NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Control Requirements for the Design of Production System Architectures

Michael D. Rychener
June 1977

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

This paper has been submitted for presentation at the AI ⊗ PL Symposium, ACM SIGART - SIGPLAN, Rochester, NY, August, 1977.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

510.7808
C28r
Rychener, Michael D.
1977

Table of Contents

SECTION			PAGE
1	Introduction		1
	1.1	History and Definition	1
2	Examples of Control Requirements		4
	2.1	Sequencing and Subroutining	5
	2.2	Iteration and Possibilities Generation	9
	2.3	Hierarchical organization	11
	2.4	Selection	12
3	Summary		14
	3.1	Acknowledgments	14
4	References		14

JUL 27 '77

CARNEGIE

Abstract. Programs in the artificial intelligence domain impose unusual requirements on control structures. Production systems are a control structure with promising attributes for building generally intelligent systems with large knowledge bases. This paper presents examples to illustrate the unusual position taken by production systems on a number of control and pattern-matching issues. Examples are chosen to illustrate certain powerful features and to provide critical tests which might be used to evaluate the effectiveness of new designs.

1. Introduction

There are a number of common control usages in programs in the artificial intelligence (AI) domain that impose requirements on control structures. Production systems (PSs) are a control structure with promising attributes for building generally intelligent systems with large knowledge bases. The PS approach to a number of control issues is unusual and is sufficiently novel to warrant a detailed discussion. This paper gives a number of examples of the constructs used for control in some existing PS implementations of AI systems.

Examples are also used here to justify a number of design features of particular PS languages. That is, examples are chosen to illustrate certain powerful features and to provide critical tests which might be used to evaluate the effectiveness of new designs. This concern for explicit, detailed design justification arises from a perceived failure of language designers within AI to communicate such aspects for existing designs. Justification tends to be neglected both for basic language principles and for low-level language features such as pattern-matching primitives. Because of a number of specific objections to the PS approach historically, the evaluation of PSs as an AI language has been done with more than the usual care (Rychener, 1976). This paper draws on that evaluation experience.

1.1. History and Definition

The use of PSs in AI derives from research in several fields of computer science. Their invention as a formal specification of algorithms dates from the mid-1940s, as Post productions or Markov algorithms (Minsky, 1967). Floyd-Evans productions are a variant on the PS concept used for parsing programming languages (Evans, 1964). Most significantly, PSs have been adapted to the task of modelling human memory and problem-solving processes (Newell, 1972; Newell and Simon, 1972). Within AI, there have been a number of successful projects involving PSs or similar rule-based architectures, to various degrees: Waterman's (1970) poker learning program, the Heuristic DENDRAL program (Buchanan and Sridharan, 1973), and Shortliffe's MYCIN medical diagnosis program (Davis, et al., 1975).

A PS is a set of condition-action rules representing an algorithmic procedure on some domain. A rule, or production, applies to an element of the domain whenever its condition is true. The application of the production results in executing its action, producing another domain element. In AI applications, the domain is typically a space of symbolic models of situations. A production's condition is a conjunction of schematic patterns for symbol structures, and its action is an unconditional sequence of additions, modifications, replacements, and deletions of symbol structures. Sequences of symbolic changes, resulting when productions are applied to a model, are taken to correspond to the modelled system's dynamic behavior.

To narrow the scope to a practical or definite computational tool requires the specification of a production system architecture. Such an architecture has four components: Working Memory, Production Memory, a recognize-act cycle, and a procedure for resolving conflicts between competing productions.

Working Memory is the structure containing the dynamic knowledge state of the system, referred to above as a model of a situation. Abstractions of Working Memory elements are the primary constituents of production conditions, and manipulations of Working Memory elements are the primary constituents of production actions. Specifying the Working Memory places constraints on the attributes of its elements and on the relationships between elements.

Production Memory contains all of the productions, and its specification defines allowable forms for productions and their relationships within the memory structure. Production actions usually include operators for modifying the Production Memory.

The recognize-act cycle serves to control the application of productions. The usual form is that first a recognition occurs, in which a production or a set of productions is found to have its conditions satisfied with respect to the present Working Memory. The recognition usually involves matching abstract forms to specific elements. Then a selection from the recognized set is made, and the corresponding sequences of actions are performed. Performing the actions results in a new Working Memory state, and the cycle starts over with another recognition.

The selection from the set of recognized productions is according to conflict resolution principles. These principles are usually based on the static structure of Working Memory or Production Memory, or on dynamic aspects of the system's operation such as recency of addition.

The particular architecture and language used here is called OPS (Forgy and McDermott, 1976). Production Memory in OPS is an unstructured, unordered set of productions. Working Memory is likewise an unordered set of list structures, without duplications. It is bounded in size, by deleting elements whose last assertion occurred more than some arbitrary number of system actions in the past (currently 300).

