PRODUCTION SYSTEM CONFLICT RESOLUTION STRATEGIES

J. McDermott and C. Forgy
December, 1976

Abstract: Production systems designed to function and grow in environments that make large numbers of different, sometimes competing, and sometimes unexpected demands require support from their interpreters that is qualitatively different from the support required by systems that can be carefully hand crafted to function in constrained environments. In this paper we explore the role of conflict resolution in providing such support. Using criteria developed in the paper, we evaluate both individual conflict resolution rules and strategies that make use of several rules.

This work was supported in part by the Defense Advanced Research Projects Agency (F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research.

I. INTRODUCTION

The typical Artificial Intelligence system of the sixties labored within a highly constrained environment. The recent development of a number of powerful programming tools has made it feasible to build systems that can function intelligently in more interesting environments. The production system control structure [Davis and King, 1976; Newell, 1973; Rychener, 1976] is one such tool. In this paper we argue that the production system control structure -- provided it makes use of a carefully devised conflict resolution strategy -- is particularly suitable for systems that must respond in reasonable fashion to frequent, sometimes competing, and sometimes unexpected demands from their environments.

A production system consists of a collection of productions held in production memory and a collection of assertions held in working memory. A production, P, is a conditional statement composed of zero or more condition elements, C's, and zero or more action elements, A's; a production has the form

$$P_j \ (C_1 \ C_2 \ \ldots \ C_n \ \text{--->} \ A_1 \ A_2 \ \ldots \ A_m)$$

Most action elements modify working memory by deleting, adding, or modifying working memory elements. Condition elements are templates; when each can be matched by an element in working memory, the production containing them is said to be instantiated. An instantiation is an ordered pair of a production and the elements from working memory that satisfy the conditions of the production. The production system interpreter operates within a control framework called the recognize-act cycle. In recognition, it finds the instantiations to be executed, and in action, executes them, performing whatever actions occur in their action sides. The recognize-act cycle is repeated until either no production can be instantiated or an action element explicitly stops the processing. Recognition can be further divided into match and conflict resolution. In match, the interpreter finds the conflict set, the set of all instantiations of productions that are satisfied on the current cycle. In conflict resolution, it selects from the conflict set one or more instantiations to be executed.

This paper will explore the role of conflict resolution in production systems designed to function intelligently in dynamic environments. In the next section we propose a set of criteria for determining the adequacy of conflict resolution rules. In section III, several specific rules are described. Then in section IV, these rules are evaluated in terms of the proposed criteria. It will become evident in this section that for systems designed for dynamic environments, no single rule is adequate, and thus that several rules have to be used in conjunction with one another. Finally, in section V, we describe a number of different combinations of rules that do meet the criteria of adequacy.

II. CONFLICT RESOLUTION

If a system is to be capable of functioning intelligently in an environment that makes varied and sometimes unexpected demands, it must meet two requirements. First, it must be responsive to its environment. When the environment makes a demand, the system must be able to attend to that demand, decide what action is necessary, and then, take that action. Secondly, it must be able to learn. When encountering a new aspect of the environment or when shown that a previously learned behavior is inadequate, the system must be ready to acquire a new behavior. Systems that are both responsive to their environment and able to augment and refine their knowledge of that environment would, given sufficient time and instruction, be able to behave appropriately in any situation.

A production system, if it is to meet both of these requirements, must be given substantial support by its interpreter. Because a production system becomes aware of changes in its environment only during the recognize phase of the recognize-act cycle, responsiveness suffers if too much time is spent in the act phase. Thus to meet the responsiveness requirement, the number of productions fired on each cycle must be limited. Perhaps the most obvious way to limit the number fired is to make them applicable to mutually exclusive situations. But to do this requires that the productions be given knowledge of each others domains of applicability, and this severely restricts the system's ability to learn. For in order for a production system to add productions to its production memory without having to modify many of the productions already there, the productions must have a high degree of autonomy. Using conflict resolution to limit the number of productions fired enables the necessary degree of autonomy to be maintained. Since conflict resolution can select on the basis of global considerations unknown to the individual productions, each production is required to say only that if no other production more applicable to the current situation is ready, it is. As other productions are added, as more knowledge becomes available, as the overall goals of the system change, the role of individual productions remains the same; each has to say only that it understands and is ready to respond to some tiny piece of the current situation.

If a production system is to function by performing only a small number of actions per cycle, as we just argued it must in order to be responsive to its environment, it must meet a requirement in addition to the two mentioned above. Since some of the system's behaviors will involve long sequences of actions, it must be able to coordinate the firing of several productions, each of which will perform only a few actions. The most obvious way to effect this coordination is to require that each production explicitly evoke its successor. But if this path is taken, production autonomy is lost. Again, conflict resolution can provide a solution. With conflict resolution to make the final choice of the productions to be fired, a production need specify only what is to be done, rather than who is to do it. A small distinction, but enough. New productions may be added to a system employing this mechanism for control with no knowledge required of the existing system beyond the knowledge of the names of a few goals.

A production system which is responsive to the demands of its environment will be said to display sensitivity. One which is able to maintain continuity in its behavior will be said to display stability. We have argued that the function of conflict resolution is to provide a mechanism that can preserve sensitivity and stability without sacrificing production autonomy. We have not yet shown that conflict resolution can preserve both simultaneously. The following two subsections consider in detail what characteristics an interpreter must have in order for production systems having both stability and sensitivity to be implemented easily. Later sections will be more concrete and will show how particular conflict resolution rules contribute to sensitivity and stability.

Sensitivity

Attempts to build production systems capable of operating in dynamic environments have shown that such systems are significantly easier to construct when the interpreter provides support of certain kinds. Below we list five characteristic kinds of support which have proven useful. Of course, we make no claim of completeness.

