/action rules) in which all of the system's permanent knowledge is encoded; *(ii)* a single data memory (called working memory) containing assertions that are in the process of being assimilated or modified by the system; and *(iii)* an interpreter that repeatedly matches the conditions in each production against the assertions in data memory, selects from the set of satisfied productions one production to fire, and then executes the actions specified in that production.

The basic appeal of production systems is that they constrain search in a context-dependent (one might say "natural") way. A production's conditions are satisfied if and only if its actions are likely to be relevant to the current state of the world (as represented in data memory). The interpreter, in order to select a production to fire, has only to search among pieces of knowledge that are possibly relevant at that moment. This is what we meant earlier when we indicated that PSs do not follow the "exploring a search space" paradigm; rather, they carry out myriad small parallel searches, first for relevant pieces of knowledge, then for the most promising one of these to use.

Various constraints on the production system architecture (implicit in the description above) were imposed by experimenters in order to insure that it would be sufficiently general to enable a PS to able to behave in a reasonable fashion in any domain. Much of the current work in AI is focused on the development of large, knowledge-based systems which are expert in one very sophisticated -- but also very specialized — task (e.g., identification of mass spectrograms, anti-microbial therapy selection, speech understanding, etc.). One way of encoding knowledge in such systems is as a collection of pattern-invoked rules, and thus the production system architecture appears to be a viable candidate for supporting such systems. But is the *domain-independent* architecture described above the best production system architecture to use for such systems?

It has become obvious within the last few years, we think, that it is not. Empirical justification for this conclusion is provided by the recent development of what have been

Historically, the constraints were imposed so that a production system would be able to simulate the cognitive behavior of humans — who, at least purportedly, are able to behave in a reasonable fashion in all domains.

This work was supported in parl by tht Defense Advanced Research Projects Agency (F44620-73-C-0074) end monitored by tht Air Force Office of Scientific Research

called rule-based inference systems [Feigenbaum, 1971; Buchanan, 1974; Shortliffe, 1974; Davis, 1976; Lenat, 1976; McCracken, 1977; Barstow, 1977; Duda, 1977]. Most of these systems are closely related to production systems: they divide the world into data and condition/action rules, and make use of an interpreter that repeatedly matches rules against data and then executes the actions of one or more of the rules whose conditions are satisfied. But because each of these systems was designed for a single, well-defined task, it was unnecessary for their designers to impose all of the constraints on the form of data memory, production memory, and the interpreter that are found in most domain-independent systems. The purpose of this paper is to explore, somewhat more systematically than has been done before, the ways in which these domain-independent constraints can be relaxed, and the implications of doing so[4].

2. Data Memories

Domain-independent architectures, though they limit the number of elements in data memory (DM), allow those elements to be arbitrarily complex. Consequently, the amount of knowledge that can be stored in DM is theoretically unlimited. However, other constraints placed on these architectures make it very difficult to use DM as anything but an intermediate memory containing rather simple assertions. The memory is forced to be intermediate (ephemeral, temporary, "working") because no data element can be linked directly to any other data element; their only relationship is that they have been deposited in DM by productions sensitive to the current context. The data elements are forced to be simple assertions because no operations on data sub-elements are permitted unless those sub-elements have been matched by the condition part of a production and because all of the operations performed on these structures must be indicated explicitly in productions (except for a few operations that are performed indiscriminately by the interpreter on all elements).

These restrictions can be weakened by *(i)* permitting many separate DMs to exist, each defined by a distinct set of primitive operations that can access and modify it, and (a) allowing some DMs to contain statically interrelated elements. Thus DMs could be of two types. Some could hold permanent knowledge and thus would have a significance similar to production memory (the set of productions), but their structure as well as their content could convey information. Other DMs could be working (intermediate) memories; that is, they would serve the same focusing function that the lone DM serves in domain-independent architectures.