For conflict resolution the following rules apply, in order (McDermott and Forgy, 1977).

1. Refraction: a production is not fired twice on the same data (instantiation of a pattern) unless some part of that data has been re-inserted into Working Memory since the previous firing. This prevents most infinite loops and other useless repetitions.

2. Lexicographic recency: the production using the most recently inserted elements of Working Memory is preferred. "Most recent" is determined lexicographically, i.e., if there is a tie on the most recent element used, the next-most recent elements are compared, and so on. This rule serves to focus the attention of the system very strongly on more recent events, allowing current goals to go to completion before losing control.
3. Special case: a production is preferred that has more conditions, including negative conditions which do not match to specific memory elements. Most of the meaning of having one production be a special case of another is captured by rule 2, since a special case that uses more data than a general one is lexicographically more recent (by the OPS definition). Preferring special cases to general ones follows the expectation that a specific method is more appropriate to a situation than a more general one.
4. Production recency: the more recently created production is preferred. This is used only in systems that grow by adding productions dynamically and only in the case of productions with identical conditions. In such a context, the more recent production is taken as more appropriate.
5. Arbitrary: a selection is made among multiple matches to the same production using the same data.

As a matter of practice, conflict resolution rarely requires more than the first two rules.

OPS has several other distinguishing features. The pattern matching allows a limited form of segment variables, namely, a variable may match an indefinite-sized tail of a list. The Pattern-And (Pand) feature allows an expression to be matched to several patterns, and then bound to a variable. OPS allows complex negative conditions to be specified, for instance, including the negation of an entire production condition within the condition of another production. Productions in OPS are compiled into an efficient network form, rather than interpreted. OPS has an operator for adding productions to Production Memory which have been formed (in terms of an appropriate data structure) in Working Memory; such additions are done directly into the compiled network during the runtime cycle.

The following section will explain the OPS notation as examples are introduced.

In addition to the definition of our PS given above, our approach has a number of distinctive features. A major part of our approach lies in representation assumptions. Working Memory, though large, is considered to be short-term only. All long-term facts and interconnections between them (e.g. semantic networks) are stored as productions. Thus all augmentation of a PS by itself is done by forming new productions. The way that action develops from the PS differs from some others in being a forward recognition-driven cycle, rather than a backward-chaining, goal-driven cycle, as in the MYCIN system (Davis, et al., 1975). The system is controlled by signals and symbol structures in the global Working Memory, called goals, which are included explicitly in production conditions when appropriate. This is in contrast to MYCIN and to DENDRAL (Buchanan and Sridharan, 1973). The PS architecture is used as the total system, rather than having it be one of a number of procedural components. Other systems have employed additional, non-PS procedures for such activities as modifying and analyzing the PS. Working Memory is arbitrary list structures in an extensive database-like structure, with a vast majority of items explicitly stored rather than represented as computable predicates. Production conditions make use of general pattern-matching capabilities, as is common in other recent

AI languages (Bobrow and Raphael, 1974). Though the general architecture derives from concern for human cognition (Newell, 1972, Newell and Simon, 1972), little consideration is given to psychological constraints.

2. Examples of Control Requirements

The examples in this section demonstrate the effect of conflict resolution principles and of pattern-matching primitives on the ease of achieving control. Control in PSs is primarily through goals in Working Memory. A goal is loosely defined to encompass:

1. a description of the purpose or desired final state of processing, or a specification of a problem operator to be applied; thus it is a focus, or something to come back to during processing; there may be tests (productions) associated with a goal, to ensure that it is properly achieved;
2. possible relations to other goals, e.g. subgoal-supergoal;
3. a history of attempts to achieve the goal, and of their results;
4. associated methods, operators, data objects, heuristics, and priority orderings.

The following is an example of a goal:

```
(PUTON SET (WANT) SET-3 ON BLOCK-4)
```

This might be read: "want to puton the set Set-3 on Block-4"; "puton" is a specific operator for the system in which the goal occurs. Goals and other Working Memory elements are represented, by convention, as lists (in LISP notation) whose first two positions give the main goal class (e.g. PUTON SET), whose third position gives a "modality" (e.g. (WANT)), and whose remaining positions are a description. This example is a simple form, having only the first component of the above definition. More complex goals and other examples of simple goals are discussed below.

Productions assert such control goals to achieve sequencing between steps in a process, to coordinate hierarchies of control, to evoke methods to achieve subgoals, and to control iterations. Other productions respond to such goals, taking account of both their content and the surrounding Working Memory data context. Goals may vary in complexity from single list structures to complex Working Memory and Production Memory combinations. Longer-term control can be achieved by adding productions to Production Memory that can respond to such goals. There is an OPS action to facilitate this.