1. The interpreter should aid the system in its attempts to remain sensitive to its environment while focusing its attention on a single task.
2. The interpreter should aid the system in its attempts to be sensitive simultaneously to multiple aspects of its own processing.
3. The interpreter should aid the system in its attempts to deal intelligently with the existence in working memory of conflicting data.
4. The interpreter should recognize when multiple instantiations are attending to aspects of the same situation and take some reasonable action.
5. All actions taken by the interpreter should be observably deterministic to the system.

In the following paragraphs we describe these five characteristics more fully.

Although the needs of sensitivity and stability are not conflicting, an implementation strongly biased towards one could be weak in its treatment of the other. The first characteristic above implies only that the interpreter should ignore neither. (Since this characteristic concerns the interaction between sensitivity and stability, it could have been proposed instead in the following subsection.)

The second characteristic, like the first, concerns the interaction between the need for sensitivity in a system and the need for direction in the system's processing. For the second characteristic, however, the sensitivity is sensitivity to the results of its own actions. As the system engages in activity directed toward a particular goal, there is the possibility that, at any time, an important, but unexpected event may transpire as a side effect of its processing. For example, the system might generate evidence (such as a repeated state) that it is looping, or it might, while working on one piece of a problem, transform the problem in some significant way. In such cases, it is

certainly desirable for the system to recognize what has happened and take action accordingly.

A production system, with its single global data base (working memory) and no local memory, is particularly vulnerable to the frame problem. For example, a group of productions might communicate using structures similar in form to structures used earlier by another group, or the environment might force on the system information that conflicts with information already in working memory. An interpreter that possesses the third characteristic can aid production systems in their attempts to deal with the problem of distinguishing relevant information from information that is no longer relevant.

In contrast to the first three characteristics, which are concerned with ways in which the interpreter can aid a production system in its quest for sensitivity, the fourth characteristic is concerned with a sensitivity that the interpreter itself must possess. Since a production system needs knowledge of varying degrees of specificity to function in a complex environment, a given demand will often find more than one production ready to respond. Preventing this situation is quite beyond the power of the production system; attempting to be always ready to recognize the situation and take appropriate action would be highly constraining for the system, even with support from the interpreter. Thus, it must be the responsibility of the interpreter to recognize the situation and take some kind of action.

The fifth characteristic, that the actions of the interpreter be observably deterministic[1], is important if systems using the interpreter are to learn from experience. Experience would, after all, be of small value in a world without causality.

Stability

As pointed out in the previous subsection, attending too closely to the needs of either stability or sensitivity can result in a loss of the other. In particular, if sensitivity is not to be lost, the designer of a production system must walk a narrow path when building stability into his system, carefully dividing the responsibility for stability between the interpreter and the system itself. The designer cannot put all of the responsibility on the interpreter, in essence adding a program counter to the production system, without losing the potential for sensitivity. Neither can he put all of the responsibility on the production system, essentially using the production system to program an interpreter for another language, for this extreme also results in a system that has lost the potential for sensitivity.

The method employed here to arrive at a reasonable division of responsibility was to determine the forms the needed coordination of firings could take, and then to determine the minimum support that must be provided by the interpreter in order to allow these forms to be implemented without the use of executive productions. The forms the coordination can take were determined by an examination of existing

1 Presumably all actions taken by a computer are deterministic, but that determinism is of little value to the system if it depends on state variables the system cannot examine.

production systems. It could be argued that the number of existing production systems is too small to give validity to results obtained in this fashion. However, this worry seems unfounded. The forms of coordination that we found correspond closely to the major control constructs used in conventional languages, and the few deviations are easily explained. These forms will be presented below in terms of the analogous conventional control constructs.

In general, there is only one way to coordinate production firings and that is by modifying the contents of working memory. A data element placed in working memory to enable the firing of some production or group of productions will be called a signal. Signals can have a significance beyond their control function, but only the control function is of interest here. A data element used to effect control will be called a signal regardless of any other uses.

The basic control construct in conventional languages is the GOTO (and sequential execution which is, after all, just a variant of the GOTO). The use of signals to evoke productions and groups of productions is closely analogous to the use of GOTOs -- provided there is assurance that the enabled productions will be the next to fire.

The FOREACH construct provides iteration over a set of data. Since the condition part of a production describes elements of a set, an analogous control construct in production systems is one that allows each of the instantiations of a single production to be executed once, effectively causing the production to loop over the set of data it selects from working memory.

As the FOREACH provides for iteration over a set of data, the FORK-JOIN provides for iteration over a set of processes. At the FORK a number of parallel processes (parallel in the weak sense that there is no specified ordering among them) are initiated; at the JOIN the multiple control paths are merged back into one. In production systems, the FORK enables a number of productions without giving a preferred order of firing. The JOIN is a production that fires upon completion of the processing initiated by the enabled productions.

One control construct often used in production systems has no close analogue among the conventional control constructs. (It has, perhaps, a distant analogue in the ASSIGN-GOTO.) This construct, called EXTERNAL SEQUENCING, allows one production to enable multiple productions and to specify the order in which their instantiations are to be executed.

To implement SUBROUTINEs requires the capabilities of transfering control to the SUBROUTINE, passing data to the SUBROUTINE, and returning control to the calling process. Transfer of control to the SUBROUTINE entails only the execution of a GOTO, discussed above. Return of control to the calling process in conventional languages entails execution of a GOTO to an address passed to the SUBROUTINE. Something similar can be accomplished in production systems by passing to the SUBROUTINE a signal encoded so that it will not take effect immediately and then relying on the SUBROUTINE to decode the signal as its last operation. It is perhaps simpler, though, to use EXTERNAL SEQUENCING to cause the production at the return point to be fired on termination of the SUBROUTINE. In this case the SUBROUTINE need do nothing to

effect the return. Data can be passed to SUBROUTINEs reliably only if it can be guaranteed both that the SUBROUTINE will get all of the data passed to it and that it will be able somehow to distinguish this data from all other elements in working memory.