[3] For a discussion of how the nature of the "mathematical discovery" task influenced the design of AM, see [Lenat & Harris 1977].

For a discussion of the various dimensions along which existing production systems differ, see [Davis & King 1975].

The motivation for multiple DMs is that certain tasks can be viewed most naturally as developing and enlarging a body of declarative knowledge. Consider a PS that is to discover new mathematics, or one that is to do research in organic chemistry. Such a system might have two DMs, one containing highly inter-related domain-dependent facts (representing the existing theory), the other containing an ever-changing set of goals for augmenting the permanent knowledge stored in the first DM. There are at least two gains in using multiple DMs. First, since each DM has its own set of proper operations, modifications to the data base can be specified at a higher, more natural level (and in some cases need not be specified at all). Secondly, because the relationships among pieces of knowledge in DMs correspond to relationships in the task domain, the system's DMs are statically intelligible.

There are, of course, some negative consequences that result from not storing all permanent knowledge within the production rules. In particular, knowledge within productions is directly pattern-invokable, while knowledge within a DM is not. Hence, some processes will take longer, will involve inferencing, search, etc. In fact, there may be cases where relevant knowledge buried within the DM will simply not be successfully retrieved, due to the enormity of the search needed to find it. Thus there are restrictions on when DMs may appropriately be used to hold permanent knowledge. For tasks in which speed is crucial, or for tasks in which knowledge cannot be structured in a way that facilitates search, such a use will be inappropriate. As a final note, it should be pointed out that weakening the constraints on DM will affect the rules and interpreter design as well. Additional rules may be needed just to manipulate knowledge in the DMs, reorgani?e it, reason about it, etc.

3. Production Memories

3.1. Multiple Production Memories

The reason that only a single production memory (PM) is allowed in domain-independent architectures is, as with DM, to insure that all relevant knowledge is always immediately accessible. But again, for some domains, this goal can be achieved even though the constraint is relaxed. Multiple production memories could be used in two ways. Each production memory could be paired with its own data memory. Or production memories could be hierarchically ordered so that although all rules would be matched against the same data, not all rules would be matched on every cycle; rather, a subset of productions could be specified (e.g., by an action) to be the ones to be matched against on the next cycle.

The only justification for the use of multiple PMs is that it allows the same set of conditions to be associated with several different sets of actions. For the most part, this is simply a programming convenience: if enough is known to distribute productions among many memories, then enough is known to add conditions to the productions so that only those productions that are actually appropriate will fire.

This "convenience" may, nevertheless, be of considerable help since the amount of effort required to provide sufficiently discriminating conditions might be immense. In some cases, moreover, it may be impossible to provide the necessary conditions unless productions that attend to different aspects of the environment can *never* be satisfied on the same cycle.

Analogues of the dangers of multiple DMs are present when multiple PMs are allowed. Relevant knowledge might not be retrieved (because it happens to be in a production inside a PM which is not currently active). Thus there are restrictions on when multiple PMs may appropriately be used. They are appropriate when enough is Known about the task domain to make it clear when particular sets of productions can be safely "deactivated".

3.2. Productions Treated as Data

If DMs are allowed to hold permanent knowledge, then the distinction between PM and DM becomes less sharply defined. The demarcation can be further blurred by relaxing the restriction (imposed on the domain-independent architectures) that PM cannot be read directly.[pr] The reason for this restriction is somewhat difficult to infer. It appears to be due to a view of productions which holds that they are not themselves part of a system's knowledge, but only the vehicle of such knowledge. The obvious way to relax this restriction is simply to treat PMs as full- fledged DMs.

The ability to read PM would be particularly useful in tasks that call for reasoning about the knowledge being employed to do the task. Some math and natural science tasks are of this character. Also, any PS which is designed to nontrivially *team* can make use of this ability. It could have a set of productions whose specialty is dealing with other productions; thus the system could inspect, modify, augment, correlate, compare, and delete productions in the course of adding new productions to its PM.