The following discussion of control is based primarily on analysis of the PS implementations of a number of "classical" AI programs (Rychener, 1976): Bobrow's Student, Feigenbaum's Epam, GPS of Newell et al., a King-Pawn-King chess endgame program, and a natural language understanding program coupled with a toy blocks problem-solving program as done by Winograd.

2.1. Sequencing and Subroutining

Sequencing is performing the steps of a process in a specified order. Because PSs are sequenced only by the recognize-act cycle, in which potentially a large set of productions are candidates for firing, there is both the need for explicit sequencing by Working Memory goals and the opportunity for developing forms of sequencing not available in conventional control structures.

Subroutining is the suspension of a process while essential secondary results are obtained by some other process. This encompasses such mechanisms as the conventional subroutine and the evocation of a subgoal in the process of achieving a goal. Conventionally, this action causes the establishment of a local control and data context, such that the subroutine has limited access to the environment. But in PSs, only the control context is local, while the data context remains the global Working Memory. The control context is local only to the extent that the "subroutine" maintains its own control goals and continues to react to them in a dominant way according to the conflict resolution principles. In the OPS architecture, at least, subroutines composed of subsets of the entire PS cannot be established by any structural means.

There are seven ways that such control is achieved.

1. Direct sequencing by goals between sets of productions: Each production in a set of productions to perform a step in a complex decision process includes as an action a goal that evokes the next step. Such goals are stated generally, as opposed to being production-specific signals. It is a characteristic of AI domains that long unconditional sequences of actions are rare. At least this is the case when AI programs are implemented as PSs, which is a fairly high level of expression. That is, it is common to alternate, at a high rate, tests of conditions with state-changing actions. (But this is to be expected, given the nature of intelligence.) Thus, it is common that sequencing between sets of productions is required to arrive at a complicated decision, with each step in the sequence contributing to some aspect. For example,

```
S1  (PUTON SET (WANT) =S ON =O)
    (SET MEMBER (HAVE TRUE) =M OF =S)
    (OBJECT SIZE (HAVE TRUE) =X OF =M)
    (NOT (SET MEMBER (HAVE TRUE) =N OF =S)
         (OBJECT SIZE (HAVE TRUE) >X OF =N) )
--> (PUTON OBJECT (WANT) =M ON =O) ;
```

An OPS production consists of a name (in this case S1), a list of conditions (the first four lists, headed by PUTON, SET, OBJECT, and NOT, in S1), an arrow "-->", and a list of actions (the second PUTON element is the only one in S1), terminated by ";". In order to fire, the condition of a production is matched to Working Memory (in the conventional pattern-match sense), setting up a correspondence of condition elements to memory elements (which usually includes the binding of variables in the conditions to tokens within memory elements). Then, using that correspondence to instantiate them, the actions of the production are performed. Unless otherwise defined, an action is a simple insertion of the instantiated element into Working Memory. The most common other action is DELETE, which removes an element from Working Memory.

Pattern variables are denoted by a variety of prefixes: "=" for ordinary ones which can have values bound to them or which, when bound, must match that value; "≠" for those that match anything other than the value bound; ">" and "<" that match if the value to be matched is greater than or less than the value bound to the variable, respectively. "NOT" specifies that the pattern match fails if an attempt to match the elements in its scope succeeds.

S1 is part of a process to put a set of objects onto another object, in a toy blocks world environment. It responds to the main "puton" goal (first condition), selects a member of the set to be put (second condition), and force the selected member to be the one with the greatest size (third and fourth conditions; the "NOT" says there is no other set member with a greater size). The action of S1 is to assert a subgoal to "puton" the object selected from the set. (This is a simplified version which ignores subtleties of bookkeeping and maintaining progress through the set of objects.) For instance, S1 would match,

```
(PUTON SET (WANT) SET-3 ON BLOCK-4)
(SET MEMBER (HAVE TRUE) BLOCK-1 OF SET-3)
(OBJECT SIZE (HAVE TRUE) 15 OF BLOCK-1)
```

and insert a goal,

```
(PUTON OBJECT (WANT) BLOCK-1 ON BLOCK-4)
```

S1 is supposedly part of a multi-step process, each step of which makes some decision or selection and then directly evokes the next step. For instance, it might be: select the largest set member, verify that space is available to put it on, select the exact location, and do the actual put. Each such step might involve several productions (or evocations of subgoals for more complicated processing), to handle the various conditions possible in the environment.