Since a single control construct used in isolation provides little power, an ability to create hierarchies of control constructs is necessary. If the system is using one construct and some production initiates another type of control, the first must be suspended until the second completes. If for example, a production in a FORK-JOIN initiates a FOREACH, the FOREACH must be allowed to finish before the FORK-JOIN resumes. If strictly hierarchical control is assumed, nothing more than a stack of control signals is needed.

Two control constructs commonly used in conventional languages, the IF and the WHILE-DO, are seldom explicitly constructed in production systems. The IF is already provided, of course, in the form of productions. The WHILE-DO is provided in the recognize-act control paradigm, which on every cycle determines if there is anything to do (i.e., if there is at least one instantiation ready to be executed) and if so, does it (i.e., executes one or more of the instantiations).

III. POSSIBLE CONFLICT RESOLUTION RULES

Conflict resolution rules can be distinguished from one another in a variety of ways. Perhaps the most obvious is to distinguish among them on the basis of the criteria each uses to determine the appropriateness of an instantiation. Each of the rules we will consider makes use of one or more of the following five criteria:

A priority ordering between productions. Instantiations of a production with the higher priority are preferred to instantiations of a production with the lower priority.

A special case relationship between instantiations. There are a variety of ways of specifying when one instantiation is a special case of another. Either the general or the special case may be preferred.

The relative recency of the data of instantiations. The rules in this class differ in their choice of which subset of data to order on, their interpretation of recency, and whether they prefer more or less recent data.

Whether an instantiation is distinct from previously executed instantiations. There are several ways of specifying when two instantiations are distinct. Ordinarily, instantiations that are distinct from previously executed instantiations are given preference.

An arbitrary decision. The rules in this class give preference to some subset of instantiations without making use of any information about the instantiations.

There are many conflict resolution rules in each of the five classes defined above. Within each of these classes, rules can be distinguished from one another on the basis of the sources of knowledge on which they depend. The knowledge sources which are used by the rules in the classes defined above are production memory, working memory, and a memory maintained by the interpreter. Conflict resolution rules can also be distinguished on the basis of their selectivity. Some rules can guarantee that a single instantiation will be selected to be executed. Other rules are much less selective and cannot be used in isolation without introducing the danger that the number of instantiations executed on a given cycle, and hence the number of actions performed, might be arbitrarily large.

In this section we will describe 14 rules. They will be grouped according to the criteria by which they judge the appropriateness of instantiations. Within each group, the rules will be distinguished on the basis of their selectivity as well as on the basis of what knowledge sources they use. The rules that we will present do not cover the space of possible rules.[2] Our aim was simply to select a few representative rules having interesting properties. We have included most of the conflict resolution rules used by existing production systems.

Production Order Rules, POs

Production order rules use a pre-established priority ordering on productions as their criterion of selection. Their source of knowledge, then, is always production memory. We will consider only two such rules. Under the first, which we will call PO1, the relation of dominance totally orders the productions. The second rule, PO2, is a generalization of rule PO1. The relation of dominance under PO2 is given by a directed graph where the vertices in the graph represent productions and the edges represent dominance relations between the vertices joined by the edges. The graph for PO2 is disconnected. Each component of the graph contains productions all of which are applicable to the same task, and hence, there is no relation of dominance between productions related to different tasks.

PO1 is a strongly selective rule. Given a completely ordered set of productions and the set of instantiations of those productions, it prefers instantiations of the production that is highest in the priority ordering. This rule is, of course, of limited usefulness to systems designed to function in multi-task domains in which the productions germane to different tasks cannot be meaningfully ordered. In such domains, a less constrained rule, such as PO2, is more appropriate.

PO2 is much less selective than PO1. Using the pre-established dominance relation on productions to establish a dominance relation on instantiations, PO2 prefers every instantiation not dominated by another. Since the graph for PO2 is disconnected, each component of the graph which contains an enabled production will contribute at least one instantiation to the set of preferred instantiations. Moreover,

---

2 In particular, we do not consider rules that can be appropriately used only by systems that place very weak restrictions on the amount of processing that can be done during the action phase of a cycle. For a discussion of such rules, see Hayes-Roth and Lesser [1977].

since the productions within each component need not be completely ordered, each component may contribute several instantiations.

Unlike the other rules we will describe, production order rules require the production system to specify for each production built where in the priority ordering that new production is to lie. One scheme for adding productions to a system using a production order rule was proposed by Waterman [1974]. His system, which uses PO1, inserts a newly built production just before the first production that has either a condition element or an action element in common with the new production. The rationale for this scheme is that a production being added should mask (or at least partially mask) other productions with a similar function, but should do so in a way that interferes as little as possible with the already established ordering. The system uses as a heuristic the assumption that if two productions have a condition or action element in common, they have a similar function.

Special Case Rules, SCs

Special case rules use the presence of a special case relationship between instantiations as their criterion of selection. They may use production memory, working memory, or both as their source of knowledge. We will present four special case rules, SC1-SC4. SC1, which has production memory as its knowledge source, is sensitive to a special case relationship between the productions of instantiations. SC2, which has working memory as its knowledge source, is sensitive to a special case relationship between the data of instantiations. The two other rules, SC3 and SC4, make use of both of these knowledge sources. Because not all pairs of instantiations have a special case relationship, SC rules are only weakly selective.

SC1 defines the special case relationship in the following way: A production, P_s is a special case of another production, P_g, if (1) P_s has at least as many condition elements as P_g, (2) for each condition element in P_g containing constant elements, there is a corresponding condition element in P_s containing those elements as a subset, and (3) P_s and P_g are not identical. SC1 prefers the instantiations of those productions that do not have a special case.[3]

The definition of special case used by SC2 is very different from the definition used by SC1. SC2 considers instantiation I_s to be a special case of instantiation I_g if I_s contains as a proper subset all of the memory elements contained in I_g. SC2 prefers those instantiations that do not have a special case.