3.3. Complex Rules

Many constraints are placed on the form of the condition and action sides of productions by the domain-independent architectures. Again, these restrictions are imposed to insure that all relevant knowledge will always be available to the system. The main restriction that is placed on the condition side of productions is that condition elements be forms (templates), and that the match be nothing but a simple membership test on a working memory of limited size. This restriction allows the match to be very efficient and thus insures that changes to working memory that indicate that some piece of knowledge has become relevant will be quickly noticed. Alternatives to simple tests of set membership abound. The match can be performed against the data elements in several different data memories (some of which might have an extremely complex structure). The membership match can be extended to "almost member"; that is, partial matching may be allowed. Or the testing functions may be extended to include arbitrarily complex tests on DMs, and may even involve calling on a production subsystem to determine if a currently unsatisfied production can be satisfied [e.g., see Barstow, 1977].

The main restriction that is placed on the action side of productions is the restriction that action elements be unconditional. The requirement of unconditionally guarantees that only a few actions are performed on each cycle (since complex actions require tests) and thus that all decisions made by the system are made in a global context (i.e., in the context of all possibly relevant information). If conditionally is allowed on the action part of a rule, then complex actions can be performed (including evoking production sub-systems).

The basic effect of these condition-side and action-side relaxations would be to increase the "grain size" of each production. Instead of a quickly locatable and quickly executable stimulus/response coupling, a production could represent a sophisticated, relatively self-sufficient chunk of domain-dependent knowledge, suitable for guiding an actor in a particular domain. Consider, for example, one of AM's 253 rules:

```
IF the current task is to find examples of concept X,
   and X is a predicate,
   and >100 items are known to be in Domam(X),
   and over 10 cpu sees, have been spent on this task,
   and there are no more relevant rules to fire,
   and X has returned "True" at least once,
   and X has returned "True" under 5% of the time,
THEN consider the following as a future task:
   Defining new predicates, similar to X,
   which are generalizations of X,
   because X is rarely satisfied,
   hence a slightly less restrictive concept may be
         more interesting than X.
```

This rule embodies the piece of advice that a predicate should be generalized (weakened) if it returns "False" too often. AM used this rule, for example, after "Equality-of-sets" was found to be rarely satisfied, and one of the resulting generalizations was the valuable concept "Same-length".

Many of us can understand and appreciate this rule, in its entirety, immediately upon seeing it. If it were split into a dozen tiny rules, heavily intercoupled, it would become much less understandable. For systems whose rules are to be formulated by experts from the task domain (e.g., physicians formulating MYCIN rules [see Shortliffe, 1974]), a coarse level of granularity will be helpful. Each

5 Most domain-independent PS architectures have some form of this constraint, although many do allow rules to access other rules under certain conditions (e.g., to access the rule fired on the previous cycle.)

6 The requirement makes sense for cognitive simulation: humans do not appear to have direct access to all of the rules that they have.

production will be a meaningful unit and will be easy for other experts in that domain to understand . For some task domains, this heightened intelligibility will more than pay for any costs that the larger grain size introduces.

One of these costs is the large amount of change that could occur "all at once" in the DMs during a single uninterruptable cycle. This cost is surely unacceptable to a system that has to respond to changes in an environment that is not fully under its control. For a system that is in control, however, the problem becomes manageable. Either cycle time is never a concern or it is of concern at particular (isolatable) times. In this latter case, the PS can be provided with sufficient knowledge to control the length of the cycle. This knowledge can be encoded directly in the productions (by making each appropriately complex) or it can be supplied by a DM.

4. Interpreters

In domain-independent architectures the interpreter is a single program that sits above (out of reach of) the productions. It has its own memory (IM) which cannot be read or written into by productions and in which it stores stale information. Its three functions are (i) to search for satisfied productions by applying a pre-specified set of contextually independent matching rules, (ii) to select one production to fire on the basis of a pre-specified set of conflict resolution rules that make use of the state information[8], and *(Ui)* to execute the actions of the production selected.

