2

The Origin of Rule-Based Systems in AI

Randall Davis and Jonathan J. King

Since production systems (PS's) were first proposed by Post (1943) as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. We present an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other and to other methodologies. Accordingly, we use the term *production system* in a broad sense and show how most systems that have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally based techniques, but also with reference to more recent developments in programming and the organization of data and knowledge bases.

This chapter begins by offering a review of the essential structure and function of a PS, presenting a picture of a "pure" PS to provide a basis for subsequent elaborations. Current views of PS's fall into two distinct classes, and we shall demonstrate that this dichotomy may explain much of the existing variation in goals and methods. This is followed by some speculations on the nature of appropriate and inappropriate problem domains for PS's—i.e., what is it about a problem that makes the PS methodology appropriate, and how do these factors arise out of the system's basic structure and function? Next, we review characteristics common to all systems, explaining how they contribute to the basic character and noting their

This chapter is based on an article taken with permission from *Machine Intelligence 8: Machine Representations of Knowledge*, edited by E. W. Elcock and D. Michie, published in 1977 by Ellis Horwood Ltd., Chichester, England.

interrelationships. Finally, we present a taxonomy for PS's, selecting four dimensions of characterization and indicating the range of possibilities suggested by recent efforts.

Two points of methodology should be noted. First, we make frequent reference to what is "typically" found, and what is "in the spirit of things." Since there is really no one formal design for PS's and recent implementations have explored variations on virtually every aspect, their use becomes more an issue of a programming *style* than of anything else. It is difficult to exclude designs or methods on formal grounds, and we refer instead to an informal but well-established style of approach. A second, related point is important to keep in mind as we compare the capabilities of PS's with those of other approaches. Since it is possible to imagine coding any given Turing machine in either procedural or PS terms [see Anderson, (1976) for a formal proof of the latter], in the formal sense their computational power is equivalent. This suggests that, given sufficient effort, they are ultimately capable of solving the same problems. The issues we wish to examine are not, however, questions of absolute computational power but of the impact of a particular methodology on program structure, as well as of the relative ease or difficulty with which certain capabilities can be achieved.

2.1 "Pure" Production Systems

A production system may be viewed as consisting of three basic components: a set of rules, a data base, and an interpreter for the rules. In the simplest design a rule is an ordered pair of symbol strings, with a left-hand side and a right-hand side (LHS and RHS). The rule set has a predetermined, total ordering, and the data base is simply a collection of symbols. The interpreter in this simple design operates by scanning the LHS of each rule until one is found that can be successfully matched against the data base. At that point the symbols matched in the data base are replaced with those found in the RHS of the rule and scanning either continues with the next rule or begins again with the first. A rule can also be viewed as a simple conditional statement, and the invocation of rules as a sequence of actions chained by *modus ponens*.

2.1.1 Rules

More generally, one side of a rule is *evaluated* with reference to the data base, and if this succeeds (i.e., evaluates to TRUE in some sense), the action specified by the other side is performed. Note that *evaluate* is typically taken

to mean a passive operation of "perception," or "an operation involving only matching and detection" (Newell and Simon, 1972), while the action is generally one or more conceptually primitive operations (although more complex constructs are also being examined; see Section 2.4.9). As noted, the simplest evaluation is a matching of literals, and the simplest action, a replacement.

Note that we do not specify which side is to be matched, since either is possible. For example, given a grammar written in production rule form,[1]

$$
\begin{aligned}
&S \rightarrow A\ B\ A\\
&A \rightarrow A\ 1\\
&A \rightarrow 1\\
&B \rightarrow B\ 0\\
&B \rightarrow 0
\end{aligned}
$$

matching the LHS on a data base that consists of the start symbol S gives a generator for strings in the language. Matching on the RHS of the same set of rules gives a recognizer for the language. We can also vary the methodology slightly to obtain a top-down recognizer by interpreting elements of the LHS as goals to be obtained by the successful matching of elements from the RHS. In this case the rules "unwind." Thus we can use the same set of rules in several ways. Note, however, that in doing so we obtain quite different systems, with characteristically different control structures and behavior.

The organization and accessing of the rule set is also an important issue. The simplest scheme is the fixed, total ordering already mentioned, but elaborations quickly grow more complex. The term *conflict resolution* has been used to describe the process of selecting a rule. These issues of rule evaluation and organization are explored in more detail below.

2.1.2 Data Base

In the simplest production system the data base is simply a collection of symbols intended to reflect the state of the world, but the interpretation of those symbols depends in large part on the nature of the application. For those systems intended to explore symbol-processing aspects of human cognition, the data base is interpreted as modeling the contents of some memory mechanism (typically short-term memory, STM), with each symbol representing some "chunk" of knowledge; hence its total length (typically around seven elements) and organization (linear, hierarchical, etc.) are im-

[1]One class of production systems we will not address at any length is that of grammars for formal languages. While the intellectual roots are similar (Floyd, 1961; Evans, 1964), their use has evolved a distinctly different flavor. In particular, their nondeterminism is an important factor that provides a different perspective on control and renders the question of rule selection a moot point.

portant theoretical issues. Typical contents of STM for psychological models are those of PSG (Newell, 1973), where STM might contain purely content-free symbols such as:

```
QQ
(EE FF)
TT
```

or of VIS (Moran, 1973a), where STM contains symbols representing directions on a visualized map:

```
(NEW C-1 CORNER WEST L-1 NORTH L-2)
(L-2 LINE EAST P-2 P-1)
(HEAR NORTH EAST % END)
```

For systems intended to be knowledge-based experts, the data base contains facts and assertions about the world, is typically of arbitrary size, and has no *a priori* constraints on the complexity of organization. For example, the MYCIN system uses a collection of quadruples, consisting of an associative triple and a *certainty factor* (CF), which indicates (on a scale from -1 to 1) how strongly the fact has been confirmed ($CF > 0$) or disconfirmed ($CF < 0$):

```
(IDENTITY ORGANISM-1 E.COLI .8)
(SITE CULTURE-2 BLOOD 1.0)
(SENSITIVE ORGANISM-1 PENICILLIN -1.0)
```

As another example, in the DENDRAL system (Feigenbaum et al., 1971; Lindsay et al., 1980) the data base contains complex graph structures that represent molecules and molecular fragments.

A third style of organization for the data base is the "token stream" approach used, for example, in LISP70 (Tesler et al., 1973). Here the data base is a linear stream of tokens, accessible only in sequence. Each production in turn is matched against the beginning of the stream (i.e., if the first character of a production and the first character of the stream differ, the whole match fails), and if the rule is invoked, it may act to add, delete, or modify characters in the matched segment. The anchoring of the match at the first token offers the possibility of great efficiency in rule selection since the productions can be "compiled" into a decision tree that keys off sequential tokens from the stream. A very simple example is shown in Figure 2-1.

Whatever the organization of the data base, one important characteristic that should be noted is that it is the sole storage medium for all state variables of the system. In particular, unlike procedurally oriented languages, PS's do not provide for separate storage of control state information—there is no separate program counter, pushdown stack, etc.—and all information to be recorded must go into the single data base. We refer to this as *unity of data and control store* and examine some of its implications below. This store is, moreover, universally accessible to every rule in the

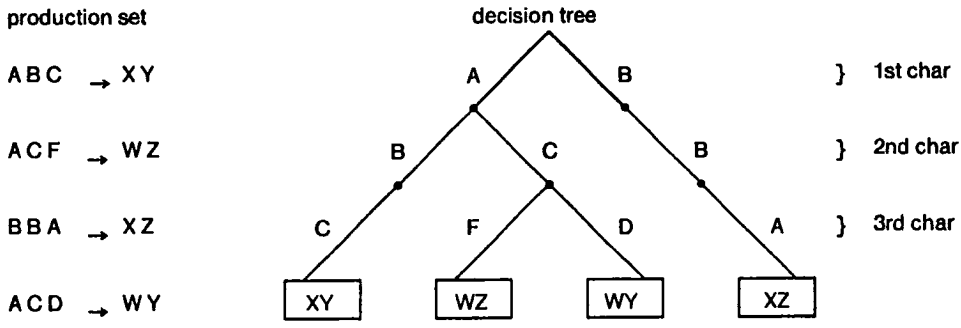

FIGURE 2-1 Production rule and decision tree representations of a simple system that replaces sequences of three symbols in the data base with sequences of two others.

system, so that anything put there is potentially detectable by any rule. We shall see that both of these points have significant consequences for the use of the data base as a communication channel.

2.1.3 Interpreter

The interpreter is the source of much of the variation found among different systems, but it may be seen in the simplest terms as a *select-execute* loop in which one rule applicable to the current state of the data base is chosen and then executed. Its action results in a modified data base, and the select phase begins again. Given that the selection is often a process of choosing the first rule that matches the current data base, it is clear why this cycle is often referred to as a *recognize-act*, or *situation-action*, loop. The range of variations on this theme is explored in Section 2.5.3 on control cycle architecture.