SC3, which uses both production memory and working memory as knowledge sources, defines the special case relationship in much the same way that SC2 does. It simply augments SC2's definition in such a way that negated condition elements are taken into account. For SC3, I_s is a special case of I_g if (1) I_s contains all of the

3 SC1 could have been presented as a production order rule. We chose to present it as a special case rule because (1) it is worthwhile to compare its use of a special case relationship with other rules making use of that relationship, and (2) most systems that make use of production order use criteria for ordering productions that are more complex (or at least less easy to make explicit) than the special case criterion given above.

memory elements contained in I_g as a proper subset, and (2) P_s has more condition elements than P_g. SC3, like SC2, prefers those instantiations that do not have a special case.

SC4 defines the special case relationship in exactly the same way that SC3 defines it. The two rules differ only in the instantiations which they prefer. SC4 prefers those instantiations that do not have a general case.

Figure 1 shows which instantiations in the conflict set ($I1_1$-$I4_3$) would be preferred by each of the special case rules. It also shows the instantiations that would be preferred by the recency rules, which we will consider next. Assume that the memory elements (P S) and (Q T) were asserted on the previous cycle, (P T) and (R V) two cycles ago, (Q S) three cycles ago, (P V) four cycles ago, and (W V) and (W T) 101 cycles ago. Note that the symbol "=" in a production signifies that the symbol immediately following it is to be treated as a variable. If the same variable occurs more than once on the condition side of a production, all occurrences must be bound to the same value in order for the match to succeed. The symbol "¬" preceding a condition element signifies that the match is to fail if there is an assertion in working memory which matches that condition element.

Recency Rules, Rs

Recency rules use the amount of time (e.g., number of cycles) that elements have been in working memory as their selection criterion. In most cases, and for a number of different reasons, the rules favor more recent elements. All recency rules use working memory as their knowledge source. We will consider five recency rules, R1-R5. To make it easier to state the various definitions of recency used by these rules, we will assume that working memory contains (A_1 A_2 A_3 ... B_1 B_2 B_3 ... C_1 ...), where all elements beginning with the same alphabetical character were asserted during a single cycle, A_i after B_i, B_i after C_i, and where for any two elements, X_i and X_{i+1}, X_i was asserted after X_{i+1}.

Rules R1, R2, and R3 are all quite selective. The first of these, R1, considers a memory element, M_j, to be more recent than another, M_k, if M_j was asserted after M_k. For example, A_1 is more recent than A_2. This rule orders instantiations on the basis of the most recent memory element contained in each. If B_1 is the most recent element in I_m, B_2 is the most recent element in both I_n and I_o, and C_1 is the most recent element in I_p, then when this rule is applied, I_m will be preferred to I_n, I_o and I_p; I_n and I_o would be considered equivalent and both would be preferred to I_p. Applying this rule results in a complete ordering on equivalence classes. The instantiations in the highest equivalence class are preferred to all other instantiations.

R2 is similar to R1 in that it also orders instantiations on the basis of the most recent element that each contains. R2, however, uses a different definition of recency. All memory elements asserted during a single cycle are equally recent and are more recent than any element asserted on a previous cycle. Using the example above, when this rule is applied, I_m, I_n and I_o will be selected in preference to I_p.

The third rule, R3, uses still another definition of recency. R3 considers two memory elements to be equally recent if the cycles, CYC_j and CYC_k, on which they

WM ((P S) (Q T) (P T) (R V) (Q S) (P V) ... (W V) (W T))

P1 ((Q =X) (P =X) --> ...)
$I1_1$ [P1 ((Q T) (P T))]
$I1_2$ [P1 ((Q S) (P S))]

P2 ((P S) (P =X) (W =X) --> ...)
$I2_1$ [P2 ((P S) (P T) (W T))]
$I2_2$ [P2 ((P S) (P V) (W V))]

P3 ((=X S) (=X =Y) (W =Y) (R =Y) (Q S) --> ...)
I3 [P3 ((P S) (P V) (W V) (R V) (Q S))]

P4 ((Q S) ¬(U S) (P =X) ¬(U V) ¬(U T) --> ...)
$I4_1$ [P4 ((Q S) (P S))]
$I4_2$ [P4 ((Q S) (P T))]
$I4_3$ [P4 ((Q S) (P V))]

SC1 $\{I2_1\ I2_2\ I3\ I4_1\ I4_2\ I4_3\}$
SC2 $\{I1_1\ I2_1\ I3\ I4_2\}$
SC3 $\{I1_1\ I2_1\ I3\ I4_1\ I4_2\ I4_3\}$
SC4 $\{I1_1\ I1_2\ I2_1\ I2_2\ I4_2\ I4_3\}$

R1 $\{I1_2\ I2_1\ I2_2\ I3\ I4_1\}$
R2 $\{I1_1\ I1_2\ I2_1\ I2_2\ I3\ I4_1\}$
R3 $\{I1_1\ I1_2\ I4_1\ I4_2\}$
R4 $\{I1_1\ I1_2\ I4_1\ I4_2\ I4_3\}$
R5 $\{I2_1\}$

Figure 1: Conflict Resolution Using Special Case and Recency Rules

were asserted have the following relationship: $\lfloor \log_2 CYC_j \rfloor = \lfloor \log_2 CYC_k \rfloor$; otherwise, it considers the element asserted on the later cycle more recent. Thus if A_1 was asserted on the previous cycle and D_1 was asserted three cycles before that, they are not considered to be equally recent. After three more cycles, however, they become equally recent.[4] R3 orders instantiations on the basis of the least recent element that each contains; it prefers the instantiation whose least recent element is most recent. For example, if Z_1 is the least recent element in I_m and C_1 is the least recent element in I_n, then R3 will prefer I_n to I_m. As with R1 and R2, applying this rule always results in a complete ordering on equivalence classes. The instantiations in the highest equivalence class are preferred to all other instantiations.