In general, the examples to be presented are just isolated parts of a large Production Memory. There are usually a number of productions with similar conditions, responding to similar goals under various contexts, etc. What is presented here can only be informal hints about the surrounding memory context and problem environment. Also, productions are sometimes simplified slightly to emphasize the essential control aspects.

2. Fall-back control: A process evokes another by some goal structure, and along with that asserts a continuation signal, which, when it eventually becomes dominant in conflict resolution by its recency, evokes whatever is to follow. For this case, the results developed by the evoked process to satisfy its own goals are not used directly by the continuation in the evoking process. (Variations are given below.)

```
S2   (GRASP OBJECT (WANT) =B)
     (GRASPING OBJECT (HAVE TRUE) =B2 $ ≠B)
-->  (GETRIDOF OBJECT (WANT) =B2)
     (GRASP OBJECT (WANT (STEP 2)) =B) ;

S2'  (PICKUP OBJECT (WANT) =B)
-->  (GRASP OBJECT (WANT) =B)
     (RAISE HAND (WANT)) ;
```

The goal in S2 is to grasp an object. S2 gives the case where the hand is already grasping some other object, so that a goal to get rid of the other object is necessary. The "$" notation in the second condition stands for "pattern-and", a conjoiner of two patterns. That is, "$" forces both "=B2" and "≠B" to match the same token from the Working Memory element. In this case, it amounts to allowing the variable B2 to be bound during the match to anything except the value bound to B. The two goals that are actions in S2 are to be ordered in terms of recency, with the leftmost one being the more recent. That is, in OPS, actions within the same production are given distinct "times" with respect to conflict resolution, and are ordered in decreasing recency from left to right. The effect in this case is that the GETRIDOF goal becomes dominant first, and later on, control falls back to focus on the GRASP goal. Notice that the GRASP goal is stated as a continuation of the main GRASP goal, as denoted by the (STEP 2) marker in the modality position.

S2' gives another goal, to pick up something. It breaks the goal into two subgoals, GRASP and RAISE, which are sequenced by their order in the production. Note that there is no continuation of the main goal as in S2, as a result of the adequacy of the two subgoals given, both to solve the task completely, and to do so without further testing of conditions between the two steps.

3. Direct result usage: A process evokes another, and perhaps asserts a continuation goal, but unlike the preceding, the continuation makes direct use of results of the evoked process, so that continuation occurs as soon as sufficient results are developed by the evoked process. Note that using the Working Memory recency conflict resolution principle allows some eventual "fall-back" to the evoked process, which might give rise to yet more results, in case the evoked process is left unfinished by the initial result-use continuation.

```
S3  (GRASP OBJECT (WANT (STEP 2)) =B)
    (GRASPING OBJECT (HAVE TRUE))
--> (MOVE HAND (WANT) TO =B)
    (GRASP OBJECT (WANT (STEP 3)) =B) ;
```

S3 continues the method to achieve the GRASP goal by responding to a GRASPING memory element with nothing in the fourth position. Note that S3 becomes dominant as a result of a new GRASPING element, while generally the GRASP OBJECT goal continuation would not yet be dominant. The more general case of this form of production would involve recognizing results developing from a selection or generation of elements, under the conditions that the elements can be further processed without waiting for more of the same elements to be created. The following shows an alternative.

4. Held result usage: A process evokes another, and its continuation makes use of results of the evoked process, but the continuation is held from proceeding until all of the results of the evoked process are developed. This might be necessary, for example, when some comparison or selection is to be made from among them. This is ensured through a Working Memory element that is less recent than the process evocation, and whose eventual becoming dominant in conflict resolution results in firing a production that asserts the continuation goal.

```
S2"  (GRASP OBJECT (WANT) =B)
     (GRASPING OBJECT (HAVE TRUE) =B2 $ ≠B)
---> (GETRIDOF OBJECT (WANT) =B2)
     (GRASP OBJECT (HOLD (STEP 2)) =B) ;

S4   (GRASP OBJECT (HOLD (STEP 2)) =B)
---> (GRASP OBJECT (WANT (STEP 2)) =B) ;
```

S2" is a minor variant of S2, with HOLD in the second action in place of WANT. Here S4 responds only to the HOLD goal, so that the WANT is not emitted until it becomes dominant, all by itself. This ensures that all action started by the GETRIDOF goal in S2" goes to completion before continuing, since that action will all be more recent than the held goal. This would be proper to a context where an entire set of objects, say, were to be generated before continuing.