This alternation between selection and execution is an essential element of PS architecture, which is responsible for one of its most fundamental characteristics. By choosing each new rule for execution on the basis of the total contents of the data base, we are effectively performing a complete reevaluation of the control state of the system at every cycle. This is distinctly different from procedurally oriented approaches in which control flow is typically the decision of the process currently executing and is commonly dependent on only a small fraction of the total number of state variables. PS's are thus sensitive to any change in the entire environment, and potentially responsive to such changes within the scope of a single execution cycle. The price of such responsiveness is, of course, the computation time required for the reevaluation.

An example of one execution of the recognize-act loop for a greatly

simplified version of Newell's PSG system will illustrate some of the foregoing notions. The production system, called PS.ONE, is assumed for this example to contain two productions, PD_1 and PD_2. We indicate this as follows:

PS.ONE: (PD_1 PD_2)

PD_1: (DD AND (EE) → BB)
PD_2: (XX → CC DD)

PD_1 says that if the symbol DD and some expression beginning with EE, i.e., (EE . . .), is found in STM, then insert the symbol BB at the front of STM. PD_2 says that if the symbol XX is found in STM, then first insert the symbol CC, then the symbol DD, at the front of STM.

The initial contents of STM are

STM: (QQ (EE FF) RR XX SS)

This STM is assumed to have a fixed maximum capacity of five elements. As new elements are inserted at the front (left) of STM, therefore, other elements will be lost (forgotten) off the right end. In addition, elements accessed when matching the condition of a rule are *refreshed* (pulled to the front of STM) rather than replaced.

The production system scans the productions in order: PD_1, then PD_2. Only PD_2 matches, so it is evoked. The contents of STM after this step are

STM: (DD CC XX QQ (EE FF))

PD_1 will match during the next cycle to yield

STM: (BB DD (EE FF) CC XX)

completing two cycles of the system.

2.2 Two Views of Production Systems

Prior work has suggested that there are two major views of PS's, characterized on one hand by psychological modeling efforts (PSG, PAS II, VIS, etc.) and on the other by performance-oriented, knowledge-based expert systems (e.g., MYCIN, DENDRAL). These distinct efforts have arrived at similar methodologies while pursuing differing goals.

The psychological modeling efforts are aimed at creating a program that embodies a theory of human performance of simple tasks. From the performance record of experimental human subjects, the modeler formulates the minimally competent set of production rules that is able to reproduce the behavior. Note that "behavior" here is meant to include *all* aspects of human performance (mistakes, the effects of forgetting, etc.),

including all shortcomings or successes that may arise out of (and hence may be clues to) the "architecture" of human cognitive systems.[2]

An example of this approach is the PSG system, from which we constructed the example above. This system has been used to test a number of theories to explain the results of the Sternberg memory-scanning tasks (Newell, 1973), with each set of productions representing a different theory of how the human subject retains and recalls the information given to him or her during the psychological task. Here the subject first memorizes a small subset of a class of familiar symbols (e.g., digits) and then attempts to respond to a symbol flashed on a screen by indicating whether or not it was in the initial set. His or her response times are noted.

The task was first simulated with a simple production system that performed correctly but did not account for timing variations (which were due to list length and other factors). Refinements were then developed to incorporate new hypotheses about how the symbols were brought into memory, and eventually a good simulation was built around a small number of productions. Newell has reported (Newell, 1973) that use of a PS methodology led in this case to the novel hypothesis that certain timing effects are caused by a decoding process rather than by a search process. The experiment also clearly illustrated the possible tradeoffs in speed and accuracy between differing processing strategies. Thus the PS model was an effective vehicle for the expression and evaluation of theories of behavior.

The performance-oriented expert systems, on the other hand, start with productions as a representation of knowledge about a task or domain and attempt to build a program that displays competent behavior in that domain. These efforts are not concerned with similarities between the resulting systems and human performance (except insofar as the latter may provide a possible hint about ways to structure the domain or to approach the problem or may act as a yardstick for success, since few AI programs approach human levels of competence). They are intended simply to perform the task without errors of any sort, humanlike or otherwise. This approach is characterized by the DENDRAL system, in which much of the development has involved embedding a chemist's knowledge about mass spectrometry into rules usable by the program, without attempting to model the chemist's thinking. The program's knowledge is extended by adding rules that apply to new classes of chemical compounds. Similarly, much of the work on the MYCIN system has involved crystallizing informal knowledge of clinical medicine in a set of production rules.

Despite the difference in emphasis, researchers in both fields have

[2]For example, the critical evaluation of EPAM must ultimately depend not on the interest it may have as a learning machine, but on its ability to explain and predict phenomena of verbal learning (Feigenbaum, 1963). These phenomena include stimulus and response generalization, oscillation, retroactive inhibition, and forgetting—all of which are "mistakes" for a system intended for high performance but are important in a system meant to model human learning behavior.

been drawn to PS's as a methodology. For the psychological modelers, production rules offer a clear, formal, and powerful way of expressing basic symbol-processing acts that form the primitives of information-processing psychology (cf. Newell and Simon, 1972). For the designer of knowledge-based systems, production rules offer a representation of knowledge that can be accessed and modified with relative ease, making it quite useful for systems designed for incremental approaches to competence. For example, much of the MYCIN system's capability for explaining its actions is based on the representation of knowledge as individual production rules. This makes the knowledge far more accessible to the program itself than it might be if it were embodied in the form of ALGOL-like procedures. As in DENDRAL, the modification and upgrading of the system occur via incremental modification of, or addition to, the rule set.

Note that we are suggesting that it is possible to view a great deal of the work on PS's in terms of a unifying formalism. The intent is to offer a conceptual structure that can help organize what may appear to be a disparate collection of efforts. The presence of such a formalism should not, however, obscure the significant differences that arise from the various perspectives. For example, the decision to use RHS-driven rules in a goal-directed fashion implies a control structure that is simple and direct but relatively inflexible. This offers a very different programming tool than the LHS-driven systems do. The latter are capable of much more complex control structures, giving them capabilities much closer to those of a complete programming language. Recent efforts have begun to explore the issues of more complex, higher-level control within the PS methodology (see Section 2.4.9).

Production systems are seen by some as more than a convenient paradigm for approaching psychological modeling—rather as a methodology whose power arises out of its close similarity to fundamental mechanisms of human cognition. Newell and Simon (1972, pp. 803–804, 806) have argued that human problem-solving behavior can be modeled easily and successfully by a production system because it in fact is being generated by one:

> We confess to a strong premonition that the actual organization of human programs closely resembles the production system organization. . . . We cannot yet prove the correctness of this judgment, and we suspect that the ultimate verification may depend on this organization's proving relatively satisfactory in many different small ways, no one of them decisive.
>
> In summary, we do not think a conclusive case can be made yet for production systems as *the* appropriate form of [human] program organization. Many of the arguments . . . raise difficulties. Nevertheless, our judgment stands that we should choose production systems as the preferred language for expressing programs and program organization.

Observations such as this have led to speculation that the interest in pro-

duction systems on the part of those building high-performance knowledge-based systems is more than a coincidence. Some suggest that this is occurring because current research is (re)discovering what has been learned by naturally intelligent systems through evolution—that structuring knowledge in a production system format is an effective approach to the organization, retrieval, and use of very large amounts of knowledge.

The success of some rule-based AI systems does lend weight to this argument, and the PS methodology is clearly powerful. But whether or not this is a result of its equivalence to human cognitive processes and whether or not this implies that artificially intelligent systems ought to be similarly structured are still open questions, in our opinion.

2.3 Appropriate and Inappropriate Domains

Program designers have found that PS's easily model problems in some domains but are awkward for others. Let us briefly investigate why this may be so, and relate it to the basic structure and function of a PS.

We can imagine two very different classes of problems—the first is best viewed and understood as consisting of many independent states, while the second seems best understood via a concise, unified theory, perhaps embodied in a single law. Examples of the former include some views of perceptual psychology or clinical medicine, in which there are many states relative to the number of actions (this may be due either to our lack of a cohesive theory or to the basic complexity of the system being modeled). Examples of the latter include well-established areas of physics and mathematics, in which a few basic tenets serve to embody much of the required knowledge, and in which the discovery of unifying principles has emphasized the similarities in seemingly different states. This first distinction appears to be one important factor in distinguishing appropriate from inappropriate domains.

A second distinction concerns the complexity of control flow. At two extremes, we can imagine two processes, one of which is a set of independent actions and the other of which is a complex collection of multiple, parallel processes involving several dependent subprocesses.

A third distinction concerns the extent to which the knowledge to be embedded in a system can be separated from the manner in which it is to be used [also known as the controversy between declarative and procedural representations; see Winograd (1975) for an extensive discussion]. As one example, we can imagine simply stating facts, perhaps in a language like predicate calculus, without assuming how those facts will be employed. Alternatively, we could write procedural descriptions of how to accomplish

a stated goal. Here the use of the knowledge is for the most part predetermined during the process of embodying it in this representation.