R4 makes use of a definition of recency which effectively partitions working

4 This somewhat bizarre definition of recency can be justified if it is the case that as memory elements become older, their absolute age becomes less significant.

memory into two sets of elements: recent and non-recent. Recent elements are those that were asserted during any of the previous 100 cycles. Non-recent elements are those that were asserted earlier. R4 is weakly selective since it prefers any instantiation which contains only recent elements.

The fifth recency rule, R5, uses the definition of recency used by R1: a memory element, M_j, is more recent than another, M_k, if M_j was asserted after M_k. Unlike the other recency rules, R5 considers the recency of all data elements of an instantiation. To order two instantiations, it first compares their most recent elements; if those elements are equally recent, it compares their next most recent elements, and so on. When it finds a pair of elements that are not equally recent, it prefers the instantiation containing the more recent element. If it exhausts the data of one instantiation without finding a pair of differing recency, it prefers the instantiation not exhausted. Only if the instantiations are exhausted simultaneously, and, hence, contain exactly the same data, are they considered equivalent under R5. If I_j is $(B_1\ Z_1\ Z_2)$, I_k is $(B_1\ Z_2)$, and I_m is $(B_1\ Z_1)$, then the rule would order the instantiations: I_j, I_m, I_k. R5 has the properties of a special case rule as well as the properties of a recency rule. If an instantiation I_s would be preferred to I_g by SC2, it would also be preferred by R5 either because I_s would contain an element more recent than the corresponding element of I_g, or (if the data of I_g were the most recent data of I_s) because the data of I_g would be exhausted first. This rule, then, is the most strongly selective of the recency rules.

Distinctiveness Rules, Ds

Distinctiveness rules select on the basis of the similarity or dissimilarity of the instantiations in the conflict set to previously executed instantiations. Knowledge of what instantiations have been executed is provided by the interpreter. We will describe two rules, D1 and D2, both of which are weakly selective.

D1 considers two instantiations to be distinct if the productions of those instantiations are different. It prefers instantiations of productions that did not fire on the previous cycle to instantiations of productions that did fire on the previous cycle.

D2 uses a stronger criterion. It considers two instantiations to be distinct if either the productions or the data of two instantiations are different. D2, like D1, gives preference to distinct instantiations. After an instantiation is executed, it becomes a member of the set of instantiations that are never to be preferred.[5]

Arbitrary Decision Rules, ADs

The final rule that we will consider, AD1, stipulates what is to be done in the absence of any information that would indicate that I_j should be preferred to I_k. This rule simply selects one instantiation at random.

---

5 A set of distinctiveness rules which we will not consider in this paper, but which are of some interest, are those that treat two instantiations as non-distinct if they have a special case relationship to one another. These rules can be used to disable all but one of a set of instantiations that bear on the same situation.

IV. EVALUATION OF THE RULES

In this section we evaluate the rules described in the previous section using the criteria developed in section II. We indicate first which rules support which of the sensitivity and stability characteristics. Then we show the degree to which each of the rules considered in isolation provides an adequate basis for conflict resolution.

Rules Supporting Characteristics 1 and 2

The first two characteristics are concerned with the problem of building systems which display both sensitivity and stability. The first characteristic is concerned with a system's sensitivity to its environment, the second with a system's sensitivity to the results of its own actions. The problem can be solved by building systems that are sensitive, not to the state of working memory, but to changes in the state of working memory. Sensitivity of this type is quite sufficient to yield a system able to interrupt its processing and respond to important events. It is, at the same time, the basis of coherence in processing. If the system is strongly sensitive to change, the changes made by the firing of one production will strongly influence the choice of which production to fire on the next cycle. Four of the rules that we have studied promote a sensitivity to change of state.

Three of these rules, R1, R2, and R5, are recency rules. Since they achieve their effect by dynamically ordering instantiations on the basis of the most recent element contained in each, these rules are strongly sensitive to change. If there are instantiations that make use of elements added on the previous cycle, whether by the system itself or by the environment, these instantiations will be preferred by all three rules. The slightly different interpretation of recency used by R2 makes its response to change somewhat more uniform than is the case with the other two. Since R2 considers all elements added during a single cycle to be equally recent, the response of a system using R2 to an element added to working memory by the environment will be the same regardless of whether the element is added before, during, or after the action part of the interpreter's cycle.

D2 also promotes a sensitivity to change, but in a manner different from, though complementary to, that of the recency rules. If D2 is used in combination with R1, R2, or R5, the resulting strategy will display an eminently useful form of sensitivity to change. The recency rule will encourage the system to go forward either with its current task or with some other more urgent task; when progress can no longer be made, the combined strategy will cause the system to go back to the last choice point still open (the recency rule will discourage the system from backing any further than necessary) and take an alternate path.

The other rules are essentially indifferent to these first two characteristics. If any of the other rules are used in conjunction with rules that support characteristics 1 or 2, they do not (or at least do not necessarily) weaken the support. Neither, however, do they do anything to promote these two characteristics.

Rules Supporting Characteristic 3

The third characteristic is concerned with the problem of dealing with conflicting memory elements. Implicit in characteristic 3 is the notion that the interpreter must make some decision about the relative usefulness of the conflicting data. A reasonable heuristic for a system trying to function in a changing environment is to assume that the older information is, the more likely it is to describe a no longer existing state. Hence, a conflict resolution rule which makes the system ignore old data when there is more recent data available supports characteristic 3. Rules R3 and R4 both have this property. A second heuristic is to assume that if information has already been used, then it should make way for other information. By this heuristic, the use of D2, which prefers instantiations that have not been executed to those that have been executed, is justified.