These constraints on the interpreter can be relaxed by (i) treating its memory as a DM that can be examined and modified by the system's productions, and *(ii)* treating the interpreter itself as a DM that can be examined and modified by the system's productions. By treating IM as a DM, the selection of the set of productions to fire on a given cycle could be made more intelligently; that is, all of the system's knowledge could he used[9]. Making the interpreter itself dynamically modifiable would allow its behavior to be governed by the current state of the world. Different sets of matching rules could be used in different contexts: simple tests of class membership where that would be appropriate, or partial matching, or arbitrarily complex tests on data memories (including itself). Different sets of conflict resolution rules could also he used: The interpreter could use rules appropriate to its current situation; its rules could be more or less selective: allowing one, several, or all satisfied productions to fire on a given cycle.

This sort of relaxation of constraints is clearly in line with our avowed reason for using PSs: to constrain search in helpful ways. By having the interpreter policies vary, the PS will spend more or less time constraining search, as it is or is not critical at that moment. There is also a certain symmetry which is open to us now: the interpreter can be viewed as containing domain-dependent wisdom at one level higher than (meta to) the knowledge contained in PM. Tor example, a theorem-proving PS might have rules like "(A and B) --> A", and might rely upon sophisticated rules in the interpreter to keep the execution from blowing up combinatorially.

Of course the price paid for this sophistication is the cost of dynamically reprogramming the interpreter at appropriate times. However, for a system whose task places only a limited number of different processing demands on its interpreter, the time spent in modifying the interpreter would be more than offset by the gain in flexibility which would allow different parts of the task to be accomplished as effectively as possible.

5. Concluding Remarks

Since production systems constrain search in a context dependent way, they may prove to be a very valuable tool. We have attempted to show some of the alternatives open to a production system designer who wants to tailor the system he is designing to the requirements of the particular task domain within which his system will function. These alternatives consist primarily in relaxing constraints (imposed by the domain-independent PS designers) on the form of data memory, production memory, and the interpreter. If it is the case that different tasks make different kinds of representational demands, then it seems clear that such relaxations can have beneficial effects. Of the benefits that we suggested above, three seem to us to be of particular note: those arising from giving productions a larger grain size; those arising from allowing productions to modify the interpreter; and those arising from allowing permanent knowledge to be stored in memories other than PM.

Systems whose task is to be (or become) expert in some particular domain so that they can be used in place of human experts (or so that a model of the knowledge necessary in some domain can be explored) can be constructed most easily if the grain size of the knowledge they store is the optimal grain size for that domain. Presumably the most desirable grain size is displayed by human experts when they communicate with one another. But when experts in domains such as chemistry, medicine, or mathematics communicate, the rules that they formulate are very much more complex than the rules that can be encoded as single productions in domain-independent systems.

[7] "Understand" has three components in this case: the domain expert should find it easier to write such a large-grained rule, easier to comprehend it if shown it statically, and better able to follow the rule dynamically when the PS actually selects and executes it.

8 For a discussion of the problems of conflict resolution in domain-independent systems, see [McDermott & Forgy, 1977].

[9] E.g., AM's "agenda" job-list is the IM, the working memory for that program's scheduler. Yet the agenda is maintained and replenished directly by the rules of that program.

The second benefit arises from the "dynamically modifiable interpreter" idea. By viewing the interpreter as a data structure that can be read *and* written into by productions (and by viewing PM in the same way), opportunities are opened up for changing the performance of the system on the basis of the Kind of subtask in which it is engaged. In any task domain, there are certain situations in which highly constrained search is appropriate and certain situations in which all relevant knowledge must be examined before any action can be taken. By making the interpreter dynamically modifiable, the search behavior could be tailored to the subtask. MYCIN provides something analogous to this by making meta-rules available, rules that can be used to select a subset of the set of satisfied productions to fire. However, a more general capability would provide more strength. Since the capability of modifying the interpreter would give the system control over the amount of processing done on each cycle, for task domains in which short cycle time is sometimes important and sometimes not, the system could tailor its cycle time to the subtask.