In all three of these distinctions, a PS is well-suited to the first description and ill-suited to the latter. The existence of multiple, nontrivially different, independent states is an indication of the feasibility of writing multiple, nontrivial, modular rules. A process composed of a set of independent actions requires only limited communication between the actions, and, as we shall see, this is an important characteristic of PS's. The ability to state what knowledge ought to be in the system without also describing its use greatly improves the ease with which a PS can be written (see Section 2.4.9).

For the second class of problems (unified theory, complex control flow, predetermined use for the knowledge), the economy of the relevant basic theory makes for either trivial rules or multiple, almost redundant, rules. In addition, a complex looping and branching process requires explicit communication between actions, in which one action explicitly invokes the next, while interacting subgoals require a similarly advanced communication process to avoid conflict. Such communication is not easily supplied in a PS-based system. The same difficulty also makes it hard to specify in advance exactly how a given fact should be used.

It seems also to be the nature of production systems to focus upon the variations within a domain rather than upon the common threads that link different facts or operations. Thus, for example, the process of addition is naturally expressed via productions as n^2 rewrite operations involving two symbols (the digits being added). The fact that addition is commutative, or rather that there is a property of "commutativity" shared by all operations that we consider to be addition, is a rather awkward one to express in production system terms. This same characteristic may, conversely, be viewed as a capability for focusing on and handling significant amounts of detail. Thus, where the emphasis of a task is on recognition of large numbers of distinct states, PS's provide a significant advantage. In a procedurally oriented approach, it is both difficult to organize and troublesome to update the repeated checking of large numbers of state variables and the corresponding transfers of control. The task is far easier in PS terms, where each rule can be viewed as a "demon" awaiting the occurrence of a specific state.[3]

The potential sensitivity and responsiveness of PS's, which arise from their continual reevaluation of the control state, has also been referred to as the *openness* of rule-based systems. It is characterized by the principle that "any rule can fire at any time," which emphasizes the fact that at any point in the computation any rule could be the next to be selected, depending only on the state of the data base at the end of the current cycle. Compare this to the normal situation in a procedurally oriented language,

[3]In the case of one PS (DENDRAL) the initial, procedural approach proved sufficiently inflexible that the entire system was rewritten in production rule terms (Lindsay et al., 1980).

where such a principle is manifestly untrue: it is simply not typically the case that, depending on the contents of that data base, any procedure in the entire program could potentially be the next to be invoked.

We do not mean to imply that both approaches *couldn't* perform in both domains, but that there are tasks for which one of them would prove awkward and the resulting system unenlightening. Such tasks are far more elegantly accomplished in only one of the two methodologies. The main point is that we can, to some extent, formalize our intuitive notion of which approach seems more appropriate by considering two essential characteristics of any PS: its set of multiple, independent rules and its limited, indirect channel of interaction via the data base.

2.4 Production System Characteristics

Despite the range of variation in methodologies, there appear to be many characteristics common to almost all PS's. It is the presence of these and their interactions that contribute to the "nature" of a PS, its capabilities, deficiencies, and characteristic behavior.

The network of Figure 2-2 is a summary of features and relationships. Each box represents some feature, capability, or parameter of interest, with arrows labeled with +'s and −'s suggesting the interactions between them. This rough scale of facilitation and inhibition is naturally very crude, but does indicate the interactions as we see them. Figure 2-2 contains at least three conceptually distinct sorts of factors: (a) those fundamental characteristics of the basic PS scheme (e.g., indirect, limited channel, constrained format); (b) secondary effects (e.g., automated modifiability of behavior); and (c) performance parameters of implementation (e.g., visibility of behavior flow, extensibility), which are helpful in characterizing PS strengths and weaknesses.

2.4.1 Indirect, Limited Channel of Interaction

Perhaps the most fundamental and significant characteristic of PS's is their restriction on the interactions between rules. In the simplest model, a pure PS, we have a completely ordered set of rules, with no interaction channel other than the data base. The total effect of any rule is determined by its modifications to the data base, and hence subsequent rules must "read" there any traces the system may leave behind. Winograd (1975, p. 194) characterizes this feature in discussing global modularity in programming:

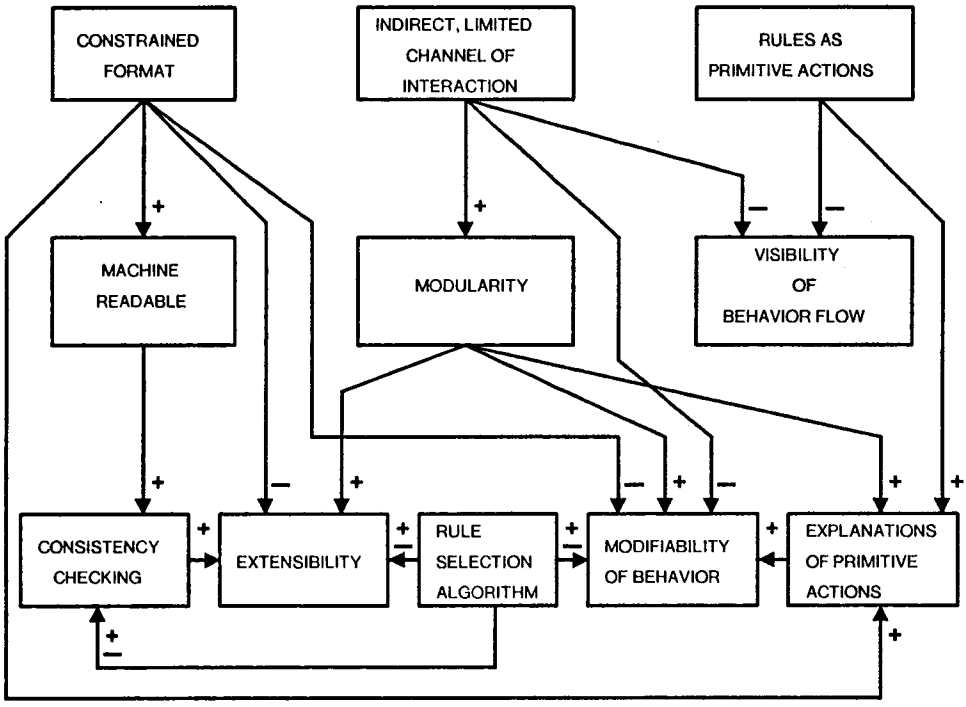

FIGURE 2-2 Basic features and relationships of a production system. Links labeled with a + indicate a facilitating relationship, while those labeled with a − indicate an inhibiting relationship.

> We can view production systems as a programming language in which all interaction is forced through a very narrow channel. . . . The temporal interaction [of individual productions] is completely determined by the data in this STM, and a uniform ordering regime for deciding which productions will be activated in cases where more than one might apply. . . . Of course it is possible to use the STM to pass arbitrarily complex messages which embody any degree of interaction we want. But the spirit of the venture is very much opposed to this, and the formalism is interesting to the degree that complex processes can be described without resort to such kludgery, maintaining the clear modularity between the pieces of knowledge and the global process which uses them.

While this characterization is clearly true for a pure PS, with its limitations on the size of STM, we can generalize on it slightly to deal with a broader class of systems. First, in the more general case, the channel is not so much

narrow as *indirect* and *unique*. Second, the kludgery[4] arises not from arbitrarily complex messages but from *specially crafted* messages, which force highly specific, carefully chosen interactions.

With reference to the first point, one of the most fundamental characteristics of the pure PS organization is that rules must interact indirectly through a single channel. Indirection implies that all interaction must occur by the effect of modifications written in the data base; uniqueness of the channel implies that these modifications are accessible to every one of the rules. Thus, to produce a system with a specified behavior, one must not think in the usual terms of having one section of code call another explicitly, but rather use an indirect approach in which each piece of code (i.e., each rule) leaves behind the proper traces to trigger the next relevant piece. The uniform access to the channel, along with the openness of PS's, implies that those traces must be constructed in the light of a potential response from any rule in the system.

With reference to Winograd's second point, in many systems the action of a single rule may, quite legitimately, result in the addition of very complex structures to the data base (e.g., DENDRAL; see Section 2.5). Yet another rule in the same system may deposit just one carefully selected symbol, chosen solely because it will serve as an unmistakable symbol for precisely one other (carefully preselected) rule. Choosing the symbol carefully provides a way of sending what becomes a private message through a public channel; the continual reevaluation of the control state assures that the message can take immediate effect. The result is that one rule has effectively called another, procedure style, and this is the variety of kludgery that is contrary to the style of knowledge organization typically associated with a PS. It is the premeditated nature of such message passing (typically in an attempt to "produce a system with specified behavior") that is the primary violation of the "spirit" of PS methodology.

The primary effect of this indirect, limited interaction is the development of a system that is strongly modular, since no rule is ever called directly. The indirect, limited interaction is also, however, the most significant factor that makes the behavior of a PS more difficult to analyze. This results because, even for very simple tasks, overall behavior of a PS may not be at all evident from a simple review of its rules.