5. Complex goals as focus of control: Goals in an organized process such as heuristic search can be composed of a number of possibly optional attributes. Such goals represent major processing states, and an executive is required to manage the goals, evaluating progress, measuring difficulty, propagating success and failure, ordering their consideration, and allocating processing effort. Such an executive can, however, be achieved with a relatively small number of productions (see the discussion of a PS for GPS, Rychener, 1976). The attributes, kept as separate elements in Working Memory (due to the variation in their relevance to different goals, it would be cumbersome to keep them together in one), are held together by association, through a token for the goal, GOAL-5 in the following:

```
(GOAL OBJECT (HAVE TRUE) GOAL-5 ARRANGEMENT-3)
(GOAL SUPER (HAVE TRUE) GOAL-5 GOAL-2)
(GOAL DIFFERENCE (HAVE TRUE) GOAL-5 (LEFT SIDE LOW))
(GOAL STATUS (HAVE TRUE) GOAL-5 SUSPENDED)
(GOAL TYPE (HAVE TRUE) GOAL-5 TRANSFORM)
```

Goal attributes can be used to give AND-OR structuring to a collection of goals, and can indicate how the search is to continue when a success or failure occurs. Such goals need to be in Working Memory when active, but are most conveniently stored in Production Memory on becoming less active. Such storage simply involves collecting the attributes for a goal into a single action side of a new production, to be evoked by the name token of the goal, on demand (presumably that token is included in the attribute of some other goal, and has become relevant to further progress). The PS approach to backtracking is, first, to avoid it wherever possible by analyzing the problem and including more intelligence as heuristics and domain knowledge. But when necessary, it can be easily achieved within such a complex goal framework, by making explicit the information on available alternatives at choice points in the search, and by augmenting the executive to make use of it (see Rychener 1976 for an example of how this worked out in practice).

6. Fork-join: Because tight control is not required, the work on several goals can proceed as if asynchronously. (A realistic simulation of this might occur if conflict resolution were loosened to allow more than one production firing per cycle.) Some process evokes a number of others ("fork"), and they each develop their results incrementally, reacting to things in the total Working Memory state. The "join" consists of a production to test that

results are obtained by the various processes. Sometimes more control is necessary to ensure they all complete before other productions become active: a test production recognizes some part of the desired results and re-evokes the incomplete aspects, by re-asserting their goals, thus making them more recent and dominant according to conflict resolution. Sequencing goals as in the fall-back control case above can be used to evoke the checking of validity of the "join". Such signals would not dominate until the other actions had run their course.

7. Default/Update: Steps in a decision are organized as an initial step in which a default answer is established in Working Memory, and then other productions are allowed to modify that result according to various special-case conditions. As in fork-join, sequencing goals can be used to enable the next stage of processing to take place.

```
S7   (FIND BOUNDARY (WANT) LOWER (RANGE 1 40) AROUND 17)
--> (FIND BOUNDARY (HAVE RESULT) LOWER 1 (RANGE 1 17))
     (FIND BOUNDARY (WANT) HIGHER (RANGE 1 40) AROUND 17) ;

S7' (FIND BOUNDARY (HAVE RESULT) LOWER =N (RANGE =R1 =R2)) $ =C1
     (OBJECT LOCATION (HAVE TRUE) >N $ <R2 $ =M FOR =O)
--> (FIND BOUNDARY (HAVE RESULT) LOWER =M (RANGE =R1 =R2))
     (DELETE =C1) ;
```

Here, the problem is to find the object with the highest location near a point within a range, a simplified one-dimensional case of the problem of finding space to place an object in a region. This can be done more concisely in the one-dimensional case than is exhibited here, but this default approach is useful in more complex higher-dimensional cases, where the various cases arising prohibit expression as a single complex condition. The goal to find the higher boundary (second action in S7) represents a continuation action that will take over after any update productions fire.

2.2. Iteration and Possibilities Generation

Iteration is a process in which the same steps are repeated a number of times as dictated by the members of some given set of data objects. Generation of possibilities is a converse process, creating a set of objects according to existing memory context or constraints. Generation poses control problems when it occurs in a context in which some other process is seeking the generation of an element with particular properties or consequences, whereupon the generation stops. Often the test for whether generation is to stop involves considerable computation, so that there is a problem of maintaining for the generator memory of its status with respect to internal control and the set being generated.