R3 may be preferable to R4. R4, which effectively disables all instantiations containing an element which has been in working memory for more than 100 cycles, in essence stipulates that all data more than 100 cycles old must be considered suspect. R3, with its relative definition of recency, does not tie the probability that an element accurately represents a current state of the environment to the element's absolute age.

Rules Supporting Characteristic 4

The fourth characteristic, like the third, implies the need for a decision by the interpreter. The decision to be made in this case is whether two instantiations are attending to the same situation. Either working memory or production memory can provide knowledge on which to base this decision. If the interpreter uses working memory, it may base the decision on any of a number of possible criteria, from requiring that the data of one instantiation be a subset of the data of another, to requiring only that the data of the instantiations have at least one element in common. However, since much of the data in working memory may have global significance, it is hard to justify any but the first of these criteria. If the interpreter uses production memory as its only knowledge source, it is restricted in the decision criteria it may use. It must assume that for each pair of productions, either the instantiations of one should always be treated as a special case of the instantiations of the other, or a special case relationship will never hold between instantiations of the two productions. Five of the rules described above are (to varying degrees) useful in determining which instantiations attend to the same situation and in prescribing a response.

Rules SC1-SC3 and R5 are consonant with the demands of characteristic 4; rule SC4 actively works against characteristic 4. If a system is to respond intelligently in situations where there is little information about how to satisfy the current demand of the environment as well as in situations where there is a good deal of information, the system must have a range of methods, from weak to strong. A system which chose a weak method in preference to a strong method when both were apparently applicable would not be behaving in a reasonable way. All of these rules except SC4 would, in choosing the instantiation making the strongest informational demand, choose the strong method.

Among the five rules, there are four different definitions of special case. SC1 defines the relation using information from production memory only. SC2 and R5 use working memory only. SC3 and SC4 use both memories. Of the four rules that use working memory as a knowledge source, SC2 is the only one that employs exactly the criterion suggested above, finding one instantiation to be a special case of another if the data of the first is a superset of the data of the second. R5 is somewhat stronger since it will order all instantiations having that relation, plus others that do not. SC3 and SC4 are weaker since they require more than the above relation before considering one instantiation to be a special case of another.

Since SC3 requires more evidence than either SC2 or R5 before it will grant that two instantiations have a special case relationship, it might seem to provide more support for characteristic 4. What this observation misses is that the more evidence a rule demands, the more likely it is that the rule will fail to recognize two instantiations that are attending to aspects of the same situation. Thus the danger that two instantiations might be falsely viewed as bearing on the same situation must be weighed against the danger that two instantiations that do bear on the same situation might not be recognized as such.

Rules Supporting Characteristic 5

All of the rules except AD1, because their effects are observably deterministic, support characteristic 5. Thus, any conflict resolution strategy not employing AD1 will support characteristic 5. Unfortunately, the use of AD1 is often necessary to prevent the loss of sensitivity that results from executing multiple instantiations. None of the other rules presented here is sufficient by itself to produce a preferred set containing only a single instantiation. Certain combinations of rules are sufficient to produce such a set, but most are not. If the conflict resolution strategy used by a system has the property that it will not generate a unique choice, the system will sometimes execute multiple instantiations and, as pointed out in the general discussion of sensitivity, lose some sensitivity because of the lengthened cycle. To force the system always to produce a unique preferred instantiation is to use AD1.

Rules Supporting Stability

Rules R1, R2, and R5 support GOTOs; rule D1 is somewhat in conflict with the needs of GOTOs. A GOTO cannot be implemented in production systems unless it is possible for a production to specify that some subset of the set of instantiations is to be preferred. Rules R1 and R5 make this possible. If the final action performed by a production asserts a signal that enables a group of productions, then the instantiations of these productions will have precedence over all other instantiations. Rule R2, if applied in isolation, also guarantees that instantiations containing a signal asserted on one cycle will be executed on the next cycle. However, instantiations not containing that signal may be executed as well. Thus if R2 is used in combination with other rules, instantiations not containing the signal may be selected in preference to those containing the signal. Rule D1 would work against a production performing a GOTO to itself.

Two forms of the FOREACH are in common use. Each requires different

support. A FOREACH can be implemented either by executing all instantiations on one cycle or by executing each on a separate cycle. Executing all instantiations on one cycle requires only that rule AD1 not be used. Executing the instantiations on separate cycles is made easier by the use of rule D2 and is made more difficult by the use of D1. When D2 is used there is no need to explicitly disable each instantiation after it is executed. Use of rule D1 makes it impossible to have a one production loop.

As with the FOREACH, there are two ways to implement a FORK-JOIN, executing all instantiations on one cycle, or executing each on a separate cycle. Executing all on one cycle again requires only that AD1 not be used. If the instantiations are to be executed on separate cycles, there must be a way to guarantee that the JOIN production is the last to be executed. Use of PO1 or PO2 with the JOIN production having lower priority than the other productions is one possibility. Use of a special case rule with the JOIN production being a general case or a special case of the other productions is another. If the JOIN production is made the general case, use of SC1, SC2, SC3, or R5 is appropriate. If the JOIN production is made the special case, rule SC4 can be used. To make the JOIN production a special case is restrictive, however, since the system cannot then add to its production memory FORK productions containing condition elements not already contained in the existing productions. D2 is again helpful in eliminating the need to disable instantiations after they are executed, though it is less essential here than in the case of FOREACH.

The simple implementation of EXTERNAL SEQUENCING requires a conflict resolution strategy which orders instantiations on the basis of the most recent element each contains and which is sensitive to element, rather than class, order so that the production performing the EXTERNAL SEQUENCING can order its signals as required. Only rules R1 and R5 meet these conditions.