To illustrate many of these issues, consider the algorithm for addition of positive, single-digit integers used by Waterman (1974) with his PAS II production system interpreter. First, the procedural version of the algorithm, in which transfer of control is direct and simple:

```
          add(m,n)   ::=
A]                   count←0; nn←n;
B]          L1:      if count = m then return(nn);
```

[4]Kludge is a term drawn from the vernacular of computer programmers. It refers to a "patch" or "trick" in a program or system that deals with a potential problem, usually in an inelegant or nongeneralized way. Thus kludgery refers to the use of kludges.

```
C]    count←successor(count);
D]    nn←successor(nn);
E]    go(L1);
```

Compare this with the set of productions for the same task in Figure 2-3. The S in Rules 2, 3, and 5 indicates the successor function. After initialization (Rules 1 and 2), the system loops around Rules 4 and 5 producing the successor rules it needs (Rule 5) and then incrementing NN by 1 for M iterations. In this loop, intermediate calculations (the results of successor function computations) are saved via (PROD) in Rule 5, and the final answer is saved by (PROD) in Rule 3. Thus, as shown in Figure 2-4, after computing 4 + 2 the rule set will contain seven additional rules; it is recording its intermediate and final results by writing new productions and in the future will have these answers available in a single step. Note that the set of productions therefore *is* memory (and in fact long-term memory, or LTM, since productions are never lost from the set). The two are not precisely analogous, since the procedural version does simple addition, while the production set both adds and "learns." As noted by Waterman (1974), the production rule version does not assume the existence of a successor function. Instead Rule 5 writes new productions that give the successor for specific integers. Rule 3 builds what amounts to an addition table, writing a new production for each example that the system is given. Placing these new rules at the front of the rule set (i.e., before Rule 1) means that the addition table and successor function table will always be consulted before a computation is attempted, and the answer obtained in one step if possible. Without these extra steps, and with a successor function, the production rule set could be smaller and hence slightly less complex.

Waterman also points out some direct correspondences between the production rules in Figure 2-3 and the statements in the procedure above. For example, Rules 1 and 2 accomplish the initialization of line A, Rule 3 corresponds to line B, and Rule 4 to lines C and D. There is no production equivalent to the "goto" of line E because the production system execution cycle takes care of that implicitly. On the other hand, note that in the procedure there is no question whatsoever that the initialization step nn ← n is the second statement of "add" and that it is to be executed just once, at the beginning of the procedure. In the productions, the same action is predicated on an unintuitive condition of the STM (essentially it says that if the value of N is known, but NN has never been referenced or incremented, then initialize NN to the value that N has at that time). This degree of explicitness is necessary because the production system has no notion that the initialization step has already been performed in the given ordering of statements, so the system must check the conditions each time it goes through a new cycle.

Thus procedural languages are oriented toward the explicit handling of control flow and stress the importance of its influence on the fundamental organization of the program (as, for example, in recent develop-

Production Rules:

Condition (LHS)	*Action (RHS)*
1] (READY) (ORDER X_1)	→ (REP (READY) (COUNT X_1)) (ATTEND)
2] (N X_1) -(NN) -(S NN)	→ (DEP (NN X_1))
3] (COUNT X_1) (M X_1) (NN X_2) (N X_3)	→ (SAY X_2 IS THE ANSWER) (COND (M X_1) (N X_3)) (ACTION (STOP)) (ACTION (SAY X_2 IS THE ANSWER)) (PROD) (STOP)
4] (COUNT) (NN)	→ (REP (COUNT) (S COUNT)) (REP (NN) (S NN))
5] (ORDER X_1 X_2)	→ (REP (X_1 X_2) (X_2)) (COND (S X_3 X_1)) (ACTION (REP (S X_3 X_1) (X_3 X_2))) (PROD)

Initial STM:

(READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

Notation:

- The X_1's in the condition are variables in the pattern match; all other symbols are literals. An X_1 appearing *only* in the action is also taken as a literal. Thus if Rule 5 is matched with $X_1 = 4$ and $X_2 = 5$, as its second action it would deposit (COND (S X_3 4)) in STM. These variables are local to each rule; that is, their previous bindings are disregarded.
- All elements of the LHS must be matched for a match to succeed.
- A hyphen indicates the ANDNOT operation.
- An expression enclosed in parentheses and starting with a literal [e.g., (COUNT) in Rule 4] will match any expression in STM that starts with the same literal [e.g., (COUNT 2)]. The expression (ORDER X_1 X_2) will match (ORDER 0 1 2 3 . . . 9) and bind $X_1 = 0$ and $X_2 = 1$.
- REP stands for REPlace, so that, for example, the RHS of Rule 1 will replace the expression (READY) in the data base with the expression (COUNT X_1) [where the variable X_1 stands for the element matched by the X_1 in (ORDER X_1)].
- DEP stands for DEPosit symbols at front of STM.
- ATTEND means wait for input from computer terminal. For this example, typing (M 4)(N 2) will have the system add 4 and 2.
- SAY means output to terminal.

FIGURE 2-3 A production system for the addition of two single-digit integers [after Waterman (1974), simplified slightly].

- (COND . . .) is shorthand for (DEP (COND . . .)).
- (ACTION . . .) is shorthand for (DEP (ACTION . . .)).
- PROD means gather all items in the STM of the form (COND . . .) and put them together into an LHS, gather all items of the form (ACTION . . .) and put them together into an RHS, and remove all these expressions from the STM. Form a production from the resulting LHS and RHS, and add it to the front of the set of productions (i.e., before Rule 1).

FIGURE 2-3 continued

ments in structured programming). PS's, on the other hand, emphasize the statement of independent chunks of knowledge from a domain and make control flow a secondary issue. Given the limited form of communication available in PS's, it is more difficult to express concepts that require structures larger than a single rule. Thus, where the emphasis is on global behavior of a system rather than on the expression of small chunks of knowledge, PS's are, in general, less transparent than equivalent procedural routines.

2.4.2 Constrained Format

While there are wide variations in the format permitted by various PS's, in any given system the syntax is traditionally quite restrictive and generally follows the conventions accepted for PS's.[5] Most commonly this means, first, that the side of the rule to be matched should be a simple predicate built out of a Boolean combination of computationally primitive operations; these involve (as noted above) only matching and detection. Second, it means the side of the rule to be executed should perform conceptually simple operations on the data base. In many of the systems oriented toward psychological modeling, the side to be matched consists of a set of literals or simple patterns, with the understanding that the set is to be taken as a conjunction, so that the predicate is an implicit one regarding the success or failure of matching all of the elements. Similarly, the side to be executed performs a simple symbol replacement or rearrangement.

Whatever the format, though, the conventions noted lead to clear restrictions for a pure production system. First, as a predicate, the matching side of a rule should return only some indication of the success or failure of the match.[6] Second, as a simple expression, the matching operation is

[5]Note, however, that the tradition arises out of a commonly followed convention rather than any essential characteristic of a PS.

[6]While binding individual variables or segments in the process of pattern matching is quite often used, it would be considered inappropriate to have the matching process produce a complex data structure intended for processing by another part of the system.

RULE	STATUS	STM AFTER RULE SUCCEEDS	NEW RULES/COMMENTS
CYCLE #1			
		(READY)(ORDER 0 1 2 3 4 5 6 7 8 9)	initial state
Rule 1	Succeeds	(COUNT 0)(ORDER 0 1 2 3 4 5 6 7 8 9) (N 2)(M 4)(COUNT 0) (ORDER 0 1 2 3 4 5 6 7 8 9)	awaits input (M 4)(N 2) after input
Rule 2	Succeeds	(NN 2)(N 2)(M 4)(COUNT 0) (ORDER 0 1 2 3 4 5 6 7 8 9)	X_1 bound to 2
Rule 3	Fails		
Rule 4	Succeeds	(S NN 2)(N 2)(M 4)(S COUNT 0) (ORDER 0 1 2 3 4 5 6 7 8 9)	
Rule 5	Succeeds	(S NN 2)(N 2)(M 4)(S COUNT 0) (ORDER 1 2 3 4 5 6 7 8 9)	X_1 bound to 0 New Rule 6: (S X_3 0) → (REP(S X_3 0)(X_3 1)
CYCLE #2			
Rule 6	Succeeds	(S NN 2)(N 2)(M 4)(COUNT 1) (ORDER 1 2 3 4 5 6 7 8 9)	X_3 bound to the literal COUNT
Rule 1	Fails		
Rule 2	Fails		
Rule 3	Fails		
Rule 4	Fails		
Rule 5	Succeeds	(S NN 2)(N 2)(M 4)(COUNT 1) (ORDER 2 3 4 5 6 7 8 9)	New Rule 7: (S X_3 1) → (REP(S X_3 1)(X_3 2))
CYCLE #3			
Rule 7	Fails		
Rule 6	Fails		
Rule 1	Fails		
Rule 2	Fails		
Rule 3	Fails		
Rule 4	Fails		
Rule 5	Succeeds	(S NN 2)(N 2)(M 4)(COUNT 1) (ORDER 3 4 5 6 7 8 9)	New Rule 8: (S X_3 2) → (REP(S X_3 2)(X_3 3))
CYCLE #4			
Rule 8	Succeeds	(NN 3)(N 2)(M 4)(COUNT 1) (ORDER 3 4 5 6 7 8 9)	X_3 bound to NN
Rule 7	Fails		
Rule 6	Fails		
Rule 1	Fails		
Rule 2	Fails		
Rule 3	Fails		
Rule 4	Succeeds	(S NN 3)(N 2)(M 4)(S COUNT 1) (ORDER 3 4 5 6 7 8 9)	
Rule 5	Succeeds	(S NN 3)(N 2)(M 4)(S COUNT 1) (ORDER 4 5 6 7 8 9)	New Rule 9: (S X_3 3) → (REP(S X_3 3)(X_3 4))
CYCLE #5			
Rule 9	Succeeds	(NN 4)(N 2)(M 4)(S COUNT 1) (ORDER 4 5 6 7 8 9)	
	.		
	.		
	etc.	<continued cycling>	Rules 10 and 11 generated
	.		
	.		
Rule 3	Succeeds	(NN 6)(N 2)(M 4)(COUNT 4) (ORDER 6 7 8 9)	Bind X_1 to 4, X_2 to 6, X_3 to 2; Prints '6 IS THE ANSWER'; Rule 12 produced; Terminates.