The third benefit arises from the fact that in some task domains, either cycle time is not an important concern or it can be controlled by varying the complexity of productions or by making the interpreter dynamically modifiable. In such cases, there may be a better way to store some knowledge than by means of autonomous rules. In some domains, a great deal is known about the structure (interconnectedness) of many of the concepts relevant to that domain. The availability of multiple data memories would allow this knowledge of the structure to be stored implicitly. Note that there is no constraint that DM entries be declarative; (e.g., a scientific theory may be best represented within a DM, even though much of it may be procedural.)

Our final conclusion is that these benefits often far outweigh the costs. When nothing is known about the domain for which a PS is designed, one may opt for the domain-independent architecture. But when much is known about the domain, it is cost-effective to build some of this knowledge into the architecture. In the limit, when we are designing a system with one specific task in mind, relaxing the old simplicity constraints is clearly indicated.

Acknowledgments

This research builds upon earlier work by the authors, some of it in conjunction with C. Forgy and G. Harris at CMU. To both of them go our sincere appreciation. We also wish to thank D. Barstow, who contributed several excellent suggestions during the planning of this paper.

References

Barstow, D., Automatic construction of algorithms and data structures using a knowledge base of programming rules, Ph.D. Dissertation, Artificial Intelligence Laboratory, Stanford University, 1977.

Buchanan, B., Scientific theory formation by computer, NATO Advanced Study Institute on Computer Oriented Learning Processes, Bonas, France, 1974.

Davis, P., Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases, Ph.D. dissertation, SAIL A1M-271, Artificial Intelligence Laboratory, Stanford University, July, 1976.

Davis, R., and King, J., An overview of production systems, Report STAN-CS-75-524, SAIL Memo AIM-271, Stanford U. CS Department, 1975.

Duda, R., Hart, P., Nillson, N., Sutherland, G., Semantic network representations in rule-based inference systems, in Waterman and Hayes-Roth (eds.), *Pattern-Directed Inference Systems,* Academic Press, 1977.

Feigenbaum, E., Buchanan, B., and Lederberg, J., On generality and problem solving: a case study using the dendral program, in' Meltzer and Michie (eds.), *Machine Intelligence* 6, 1971, pp. 165-190.

Lenat, D., AM: an artificial intelligence approach to discovery in mathematics as heuristic search, Ph.D. dissertation, SAIL AIM--286, Artificial Intelligence Laboratory, Stanford University, July, 1976. Jointly issued as Computer Science Dept. Report No. STAN-CS-76-570.

Lenat, D. and Harris, G., Designing a rule system that searches for scientific discoveries, in Waterman and Hayes-Roth (eds.), *Pattern-Directed Inference Systems,* Academic Press, 1977.

McCracken, D., A parallel production system architecture for speech understanding, CMU CS Dept. Ph.D. Thesis, 1977.

McDermott, J. and Forgy, C, Production system conflict resolution strategies, in Waterman and Hayes-Roth (eds.), *Pattern-Directed Inference Systems,* Academic Press, 1977.

Newell, A., Production Systems: Models of Control Structures, May, 1973 CMU Report, also published in (W.G. Chase, ed.) *Visual Information Processing,* NY: Academic Press, Chapter 10, pp. 463-526.

Nilsson, N., *Problem Solving Methods in Artificial Intelligence,* McGraw Hill, N.Y., 1971.

Shortliffe, E., MYCIN -- A rule-based computer program for advising physicians regarding antimicrobial therapy selection, Ph.D. Dissertation, SAIL AIM-251, Artificial Intelligence Laboratory, Stanford University, October, 1974.