As discussed above, the only mechanism needed to implement SUBROUTINEs, given GOTOs, is a parameter passing mechanism. The parameter passing mechanism must insure that the subroutine will process exactly the data passed to it, and nothing more nor less. No rule described here can guarantee that the subroutine will process all of its data before terminating. Rule R3 can, however, guarantee that the subroutine will not process more data than it should. If the data to be passed is asserted after a signal that enables the production that terminates the subroutine, R3 will cause that production to fire and terminate the subroutine before any of the other productions of the subroutine can make use of data asserted earlier than the signal.

If a strictly hierarchical control discipline is used, no more machinery is needed to implement nested control structures than is needed to implement single control structures. In such a control discipline, the order of instantiations in the control stack is exactly the order of their enabling. Hence, a conflict resolution rule sensitive to that order, such as R1, R2, or R5, is suitable for implementing the stack.

An overview of the relationships between each of the 14 conflict resolution rules and the characteristics of an adequate interpreter is given in Figure 2. A "+" indicates that a rule supports a characteristic, though in some cases additional rules are required in order to fully support that characteristic. A "-" indicates that if the rule is used, even in combination with other rules, then the characteristic cannot be

	1	2	3	4	5	GOTO	FRCH	F-J	E-S	SUBR	HIER
PO1					+			+			
PO2					+			+			
SC1				+	+			+			
SC2				+	+			+			
SC3				+	+			+			
SC4				-	+			+			
R1	+	+			+	+			+		+
R2	+	+			+	+			-		+
R3			+		+					+	
R4			+		+						
R5	+	+		+	+	+		+	+		+
D1					+	-	-				
D2	+	+	+		+		+	+			
AD1					-		-	-			

Figure 2: Rule Contributions

realized. The absence of a "+" or "-" indicates that the rule neither actively supports nor actively works against the characteristic, and that the rule can generally be combined with rules which do provide support without weakening that support.

V. COMBINATIONS OF CONFLICT RESOLUTION RULES

Since, as Figure 2 shows, no single rule supports all of the characteristics, any conflict resolution strategy using only one rule will be deficient. In this section we will show how rules can be combined so that all of the characteristics can be supported at least to some extent.

The most frequently used technique for combining rules is to place a priority ordering on the rules and then select the instantiations to be executed by means of a lexicographic sort. In other words, the first rule is applied to the instantiations in the conflict set to yield a subset of preferred instantiations; then the second rule is applied to this subset, and so on. Use of this technique gives the first rule the greatest significance and the last rule the least significance. In those cases in which one wants two or more rules to be equally significant, an alternative technique can be used. Each of the rules is applied to the same set of instantiations; the intersection of

the resulting subsets is the set of preferred instantiations. A single conflict resolution strategy may use both of these techniques. More precisely, the lexicographic technique can be modified so that at each step either a single rule or a set of rules is applied to the current set of preferred instantiations. We will use the connective "•" to indicate that two rules are to be applied to the same set of instantiations, and the connective "→" to indicate that the second of two rules (or sets of rules) is to be applied to the set of instantiations preferred by the first. "•" always has precedence over "→".

It is possible that when a rule is applied to a set of instantiations, no instantiation will be preferred. This would occur, for example, if D2 were applied to a set of instantiations all of which had been previously fired. Thus, for each rule (or set of rules) which proscribes certain instantiations, it is necessary to specify what is to be done if the set of preferred instantiations is empty. The alternatives, of course, are either to execute no instantiation or to continue with the lexicographic sort on the non-preferred set. If the instantiations in the non-preferred set are there because they are believed to be inappropriate (as opposed to less appropriate), then none of them should be executed. In our discussion below of specific conflict resolution strategies, we will indicate by enclosing a rule (or set of rules) in brackets that it excludes from further consideration all instantiations not in its preferred set.

Figure 3, which contains the same production system as Figure 1, shows the instantiations that would be preferred by the strategies that we will consider in this section. Again it should be assumed that (P S) and (Q T) were asserted on the previous cycle, (P T) and (R V) two cycles ago, (Q S) three cycles ago, (P V) four cycles ago, and (W V) and (W T) 101 cycles ago. In addition, assume that $I2_1$ has already been executed.

Before looking at conflict resolution strategies which combine rules in such a way that both the sensitivity and stability characteristics are adequately supported, we will examine a strategy which, although clearly inadequate, is of some historical interest.

One of the conflict resolution strategies used by PSG [Newell, 1973; Newell and McDermott, 1975] is typical of the strategies used in many early production system interpreters.[6] This strategy is [R4'] → P01 → R5'. Rule R4' is identical to R4 except that rather than using the number of cycles that an element has been in working memory as its criterion of recency, it uses the number of elements; only the first N elements in working memory are considered to be recent, where N is specified by the user. R5', unlike R5, uses both production memory (specifically, the order of conditions within each production) and working memory as knowledge sources. Since R5' uses the order of condition elements in a production as one of its criteria, it can be applied only to the instantiations of a single production. Those instantiations that contain the most recent element matching the first condition element in the production are preferred to all others. Within this set, those instantiations that contain the most recent element matching the second condition element are preferred, and so on.