FIGURE 2-4 Trace of production system shown in Figure 2-3. Adding 4 and 2.

precluded from using more complex control structures like iteration or recursion within the expression itself (although such operations can be constructed from multiple rules). Finally, as a matching and detection operation, it must only "observe" the state of the data base and not change it in the operation of testing it.

We can characterize a continuum of possibilities for the side of the rule to be executed. There might be a single primitive action, a simple collection of independent actions, a carefully ordered sequence of actions, or even more complex control structures. We suggest that there are two related forms of simplicity that are important here. First, each action to be performed should be one that is a conceptual primitive for the domain. In the DENDRAL system, for example, it is appropriate to use chemical bond breaking as the primitive, rather than to describe the process at some lower level. Second, the complexity of control flow for the execution of these primitives should be limited—in a pure production system, for example, we might be wary of a complex set of actions that is, in effect, a small program of its own. Again, it should be noted that the system designer may of course follow or disregard these restrictions.

These constraints on form make the dissection and "understanding" of productions by other parts of the program a more straightforward task, strongly enhancing the possibility of having the program itself read and/or modify (rewrite) its own productions. For example, the MYCIN system makes strong use of the concept of allowing one part of the system to read the rules being executed by another part. The system does a partial evaluation of rule premises. Since a premise is a Boolean combination of predicate functions such as

```
($AND (SAME CNTXT SITE)                    (the site of the culture is blood and
      (SAME CNTXT GRAM GRAMPOS)             the gramstain is grampositive and
      (DEF IS CNTXT AIR AEROBIC))           the aerobicity is definitely aerobic)
```

and since clauses that are unknown cause subproblems that may involve long computations to be set up, it makes sense to check to see if, based on what is currently known, the entire premise is sure to fail (e.g., if any clause of a conjunction is known to be false). We cannot simply EVAL each clause, since this will trigger a search if the value is still unknown. But if the clause can be "unpacked" into its proper constituents, it is possible to determine whether or not the value is known as yet, and if so, what it is. This is done via a *template* associated with each predicate function. For example, the template for SAME is

```
(SAME CNTXT PARM VALUE)
```

and it gives the generic type and order of arguments for the function (much like a simplified procedure declaration). By using this as a guide to unpack and extract the needed items, we can safely do a partial evaluation of the rule premise. A similar technique is used to separate the known and

unknown clauses of a rule for the user's benefit when the system is explaining itself (see Chapter 18 for several examples).

Note that part of the system is reading the code being executed by the other part. Furthermore, note that this reading is guided by information carried in the rule components themselves. This latter characteristic assures that the capability is unaffected by the addition of new rules or predicate functions to the system.

This kind of technique limits expressibility, however, since the limited syntax may not be sufficiently powerful to make expressing each piece of knowledge an easy task. This in turn both restricts extensibility (adding something is difficult if it is hard to express it) and makes modification of the system's behavior more difficult (e.g., it might not be particularly attractive to implement a desired iteration if doing so requires several rules rather than a line or two of code).

2.4.3 Rules as Primitive Actions

In a pure PS, the smallest unit of behavior is a rule invocation. At its simplest, this involves the matching of literals on the LHS, followed by replacement of those symbols in the data base with the ones found on the RHS. While the variations can be more complex, it is in some sense a violation of the spirit of things to have a sequence of actions in the RHS.

Moran (1973b), for example, acknowledges a deviation from the spirit of production systems in VIS when he groups rules in "procedures" within which the rules are totally ordered for the purpose of conflict resolution. He sees several advantages in this departure. It is "natural" for the user (a builder of psychological models) to write rules as a group working toward a single goal. This grouping restricts the context of the rules. It also helps minimize the problem of implicit context: when rules are ordered, a rule that occurs later in the list may really be applicable only if some of the conditions checked by earlier rules are untrue. This dependency, referred to as implicit context, is often not made explicit in the rule, but may be critical to system performance. The price paid for these advantages is twofold: first, extra rules, less directly attributable to psychological processes, are needed to switch among procedures; second, it violates the basic production system tenet that any rule should (in principle) be able to fire at any time—here only those in the currently active procedure can fire.

To the extent that the pure production system restrictions are met, we can consider rules as the quanta of intelligent behavior in the system. Otherwise, as in the VIS system, we must look at larger aggregations of rules to trace behavior. In doing so, we lose some of the ability to quantify and measure behavior, as is done, for example, with the PSG system simulation of the Sternberg task, where response times are attributed to individual production rules and then compared against actual psychological data.

A different sort of deviation is found in the DENDRAL system, and in a few MYCIN rules. In both, the RHS is effectively a small program, carrying out complex sequences of actions. In this case, the quanta of behavior are the individual actions of these programs, and understanding the system thus requires familiarity with them. By embodying these bits of behavior in a stylized format, we make it possible for the system to "read" them to its users (achieved in MYCIN as described above) and hence provide some explanation of its behavior, at least at this level. This prohibition against complex behaviors within a rule, however, may force us to implement what are (conceptually) simple control structures by using the combined effects of several rules. This of course may make overall behavior of the system much more opaque (see Section 2.4.5).

2.4.4 Modularity

We can regard the *modularity* of a program as the degree of separation of its functional units into isolatable pieces. A program is *highly modular* if any functional unit can be changed (added, deleted, or replaced) with no unanticipated change to other functional units. Thus program modularity is inversely related to the strength of coupling between its functional units.

The modularity of programs written as pure production systems arises from the important fact that the next rule to be invoked is determined solely by the contents of the data base, and no rule is ever called directly. Thus the addition (or deletion) of a rule does not require the modification of any other rule to provide for or delete a call to it. We might demonstrate this by repeatedly removing rules from a PS: many systems will continue to display some sort of "reasonable" behavior.[7] By contrast, adding a procedure to an ALGOL-like program requires modification of other parts of the code to insure that the procedure is invoked, while removing an arbitrary procedure from such a program will generally cripple it.

Note that the issue here is more than simply the "undefined function" error message, which would result from a missing procedure. The problem would persist even if the compiler or interpreter were altered to treat undefined functions as no-ops. The issue is a much more fundamental one concerning organization of knowledge: programs written in procedure-oriented languages stress the kind of explicit passing of control from one section of code to another that is characterized by the calling of procedures.

---

[7]The number of rules that could be removed without performance degradation (short of redundancies) is an interesting characteristic that would appear to be correlated with which of the two common approaches to PS's is taken. The psychological modeling systems would apparently degenerate fastest, since they are designed to be minimally competent sets of rules. Knowledge-based expert systems, on the other hand, tend to embody numerous independent subproblems in rules and often contain overlapping or even purposefully redundant representations of knowledge. Hence, while losing their competence on selected problems, it appears they would often function reasonably well, even with several rules removed.

This is typically done at a selected time and in a particular context, both carefully chosen by the programmer. If a no-op is substituted for a missing procedure, the context upon returning will not be what the programmer expected, and subsequent procedure calls will be executed in increasingly incorrect environments. Similarly, procedures that have been added must be called from *somewhere* in the program, and the location of the call must be chosen carefully if the effect is to be meaningful.

Production systems, on the other hand, especially in their pure form, emphasize the decoupling of control flow from the writing of rules. Each rule is designed to be, ideally, an independent chunk of knowledge with its own statement of relevance (either the conditions of the LHS, as in a data-driven system, or the action of the RHS, as in a goal-directed system). Thus, while the ALGOL programmer carefully chooses the order of procedure calls to create a selected sequence of environments, in a production system it is the environment that chooses the next rule for execution. And since a rule can only be chosen if its criteria of relevance have been met, the choice will continue to be a plausible one, and system behavior will remain "reasonable," even as rules are successively deleted.