1. Deliberate loops: Iteration takes place under the control of an explicit goal. The looping goal appears in each production, and is re-asserted into Working Memory at each iteration to maintain its dominance with respect to conflict resolution. In practice, the productions might be separated into "body" productions, which do the main work of the loop, and "bookkeeping" productions, which update status information and test for termination. Alternatively, each production can contain both the body and bookkeeping portions (a less modular form). An example of the separated form:

```
I1   (WORD LIST (WANT GATHER) SPECIAL IN =C) $ =C1
     (CHUNK TEXT (HAVE TRUE) =C =W .=X) $ =C2
--> (WORD LIST (WANT GATHER (STEP 2)) SPECIAL =W)
    =C1
     (CHUNK TEXT (HAVE TRUE) =C .=X)
     (DELETE =C2) ;

I1' (WORD LIST (WANT GATHER (STEP 2)) SPECIAL =W) $ =C1
     (WORD CLASS (HAVE TRUE) SPECIAL OF =W)
     (WORD LIST (GATHERING) SPECIAL .=X) $ =C3
--> (WORD LIST (GATHERING) SPECIAL .=X =W)
     (DELETE =C1) (DELETE =C3) ;

I1" (WORD LIST (WANT GATHER) SPECIAL IN =C) $ =C1
     (CHUNK TEXT (HAVE TRUE) =C)
     (WORD LIST (GATHERING) SPECIAL .=X) $ =C3
--> (WORD LIST (HAVE TRUE) SPECIAL .=X)
     (DELETE =C3) (DELETE =C1) ;
```

The loop is to collect all the words in the text of a "chunk" that are of type "special" into a separate word list. I1 and I1" are the bookkeeping productions (the latter being the termination), and I1' is the body of the loop. I1 and I1' alternate in firing until the termination condition occurs, detected by I1". A new notation element is ".", which is used to mark segment variables, variables that match an arbitrary list tail (ordinary variables only match a single list element). Note that the occurence of "=C1" in the action side of I1 causes a re-insertion into Working Memory of the goal of the production, an act necessary in general to keep the loop productions dominant in conflict resolution.

Generally, such an iteration structure is used when the body of the loop consists of many productions, making it more awkward to include the bookkeeping actions in all the body productions as would be simple to do in this example (essentially, combining I1 and I1', dropping the "step 2" goal).

2. Single-production iterations: Only one thing is done over the iteration, so that a single production expresses the action at each point along with loop bookkeeping. Typically, this is used to emit subgoals for each element of a set, along with a re-assertion of the iteration goal; it is also used to collect elements into a list in some order, where the newly-updated list serves to re-evoke the single production and continue the loop. It should be clear from the preceding example how such a form is achieved.

3. Parallel iteration: This is looping in which the iteration takes place as a result of having conflict resolution allow more than one production firing per cycle, rather than under the control of explicit looping goals. The productions specifying the action are written as if for a single object in a set, and the multiple firing ensures that all elements are processed. This has been used (Rychener, 1976) in generating language for replies in natural language programs (e.g. for several similar descriptive noun phrases at the same time), in expanding in breadth-first fashion transitive data relations (cf. spreading activation in a semantic network), in simple combinatorial generation processes, where a single condition is fulfilled by a number of generated possibilities, and in other uncomplicated iterations. There are two kinds of this parallel iteration, just as for the deliberate iteration above: single-production and multiple-production. This capability is not part of OPS at present.

4. Generation of possibilities: As mentioned above, the main problem with this process is maintaining a memory of the status of the generator, especially those items already generated and those remaining to be generated. There are a large number of ways to handle this, due to the inherent flexibility of the PS architecture, particularly its two memories. For instance, possibilities can be kept in Working Memory, and erased as tried, if the set is relatively small; possibilities can be generated as needed, by specific productions, with Working Memory storing what has already been tried; and productions can record the elements already generated, so that a simple generator of all elements, followed by erasure of the elements already generated, followed by a selection from the remaining elements, can suffice.

There are a number of decisions to make in forming a generator and thus there is a space of flexible ways of responding to the problem: whether to save the elements already generated or those not yet generated; whether to save them as productions or as Working Memory elements; whether to save the elements in a single memory structure or separately (for productions, elements might be accessible individually with the selection pre-determined by explicit conditions, or all in the same production, with a further selection necessary after firing the production); whether a production is set up to assert desired elements or erase those already generated (assuming the full set is in Working Memory); whether to generate the entire set or somehow partition it for more gradual generation; whether the set should be computed and stored, or recomputed on demand; and whether to update the status of the generator by erasure from Working Memory, by superseding an existing production with an updated one, or by adding elements to Working Memory that have to be tested and excluded in further production matching within the generator.

2.3. Hierarchical organization

There are three mechanisms used to achieve some kind of hierarchy: the supergoal-subgoal relationship represented by productions that respond to a goal by setting up subgoals; attaching tags to data items, amounting to pointers for a graph structure; and organizing processes in a bottom-up hierarchy, where each level is evoked as results from the lower level are developed. The first mechanism has already received attention in the examples above.