6 PSG offers the user a choice of conflict resolution strategies. The one described here is the default strategy.

WM ((P S) (Q T) (P T) (R V) (Q S) (P V) ... (W V) (W T))

P1 ((Q =X) (P =X) --> ...)
$I1_1$ [P1 ((Q T) (P T))]
$I1_2$ [P1 ((Q S) (P S))]

P2 ((P S) (P =X) (W =X) --> ...)
$I2_1$ [P2 ((P S) (P T) (W T))]
$I2_2$ [P2 ((P S) (P V) (W V))]

P3 ((=X S) (=X =Y) (W =Y) (R =Y) (Q S) --> ...)
I3 [P3 ((P S) (P V) (W V) (R V) (Q S))]

P4 ((Q S) ¬(U S) (P =X) ¬(U V) ¬(U T) --> ...)
$I4_1$ [P4 ((Q S) (P S))]
$I4_2$ [P4 ((Q S) (P T))]
$I4_3$ [P4 ((Q S) (P V))]

[R4'] → P01 → R5' $\{I1_1\}$

[D2] → R1 → SC2 → R3 {I3}

[D2 • R4] → R1 → SC2 $\{I1_2\ I4_1\}$

[D2 • R4] → R5 $\{I1_2\ I4_1\}$

[D2 • R4] → R5 → P01 → AD1 $\{I1_2\}$

Figure 3: Conflict Resolution Using Combinations of Rules

The conflict resolution strategy is applied in the following way: R4' excludes from consideration all instantiations containing an element whose position in working memory is greater than N. Then P01 is applied to the remaining instantiations. Finally, R5' selects the single instantiation to be executed.[7] This strategy supports only characteristics 4 and 5, the FORK-JOIN, and to a limited extent, the GOTO.

One might wonder why such a weak strategy was chosen for PSG. Part of the answer is that since PSG had no older siblings, the disadvantages of its conflict resolution strategy were not well understood. The rest of the answer is that when PSG was designed, no one knew how to efficiently implement a system capable of computing the entire conflict set.[8] Several of the rules described in section III are completely beyond the power of a system that, like PSG, does not compute the entire conflict set.

7 For reasons of efficiency, this strategy was implemented in PSG in a way that makes generating the entire conflict set unnecessary.

8 See Forgy [1977] for a description of an interpreter which can efficiently compute the entire conflict set.

The remainder of this section will be devoted to a discussion of four conflict resolution strategies that are among the more adequate combinations of rules from section III. Since the rules are chosen from the same small set, these four strategies are all rather similar. We will not attempt to decide which of the four is best.

The first of these strategies is [D2] → R1 → SC2 → R3. If we ignore negative interactions among these rules, then this strategy, which has at least one rule supporting each of the characteristics, has no apparent weaknesses. And in fact, combining the rules in this order causes few negative interactions. One of the few is between R1 and R3. Preceded by R1, rule R3 no longer gives effective support to the SUBROUTINE construct. To reverse the order and put R1 after R3 is unacceptable, however. If R1 were preceded by R3 it would no longer support any of the characteristics. Another effect that should be noted is that since SC2 precedes R3, an instantiation, I_s, which is a special case of another instantiation, I_g, will be preferred to I_g even if it contains an element which is less recent than the least recent element contained in I_g. This effect could be eliminated by putting SC2 after R3. We elected not to do so because the justification for the R3 → SC2 combination, which would severly restrict the scope of the special case rule, seems rather weak. Finally, there is a positive interaction between R1 and SC2. Preceding SC2 by R1 causes the strategy, in effect, to use data recency (R1) to choose what to do next and then to use special case relationships (SC2) to choose how to do it.

The second strategy is like the first except that R3 is replaced by R4. One apparent advantage of R3 over R4 is that R3 helps in the implementation of subroutines while R4 does not. But, as pointed out above, R3 loses this advantage if it follows R1. Since R4 is used to exclude instantiations from consideration, it is applied first (together with D2). Thus, this second strategy is [D2 • R4] → R1 → SC2. Since no instantiation which contains a memory element more than 100 cycles old can ever fire, an obvious way to implement this rule is to delete memory elements automatically when they reach the age of 101 cycles. This implementation has a positive side effect: Since a production containing a negated condition element will be enabled only if no element in working memory matches that condition element, an indefinitely long working memory makes using negated condition elements somewhat difficult. When working memory has a limited size, negation has an essentially local effect which complements characteristic 3.

The third strategy is [D2 • R4] → R5. This strategy takes advantage of the fact that R5 has both data recency and special case characteristics to allow this one rule to replace the two rules, R1 and SC2, used in the first two strategies. The support provided by this strategy is virtually identical to that provided by the second strategy. Since R5 is a stronger ordering rule than R1, the number of instantiations executed on each cycle is likely to be less under this third strategy than under the second.

The first three strategies all allow multiple firings. Since multiple firings can result in a large number of actions being performed during a single cycle and thus in a possible loss of sensitivity, a strategy which always produces a unique preferred instantiation may be desirable. Such a strategy results if the third strategy above is extended with PO1 and AD1 to give [D2 • R4] → R5 → PO1 → AD1. An interaction

between PO1 and R5 causes them almost always to prefer a unique instantiation. The equivalence class of instantiations preferred by R5 will have exactly the same data; each will be an instantiation of a different production unless one or more of the productions can instantiate the same set of data elements in more than one way. Consequently, PO1, since it uses a complete ordering on productions as its ordering criterion, will ordinarily select one instantiation. In the rare case in which PO1 selects more than one instantiation, AD1 will produce a unique instantiation.

ACKNOWLEDGMENTS

The development of many of the ideas discussed above owes much to the members of a production system project at Carnegie-Mellon University. The members of this project, in addition to the authors, are P. Langley, A. Newell, K. Ramakrishna, and M. Rychener.

REFERENCES

Davis, R. and King, J. An overview of production systems. Technical report. Department of Computer Science, Stanford University, 1976.

Forgy, C. A production system monitor for parallel computers. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1977.

Hayes-Roth F. and Lesser, V. Focus of attention in the Hearsay II speech understanding system. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1977.

Newell, A. Production systems: models of control structures. In Chase, W. (ed.), **Visual Information Processing**. Academic Press, 1973, pp. 463-526.

Newell, A. and McDermott, J. PSG manual. Department of Computer Science, Carnegie-Mellon University, 1975.

Rychener, M. D. Production systems as a programming language for artificial intelligence applications. Technical Report. Department of Computer Science. Carnegie-Mellon University, 1976.

Waterman, D. A. Adaptive production systems. Technical Report. Department of Psychology. Carnegie-Mellon University, 1974.