This inherent modularity of pure production systems eases the task of programming in them. Given some primitive action that the system fails to perform, it becomes a matter of writing a rule whose LHS matches the relevant indicators in the data base, and whose RHS performs the action. Whereas the task is then complete for a pure PS, systems that vary from this design have the additional task of assuring proper invocation of the rule (not unlike assuring the proper call of a new procedure). The difficulty of this varies from trivial in the case of systems with goal-oriented behavior (like MYCIN) to substantial in systems that use more complex LHS scans and conflict resolution strategies.

For systems using the goal-oriented approach, rule order is usually unimportant. Insertion of a new rule is thus simple and can often be totally automated. This is, of course, a distinct advantage where the rule set is large and the problems of system complexity are significant. For others (like PSG and PAS II) rule order can be critical to performance and hence requires careful attention. This can, however, be viewed as an advantage, and indeed, Newell (1973) tests different theories of behavior by the simple expedient of changing the order of rules. The family of Sternberg task simulators includes a number of production systems that differ only by the interchange of two rules, yet display very different behavior. Waterman's system (Waterman, 1974) accomplishes "adaptation" by the simple heuristic of placing a new rule immediately before a rule that causes an error.[8]

[8]One specific example of the importance of rule order can be seen in our earlier example of addition (Figure 2-3). Here Rule 5 assumes that an ordering of the digits exists in STM in the form (ORDER 0 1 2 . . .) and from this can be created the successor function for each digit. If Rule 5 were placed before Rule 1, the system wouldn't add at all. In addition, acquiring the notion of successor in subsequent runs depends entirely on the placement of the new successor productions *before* Rule 3, or the effect of this new knowledge would be masked.

2.4.5 Visibility of Behavior Flow

Visibility of behavior flow is the ease with which the overall behavior of a PS can be understood, either by observing the system or by reviewing its rule base. Even for conceptually simple tasks, the stepwise behavior of a PS is often rather opaque. The poor visibility of PS behavior compared to that of the procedural formalism is illustrated by the Waterman integer addition example outlined in Section 2.4.1. The procedural version of the iterative loop there is reasonably clear (lines B, C, and E), and an ALGOL-type

```
FOR I := 1 UNTIL N DO . . .
```

would be completely obvious. Yet the PS formalism for the same thing requires nonintuitive productions (like 1 and 2) and symbols like NN whose only purpose is to "mask" the condition portion of a rule so it will not be invoked later [such symbols are termed *control elements* (Anderson, 1976)].

The requirement for control elements, and much of the opacity of PS behavior, is a direct result of two factors noted above: the unity of control and data store, and the reevaluation of the data base at every cycle. Any attempt to "read" a PS requires keeping in mind the entire contents of the data base and scanning the entire rule set at every cycle. Control is much more explicit and localized in procedural languages, so that reading ALGOL code is a far easier task.[9]

The perspective on knowledge representation implied by PS's also contributes to this opacity. As suggested above, PS's are appropriate when it is possible to specify the content of required knowledge without also specifying the way in which it is to be used. Thus, reading a PS does not generally make clear how it works so much as what it may know, and the behavior is consequently obscured. The situation is often reversed in procedural languages: program behavior may be reasonably clear, but the domain knowledge used is often opaquely embedded in the procedures. The two methodologies thus emphasize different aspects of knowledge and program organization.

2.4.6 Machine Readability

Several interesting capabilities arise from making it possible for the system to examine its own rules. As one example, it becomes possible to implement automatic consistency checking. This can proceed at several levels. In the simplest approach we can search for straightforward syntactic problems such as contradiction (e.g., two rules of the form A & B → C and A & B → -C) or subsumption (e.g., two rules of the form D & E & F → G and D

[9]One of the motivations for the interest in structured programming is the attempt to emphasize still further the degree of explicitness and localization of control.

& F → G). A more sophisticated approach, which would require extensive domain-specific knowledge, might be able to detect "semantic" problems, such as, for example, a rule of the form A & B → C when it is known from the meanings of A and B that A → B. Many other (domain-specific) tests may also be possible. The point is that by automating the process, extensive (perhaps exhaustive) checks of newly added productions are possible (and could perhaps be run in background mode when the system is otherwise idle).

A second sort of capability (described in the example in Section 2.4.2) is exemplified by the MYCIN system's approach to examining its rules. This is used in several ways (Davis, 1976) and produces both a more efficient control structure and precise explanations of system behavior.

2.4.7 Explanation of Primitive Actions

Production system rules are intended to be modular chunks of knowledge and to represent primitive actions. Thus explaining primitive acts should be as simple as stating the corresponding rule—all necessary contextual information should be included in the rule itself. Achieving such clear explanations, however, strongly depends on the extent to which the assumptions of modularity and explicit context are met. In the case where stating a rule does provide a clear explanation, the task of modification of program behavior becomes easier.

As an example, the MYCIN system often successfully uses rules to explain its behavior. This form of explanation fails, however, when considerations of system performance or human engineering lead to rules whose context is obscure. One class of rule, for example, says, in effect, "If A seems to be true, and B seems to be true, then that's (more) evidence in favor of A."[10] It is phrased this way rather than simply "If B seems true, that's evidence in favor of A," because B is a very rare condition, and it appears counterintuitive to ask about it unless A is suspected to begin with. The first clause of the rule is thus acting as a strategic filter, to insure that the rule is not even tried unless it has a reasonable chance of succeeding. System performance has been improved (especially as regards human engineering considerations), at the cost of a somewhat more opaque rule.

2.4.8 Modifiability, Consistency, and Rule Selection Mechanism

As noted above, the tightly constrained format of rules makes it possible for the system to examine its own rule base, with the possibility of modifying it in response to requests from the user or to ensure consistency with

[10]These are known as *self-referencing rules*; see Chapter 5.

respect to newly added rules. While all these are conceivable in a system using a standard procedural approach, the heavily stylized format of rules, and the typically simple control structure of the interpreters, makes them all realizable prospects in a PS.

Finally, the relative complexity of the rule selection mechanism will have varying effects on the ability to automate consistency checks, or behavior modification and extension. An RHS scan with backward chaining (i.e., a goal-directed system; see Section 2.5.3) seems to be the easiest to follow since it mimics part of human reasoning behavior, while an LHS scan with a complex conflict resolution strategy makes the system generally more difficult to understand. As a result, predicting and controlling the effects of changes in, or additions to, the rule base are directly influenced in either direction by the choice of rule selection mechanism.

2.4.9 Programmability

The answer to "How easy is it to program in this formalism?" is "It's reasonably difficult." The experience has been summarized (Moran, 1973a):

> Any structure which is added to the system diminishes the explicitness of rule conditions. . . . Thus rules acquire implicit conditions. This makes them (superficially) more concise, but at the price of clarity and precision. . . . Another questionable device in most present production systems (including mine) is the use of tags, markers, and other cute conventions for communicating between rules. Again, this makes for conciseness, but it obscures the meaning of what is intended. The consequence of this in my program is that it is very delicate: one little slip with a tag and it goes off the track. Also, it is very difficult to alter the program; it takes a lot of time to readjust the signals.

One source of the difficulties in programming production systems is the necessity of programming "by side effect." Another is the difficulty of using the PS methodology on a problem that cannot be broken down into the solution of independent subproblems or into the synthesis of a behavior that is neatly decomposable.

Several techniques have been investigated to deal with this difficulty. One of them is the use of tags and markers (control elements), referred to above. We have come to believe that the manner in which they are used, particularly in psychological modeling systems, can be an indication of how successfully the problem has been put into PS terms. To demonstrate this, consider two very different (and somewhat idealized) approaches to writing a PS. In the first, the programmer writes each rule independently of all the others, simply attempting to capture in each some chunk of required knowledge. The creation of each rule is thus a separate task. Only when all of them have been written are they assembled, the data base initialized,

and the behavior produced by the entire set of rules noted. As a second approach, the programmer starts out with a specific behavior that he or she wants to recreate. The entire rule set is written as a group with this in mind, and, where necessary, one rule might deposit a symbol like A00124 in STM solely to trigger a second specific rule on the next cycle.

In the first case the control elements would correspond to recognizable states of the system. As such, they function as indicators of those states and serve to trigger what is generally a large class of potentially applicable rules.[11] In the second case there is no such correspondence, and often only a single rule recognizes a given control element. The idea here is to insure the execution of a specific sequence of rules, often because a desired effect could not be accomplished in a single rule invocation. Such idiosyncratic use of control elements is formally equivalent to allowing one rule to call a second, specific rule and hence is very much out of character for a PS. To the extent that such use takes place, it appears to us to be suggestive of a failure of the methodology—perhaps because a PS was ill-suited to the task to begin with or because the particular decomposition used for the task was not well chosen.[12] Since one fundamental assumption of the PS methodology as a psychological modeling tool is that states of the system correspond to what are at least plausible (if not immediately recognizable) individual "states of mind," the relative abundance of the two uses of control elements mentioned above can conceivably be taken as an indication of how successfully the methodology has been applied.

A second approach to dealing with the difficulty of programming in PS's is the use of increasingly complex forms within a single rule. Where a pure PS might have a single action in its RHS, several psychological modeling systems (PAS II, VIS) have explored the use of more complex sequences of actions, including the use of conditional exits from the sequence.