The second mechanism is used to keep track of tree-structured expressions, as might be used to represent linear algebraic equations (cf. Bobrow's Student). It is also used more generally to keep track of goal structures. As an example, suppose a sentence in an algebra word problem were being parsed to form an arithmetic expression. The main operator in the sentence would cause the text to be split into left and right operands, and then those would be further parsed to determine their expressions. Labels might be set up to record the relation between the operands and the containing expression as follows:

```
(EXPR LABEL (HAVE TRUE) LEFT CHUNK-5 PARENT CHUNK-3)
(EXPR LABEL (HAVE TRUE) RIGHT CHUNK-6 PARENT CHUNK-3)
(EXPR OPERATOR (HAVE TRUE) TIMES FOR CHUNK-3)
```

Suppose the chunks of text for the left and right halves were parsed to produce,

```
(CHUNK EXPR (HAVE TRUE) CHUNK-5 (PLUS X Y))
(CHUNK EXPR (HAVE TRUE) CHUNK-6 2)
```

Then a production can recognize this and combine the results:

```
H1  (CHUNK EXPR (HAVE TRUE) =C1 =E1)
    (EXPR LABEL (HAVE TRUE) LEFT =C1 PARENT =C)
    (EXPR LABEL (HAVE TRUE) RIGHT =C2 PARENT =C)
    (CHUNK EXPR (HAVE TRUE) =C2 =E2)
    (EXPR OPERATOR (HAVE TRUE) =O FOR =C)
--> (CHUNK EXPR (HAVE TRUE) =C (=O =E1 =E2)) ;
```

Here, the various tags have served to record the structure, and the variable bindings have used the tags to recover it in the final expression. The tags are ad hoc, in the sense that other systems of tags are used for other tasks, e.g. maintaining goal interrelationships as mentioned above. But the PS approach allows flexibility of choice to meet task demands.

The third hierarchy mechanism, bottom-up organization, is used in processing natural language text, representing lexical, syntactic, semantic, and pragmatic levels of processing. Like the first mechanism above, it is represented in Production Memory as connections between the results produced by a level in the hierarchy and the data and goals of the next higher level. For instance, a word recognized at the lexical level is given a word class, which then can be used at a grammatical level to check that the class occurs appropriately for the grammatical context; the success of the grammar check (represented in Working Memory as the resulting grammatical function performed by the word) then leads to the lowest-level semantic consequences, and so on. Note that this is counter to a style that would establish goals to apply the various knowledge levels. Rather, the triggering of a level is directly dependent on lower-level results. In many cases, the higher levels are evoked only after a number of words have been processed, allowing a suitable higher-level result to be assembled (see Rychener, 1976, for details).

2.4. Selection

The power of the PS match is exploited in complex selections, which occur quite frequently in AI tasks. These selections typically conjoin a number of conditions, each of which narrows down the set of matching candidates. Two powerful means for facilitating this narrowing down are: the use of computable predicates on the values bound to match variables (as opposed to pattern matches that simply bind variables to values from corresponding Working Memory items); and the use of a "maximal" ("minimal") operator, which selects from a set of possible variable bindings during the match the one that is maximal (minimal) according to a computable predicate. The maximal (minimal) operator corresponds exactly to the narrowing-down concept, and it expresses concisely what would otherwise be a complex logical condition. At present, the maximal and minimal operators are not implemented in OPS. The ">" and "<" variable prefixes occurring in some examples above are special cases of the use of computable predicates in the match. OPS does have more generality in this respect than is indicated in examples here.

Some examples of complex selections:

1. Selection of an old goal in GPS: This complex test involves conjoining the following tests: the goal is type Reduce; the goal is not in state "methods-exhausted"; the goal is minimal according to its difficulty; the goal's supergoal is not an apply type goal where there also exists such a goal whose supergoal is a transform type goal; and the goal is minimal (or equivalently maximal) according to some arbitrary predicate whose function is to select from a set of otherwise-equivalent choices - in OPS, one arbitrary way is automatically provided in the lexicographic event recency order, which distinguishes between elements according to their time of assertion.

This complex selection can be represented by the production,

```
C1  (SELECT GOAL (WANT) OLD)
    (GOAL TYPE (HAVE TRUE) =G REDUCE)
    (NOT (GOAL STATUS (HAVE TRUE) =G METHODS-EXHAUSTED) )
    (GOAL DIFFICULTY (HAVE TRUE) =G =N)
    (MINIMAL =N GREATERP)
    (GOAL SUPER (HAVE TRUE) =G =G2)
    (NOT (GOAL TYPE (HAVE TRUE) =G2 APPLY)
         (GOAL TYPE (HAVE TRUE) <>G $ =G3 REDUCE)
         (NOT (GOAL STATUS (HAVE TRUE) =G3 METHODS-EXHAUSTED) )
         (GOAL DIFFICULTY (HAVE TRUE) =G3 =N)
         (GOAL SUPER (HAVE TRUE) =G3 =G4)
         (GOAL TYPE (HAVE TRUE) =G4 TRANSFORM) )
    (MINIMAL =G ARBITRARYP)
--> (SELECT GOAL (HAVE RESULT) =G) ;
```

Note that the next-to-the-last clause in the selection to be done, as stated informally above, has an awkward expression as the seventh condition in C1. The maximal and minimal operators are designed to avoid such code, but are not sufficiently powerful in this case. For instance, the first "minimal" condition in C1 is equivalent to "not: a reduce goal, not methods-exhausted, with difficulty less than n". This example's principal lesson, perhaps, is that the NOT operator as formulated here is very powerful in expressing selections. Though maximal and minimal supplement its power, there may be still better primitives.

2. Select a block to put next in a stack: This test involves: the block has not already been tried, for this stack; the block is not already on the stack; this block is maximal on block size of such blocks; this block is minimal on some arbitrary predicate (cf. the last test in 1. above).

This selection and the following one are made more complicated than they seem because the size of a block is not stored explicitly in Working Memory but is always computed as a function of the three linear dimensions of the block (this is an arbitrary restriction inherited from the blocks problem-solving PS previously implemented; it is kept here to illustrate a requirement for the power of maximal). The actual size function used can depend on the context. In this case, the height of the block is irrelevant, for instance. The condition here is expressed,

```
(MAXIMAL (PLUS =X =Y) GREATERP)
```

That is, the pair such that their sum is minimal is wanted, where X and Y are bound during the match to the length and width of the block.

3. Select a block to move to make space: This involves: the block is on the block on which space is to be made; the block is large enough so that moving it will create the needed amount of space; the block has minimal size among such blocks; the block is minimal according to some arbitrary predicate (cf. the last test in 1. above).

3. Summary

The purpose of this extended presentation and discussion of detailed examples has been to exhibit a number of useful control requirements for testing new PS designs. The aspects of control touched on here are considered basic, especially for AI applications, and are rather different from control constructs in more conventional langauges. This is due both to domain characteristics and to the unique perspective imposed by using PSs. Surveying the examples presented, the contribution to control by the use of Working Memory recency as a conflict resolution principle (rule 2 in Section 1.1) is central. It allows the expression of control demands as global goals, avoiding the ad hoc inter-production signals that plagued early PS programming attempts. The goals used here achieve control using conventions that are uniform over all such goals, rather than private to particular productions. A secondary purpose has been to present evidence that PSs have openness and flexibility in the varieties of control achievable in AI programs, to an extent surpassing conventional control structures.

3.1. Acknowledgments

Allen Newell provided the initial motivation for exploring the power of PSs as programming languages. Lanny Forgy and John McDermott implemented the OPS language used in this paper, improving on past PS designs.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

4. References

Bobrow, D. G. and Raphael, B. R., 1974. "New programming languages for artificial intelligence research", *Computing Surveys*, Vol. 6: 3, pp. 153-174.

3 8482 00571 6176

Buchanan, B. G. and Sridharan, N. S., 1973. "Analysis of behavior of chemical molecules: Rule formation on non-homogeneous classes of objects", *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 67-76. Also Stanford AI Memo 215, Stanford University Computer Science Department.

Davis, R., Buchanan, B. and Shortliffe, E., 1975. "Production rules as a representation for a knowledge-based consultation program", Report STAN-CS-75-519, Memo AIM-266. Stanford, CA: Stanford University, Computer Science Department.

Davis, R. and King, J., 1975. "An overview of production systems", Report STAN-CS-75-524, Memo AIM-271. Stanford, CA: Stanford University, Computer Science Department.

Evans, A., 1964. "An ALGOL 60 compiler", in Goodman, R., Ed., *Annual Review of Automatic Programming*, Vol. 4, pp. 87-124. New York, NY: Pergamon Press.

Forgy, C. and McDermott, J., 1976. "The OPS reference manual", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

McDermott, J. and Forgy, C., 1977. "Production system conflict resolution strategies", in D. A. Waterman and F. Hayes-Roth, Eds., *Pattern-Directed Inference Systems*, New York, NY: Academic Press. Forthcoming.

Minsky, M., 1967. *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ: Prentice-Hall. Chapter 12.

Newell, A., 1972. "A theoretical exploration of mechanisms for coding the stimulus", in Melton, A. W. and Martin, E., Eds., *Coding Processes in Human Memory*, pp. 373-434. Washington, DC: Winston and Sons.

Newell, A. and Simon, H. A., 1972. *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall.

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

Waterman, D. A., 1970. "Generalization learning techniques for automating the learning of heuristics", *AI*, Vol. 1, pp. 121-170.