Finally, one effort (Rychener, 1975) has investigated the use of PS's that are unconstrained by prior restrictions on rule format, use of tags, etc. The aim here is to employ the methodology as a formalism for explicating knowledge sources, understanding control structures, and examining the effectiveness of PS's for attacking the large problems typical of artificial intelligence. The productions in this system often turn out to have a relatively simple format, but complex control structures are built via carefully orchestrated interaction of rules. This is done with several techniques, including explicit reliance on both control elements and certain characteristics of the data base architecture. For example, iterative loops

---

[11]This basic technique of "broadcasting" information and allowing individual segments of the system to determine their relevance has been extended and generalized in systems like HEARSAYII (Lesser et al., 1974) and BEINGS (Lenat, 1975).

[12]The possibility remains, of course, that a "natural" interpretation of a control element will be forthcoming as the model develops, and additional rules that refer to it will be added. In that case the ease of adding the new rules arises out of the fact that the technique of allowing one rule to call another was not used.

are manufactured via explicit use of control elements, and data are (redundantly) reasserted in order to make use of the "recency" ordering on rules (the rule that mentions the most recently asserted data item is chosen first; see Section 2.5.3). These techniques have supported the reincarnation as PS's of a number of sizable AI programs [e.g., STUDENT (Bobrow, 1968)], but, Bobrow notes, "control tends to be rather inflexible, failing to take advantage of the openness that seems to be inherent in PS's."

This reflects something of a new perspective on the use of PS's. Previous efforts have used them as tools for analyzing both the core of knowledge essential to a given task and the manner in which such knowledge is used. Such efforts relied in part on the austerity of the available control structure to keep all of the knowledge explicit. The expectation is that each production will embody a single chunk of knowledge. Even in the work of Newell (1973), which used PS's as a medium for expressing different theories in the Sternberg task, an important emphasis is placed on productions as a model of the detailed control structure of humans. In fact, every aspect of the system is assumed to have a psychological correlate.

The work reported by Rychener (1975), however, after explicitly detailing the chunks of knowledge required in the word problem domain of STUDENT, notes a many-to-many mapping between its knowledge chunks and productions. That work also focuses on complex control regimes that can be built using PS's. While still concerned with knowledge extraction and explication, it views PS's more as an abstract programming language and uses them as a vehicle for exploring control structures. While this approach does offer an interesting perspective on such issues, it should also be noted that as productions and their interactions grow more complex, many of the advantages associated with traditional PS architecture may be lost (for example, the loss of openness noted above). The benefits to be gained are roughly analogous to those of using a higher-level programming language: while the finer grain of the process being examined may become less obvious, the power of the language permits large-scale tasks to be undertaken and makes it easier to examine phenomena like the interaction of entire categories of knowledge.

The use of PS's has thus grown to encompass several different forms, many of which are far more complex than the pure PS model described initially.

2.5 Taxonomy of Production Systems

In this section we suggest four dimensions along which to characterize PS's: form, content, control cycle architecture, and system extensibility. For each dimension we examine related issues and indicate the range as evidenced by systems currently (or recently) in operation.

2.5.1 Form—How Primitive or Complex Should the Syntax of Each Side Be?

There is a wide variation in the syntax used by PS's and corresponding differences in both the matching and detection process and the subsequent action caused by rule invocation. For matching, in the simplest case only literals are allowed, and it is a conceptually trivial process (although the rule and data base may be so large that efficiency becomes a consideration). Successively more complex approaches allow free variables [Waterman's poker player (Waterman, 1970)], syntactic classes (as in some parsing systems), and increasingly sophisticated capabilities of variable and segment binding and of pattern specification (PAS II, VIS, LISP70).[13]

The content of the data base also influences the question of form. One interesting example is Anderson's ACT system (Anderson, 1976), whose rules have node networks in their LHS's. The appearance of an additional piece of network as input results in a "spread of activation" occurring in parallel through the LHS of each production. The rule that is chosen is the one whose LHS most closely matches the input and that has the largest subpiece of network already in its working memory.

As another example, the DENDRAL system uses a literal pattern match, but its patterns are graphs representing chemical classes. Each class is defined by a basic chemical structure, referred to as a *skeleton*. As in the data base, atoms composing the skeleton are given unique numbers, and chemical bonds are described by the numbers of the atoms they join (e.g., "5 6"). The LHS of a rule is the name of one of these skeletons, and a side effect of a successful match is the recording of the structural correspondence between atoms in the skeleton and those in the molecule. The action parts of these rules describe a sequence of actions to perform: *break* one or more bonds, saving a molecular fragment, and *transfer* one or more hydrogen atoms from one fragment to another. An example of a simple rule is

```
ESTROGEN → (BREAK (14 15) (13 17))
           (HTRANS +1 +2)
```

The LHS here is the name of the graph structure that describes the estrogen class of molecules, while the RHS indicates the likely locations for bond breakages and hydrogen transfers when such molecules are subjected to mass spectral bombardment. Note that while both sides of the rule are relatively complex, they are written in terms that are conceptual primitives in the domain.

A related issue is illustrated by the rules used by MYCIN, where the LHS consists of a Boolean combination of standardized predicate functions. Here the testing of a rule for relevance consists of having the stan-

[13]For an especially thorough discussion of pattern-matching methods in production systems as used in VIS, see Moran (1973a, pp. 42–45).

dard LISP evaluator assess the LHS, and all matching and detection are controlled by the functions themselves. While using functions in LHS's provides power that is missing from using a simple pattern match, that creates the temptation to write one function to do what should be expressed by several rules. For example, one small task in MYCIN is to deduce that certain organisms are present, even though they have not been recovered from any culture. This is a conceptually complex, multistep operation, which is currently (1975) handled by invocation of a single function. If one succumbs often to the temptation to write one function rather than several rules, the result can be a system that may perform the initial task but that loses a great many of the other advantages of the PS approach. The problem is that the knowledge embodied in these functions is unavailable to anything else in the system. Whereas rules can be accessed and their knowledge examined (because of their constrained format), chunks of ALGOL-like code are not nearly as informative. The availability of a standardized, well-structured set of operational primitives can help to avoid the temptation to create new functions unnecessarily.

2.5.2 Content—Which Conceptual Levels of Knowledge Belong in Rules?

The question here is how large a reasoning step should be embodied in a single rule, and there seem to be two distinct approaches. Systems designed for psychological modeling (PAS II, PSG, etc.) try to measure and compare tasks and determine required knowledge and skills. As a result, they try to dissect cognition into its most primitive terms. While there is, of course, a range of possibilities, from the simple literal replacement found in PSG to the more sophisticated abilities of PAS II to construct new productions, rules in these systems tend to embody only the most basic conceptual steps. Grouped at the other end of this spectrum are the task-oriented systems, such as DENDRAL and MYCIN, which are designed to be competent at selected real-world problems. Here the conceptual primitives are at a much higher level, encompassing in a single rule a piece of reasoning that may be based both on experience and on a highly complex model of the domain. For example, the statement "a gram-negative rod in the blood is likely to be an *E. coli*" is based in part on knowledge of physiological systems and in part on clinical experience. Often the reasoning step is sufficiently large that the rule becomes a significant statement of a fact or principle in the domain, and, especially where reasoning is not yet highly formalized, a comprehensive collection of such rules may represent a substantial portion of the knowledge in the field.

An interesting, related point of methodology is the question of what kinds of knowledge ought to go into rules. Rules expressing knowledge about the domain are the necessary initial step, but interest has been generated lately in the question of embodying strategies in rules. We have

been actively pursuing this in the implementation of *meta-rules* in the MYCIN system (Davis et al., 1977). These are "rules about rules," and they contain strategies and heuristics. Thus, while the ordinary rules contain standard object-level knowledge about the medical domain, meta-rules contain information about rules and embody strategies for selecting potentially useful paths of reasoning. For example, a meta-rule might suggest:

> If the patient has had a bowel tumor, then in concluding about organism identity, rules that mention the gastrointestinal tract are more likely to be useful.

There is clearly no reason to stop at one level, however—third-order rules could be used to select from or order the meta-rules, by using information about how to select a strategy (and hence represent a search through "strategy space"); fourth-order rules would suggest how to select criteria for choosing a strategy; etc.

This approach appears to be promising for several reasons. First, the expression of any new level of knowledge in the system can mean an increase in competence. This sort of strategy information, moreover, may translate rather directly into increased speed (since fewer rules need be tried) or no degradation in speed even with large increases in the number of rules. Second, since meta-rules refer to rule content rather than rule names, they automatically take care of new object-level rules that may be added to the system. Third, the possibility of expressing this information in a format that is essentially the same as the standard one means a uniform expression of many levels of knowledge. This uniformity in turn means that the advantages that arise out of the embodiment of any knowledge in a production rule (accessibility and the possibility of automated explanation, modification, and acquisition of rules) should be available for the higher-order rules as well.

2.5.3 Control Cycle Architecture

The basic control cycle can be broken down into two phases called *recognition* and *action*. The recognition phase involves selecting a single rule for execution and can be further subdivided into *selection* and *conflict resolution*.[14] In the selection process, one or more potentially applicable rules are chosen from the set and passed to the conflict resolution algorithm, which chooses one of them. There are several approaches to selection, which can be categorized by their rule scan method. Most systems (e.g., PSG, PAS II) use some variation of an LHS scan, in which each LHS is evaluated in turn. Many stop scanning at the first successful evaluation (e.g., PSG), and

[14]The range of conflict resolution algorithms in this section was suggested in a talk by Don Waterman.

hence conflict resolution becomes a trivial step (although the question then remains of where to start the scan on the next cycle: to start over at the first rule or to continue from the current rule).

Some systems, however, collect all rules whose LHS's evaluate successfully. Conflict resolution then requires some criterion for choosing a single rule from this set (called the conflict set). Several have been suggested, including:

(i) Rule order—there is a complete ordering of all rules in the system, and the rule in the conflict set with the highest priority is chosen.
(ii) Data order—elements of the data base are ordered, and that rule is chosen which matches element(s) in the data base with highest priority.
(iii) Generality order—the most specific rule is chosen.
(iv) Rule precedence—a precedence network (perhaps containing cycles) determines the hierarchy.
(v) Recency order—either the most recently executed rule or the rule containing the most recently updated element of the data base is chosen.

For example, the LISP70 interpreter uses (iii), while DENDRAL uses (iv).

A different approach to the selection process is used in the MYCIN system. The approach is goal-oriented and uses an RHS scan. The process is quite similar to the unwinding of consequent theorems in PLANNER (Hewitt, 1972): given a required subgoal, the system retrieves the (unordered) set of rules whose actions conclude something about that subgoal. The evaluation of the first LHS is begun, and if any clause in it refers to a fact not yet in the data base, a generalized version of this fact becomes the new subgoal, and the process recurs. However, because MYCIN is designed to work with judgmental knowledge in a domain where collecting all relevant data and considering all possibilities are very important, in general, it executes *all* rules from the conflict set rather than stopping after the first success.

The meta-rules mentioned above may also be seen as a way of selecting a subset of the conflict set for execution. There are several advantages to this. First, the conflict resolution algorithm is stated explicitly in the meta-rules (rather than implicitly in the system's interpreter) and in the same representation as the rest of the rule-based knowledge. Second, since there can be a set of meta-rules for each subgoal type, MYCIN can specify distinct, and hence potentially more customized, conflict resolution strategies for each individual subgoal. Since the backward chaining of rules may also be viewed as a depth-first search of an AND/OR goal tree,[15] we may view

[15]An AND/OR goal tree is a reasoning network in which AND's (conjunctions of LHS conditionals) and OR's (disjunctions of multiple rules that all allow the same goal/conclusion to be reached) alternate. This structure is described in detail during the discussion of MYCIN's control structure in Chapter 5.

the search tree as storing at every branch point a collection of specific heuristics about which path to take. In addition, rules in the system are inexact, judgmental statements with a model of "approximate implication" in which the user may specify a measure of how firmly he or she believes that a given LHS implies its RHS (Shortliffe and Buchanan, 1975). This admits the possibility of writing numerous, perhaps conflicting heuristics, whose *combined* judgment forms the conflict resolution algorithm.

Control cycle architecture affects the rest of the production system in several ways. Overall efficiency, for example, can be strongly influenced. The RHS scan in a goal-oriented system insures that only relevant rules are considered in the conflict set. Since this is often a small subset of the total, and one that can be computed once and stored for reference, there is no search necessary at execution time; thus the approach can be quite efficient. In addition, since this approach seems natural to humans, the system's behavior becomes easier to follow.

Among the conflict resolution algorithms mentioned, rule order and recency order require a minimal amount of checking to determine the rule with highest priority. Generality order can be efficiently implemented, and the LISP70 compiler uses it effectively. Data order and rule precedence require a significant amount of bookkeeping and processing, and hence may be slower (PSH, a development along the lines of PSG, attacks precisely this problem).

The relative difficulty of adding a new rule to the system is also determined to a significant degree by the choice of control cycle architecture. Like PLANNER with its consequent theorems, the goal-oriented approach makes it possible to simply "throw the rule in the pot" and still be assured that it will be retrieved properly. The generality-ordering technique also permits a simple, automatic method for placing the new rule, as do the data-ordering and recency strategies. In the latter two cases, however, the primary factor in ordering is external to the rule, and hence, while rules may be added to the rule set easily, it is somewhat harder to predict and control their subsequent selection. For both rule order and rule precedence networks, rule addition may be a substantially more difficult problem that depends primarily on the complexity of the criteria used to determine the hierarchy.

2.5.4 System Extensibility

Learning, viewed as augmentation of the system's rule base, is of concern both to the information-processing psychologists, who view it as an essential aspect of human cognition, and to designers of knowledge-based systems, who acknowledge that building truly expert systems requires an incremental approach to competence. As yet we have no range or even points of

comparison to offer because of the scarcity of examples. Instead, we suggest some standards by which the ease of augmentation may be judged.[16]

Perhaps the most basic question is "How automatic is it?" The ability to learn is clearly an area of competence by itself, and thus we are really asking how much of that competence has been captured in the system, and how much the user has to supply. Some aspects of this competence include:

- If the current system displays evidence of a bug caused by a missing or incorrect rule, how much of the diagnosing of the bug is handled by the system, and how much tracing must be done by the user?
- Once the bug is uncovered, who fixes it? Must the user modify the code by hand? . . . tell the system in some command language what to do? . . . indicate the generic type of the error? Can the user simply point out the offending rule, or can the system locate and fix the bug itself?
- Can the system indicate whether the new rule will in fact fix the bug or if it will have side effects or undesired interactions?
- How much must the user know about rule format conventions when expressing a new (or modified) rule? Must he or she know how to code it explicitly? . . . know precisely the vocabulary to use? . . . know generally how to phrase it? Or can the user indicate in some general way the desired rule and allow the system to make the transformation? Who has to know the semantics of the domain? For example, can the system detect impossible conjunctions (A & B, where A $\rightarrow$ not-B), or trivial disjunctions (A $\vee$ B, where A $\rightarrow$ not-B)? Who knows enough about the system's idiosyncrasies to suggest optimally fast or powerful ways of expressing rules?
- How difficult is it to enter strategies?
- How difficult is it to enter control structure information? Where is the control structure information stored: in aggregations of rules or in higher-order rules? The former makes augmentation or modification a difficult problem; the latter makes it somewhat easier, since the information is explicit and concentrated in one place.
- Can you assure continued consistency of the rule base? Who has to do the checking?

These are questions that will be important and useful to confront in designing any system intended to do knowledge acquisition, especially any built around production rules as underlying knowledge representation.

---

[16]It should be noted that this discussion is oriented primarily toward an interactive, mixed-initiative view of learning, in which the human expert teaches the system and answers questions it may generate. It has also been influenced by our experience in attacking this problem for the MYCIN system (Davis, 1976). Many other models of the process (e.g., teaching by selected examples) are of course possible.

2.6 Conclusions

In artificial intelligence research, production systems were first used to embody primitive chunks of information-processing behavior in simulation programs. Their adaptation to other uses, along with increased experience with them, has focused attention on their possible utility as a general programming mechanism. Production systems permit the representation of knowledge in a highly uniform and modular way. This may pay off handsomely in two areas of investigation: development of programs that can manipulate their own representations and development of a theory of loosely coupled systems, both computational and psychological. Production systems are potentially useful as a flexible modeling tool for many types of systems; current research efforts are sufficiently diverse to discover the extent to which this potential may be realized.

Information-processing psychologists continue to be interested in production systems. PS's can be used to study a wide range of tasks (Newell and Simon, 1972). They constitute a general programming system with the full power of a Turing machine, but use a homogeneous encoding of knowledge. To the extent that the methodology is that of a pure production system, the knowledge embedded is completely explicit and thus aids experimental verification or falsifiability of theories that use PS's as a medium of expression. Productions may correspond to verifiable bits of psychological behavior (Moran, 1973a), reflecting the role of postulated human information-processing structures such as short-term memory. PS's are flexible enough to permit a wide range of variation based on reaction times, adaptation, or other commonly tested psychological variables. Finally, they provide a method for studying learning and adaptive behavior (Waterman, 1974).

For those wishing to build knowledge-based expert systems, the homogeneous encoding of knowledge offers the possibility of automating parts of the task of dealing with the growing complexity of such systems. Knowledge in production rules is both accessible and relatively easy to modify. It can be executed by one part of the system as procedural code and examined by another part as if it were a declarative expression. Despite the difficulties of programming PS's, and their occasionally restrictive syntax, the fundamental methodology suggests a convenient and appropriate framework for the task of structuring and specifying large amounts of knowledge. (See Hayes-Roth et al., 1983, for recent uses of production systems.) It may thus prove to be of great utility in dealing with the problems of complexity encountered in the construction of large knowledge bases.