A PRODUCTION SYSTEM FOR LEARNING PLANS FROM AN EXPERT [*]

D Paul Benjamin and Malcolm C Harrison

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

ABSTRACT

This paper describes a method for constructing expert systems in which control information is automatically built from the actions of an expert trainer. This control information consists of sequencing and goal information which is extracted from the trainer by a 'planning expert', and generalized by a 'generalization expert'. A set of extensions to the OPS5 system is described which facilitates the implementation of this approach within OPS5. These extensions permit the use of meta productions to effect the conflict resolution procedure; the control information is expressed as a set of such meta productions. Preliminary experiments with the system are described.

1.0 KNOWLEDGE REPRESENTATION FOR EXPERT SYSTEMS

One of the outstanding problems in constructing 'expert' programs is that of how to extract the necessary information from a human expert. Following Kowalski [3], we view this information as having two components, a logic component and a control component; for computer design, for example, the first would include the axioms and theorems of boolean algebra and the specifications of the available chips, while the second would include the design techniques contained in a textbook on logical design. Note that this is a different categorization from the more traditional declarative/procedural distinction, since the logic component is best viewed as a declarative specification of procedural information. In general, it seems that the problem of dealing with logical information is likely to be more tractable than that of dealing with control information. Accordingly, in our work we have concentrated on the problem of extracting from the human the essential components of his control expertise.

In previous work on this problem [7,8], we represented control information as descriptions of execution traces which had been generated by the expert. These were used to constrain the search space in a way similar to that discussed by Georgeff [2]. With this technique, we were able to construct descriptions which reflected surprisingly accurately the structure of the solution generated by the expert. However, these descriptions were in some cases too rigid, and did not degrade gracefully when the situation was not identical to previously encountered situations. In particular, the program had no notion of similarity, and was unable to recognize parts of the description which were really instances of a more general procedure. In order to improve this aspect of our approach, we needed to adopt a more powerful description mechanism. Since the computational problems of analysing sequences are extreme (at least NP-hard in general) even in the simpler non-probabilistic case, we felt that it would be necessary to make use of more information about the sequences. The two sources of additional information were the production rules themselves, and the human expert. In this paper we concentrate on the problem of extracting more information from the expert, particularly about the planning implicit in his solution.

This work is somewhat similar in essence to Winston's work [12], in that models are constructed and refined according to examples. However, in Winston's case, the set of examples could be chosen to aid the model construction process. In our case, the domain under examination is goal structure rather than physical structure, and is larger and therefore more sparsely populated. This makes the sequence of examples unpredictable and possibly unrelated, and makes anticipation of the changes to the model more difficult. In addition, in our problem we are often forced to make a decision with insufficient evidence. This requires a different approach to the construction and modification of models; we propose to use generalization and partial-matching.

[*] This work was supported by ONR contract number N00014-75-C-0571.

2.0 OVERALL SYSTEM ORGANIZATION

In order to permit the expert trainer to provide not only detailed information about production sequences, but also planning information, the system must provide some framework for specifying such information. In the system described below, there is knowledge of the following kinds:

1. WM elements and productions which permit the possible kinds of actions which the system must be able to do. We refer to these as the 'action' system.

2. WM elements and productions which describe the goal structure used by the system. Some of these are domain-independent, such as standard goal-tree operations for propagating solved or unsolvable markers; others are domain-dependent, such as productions which describe how goals are expanded or solved.

3. Meta productions which specify the order in which the action or goal productions are executed.

Our system constructs the domain-dependent goal productions and the meta productions from the advice of the expert trainer.

2.1 Construction And Generalization Of Plans

The system consists of two expert modules, PLEX and GENEX. PLEX is a planning expert, which interacts with the human expert. GENEX is a generalization expert, which is responsible for generalizing the goal and meta productions provided by PLEX.

The overall flow of the system is as follows. At each cycle, the (human) expert is presented with the set of applicable goal productions, ranked in order of the program's preference. If the expert chooses the top-ranked choice, it is executed and the program continues. If he chooses a lower-ranked choice, PLEX modifies the system by either constructing a new meta production, or modifying a previous production. If the expert rejects all the choices, PLEX constructs, in response to directions from the expert, a new goal and a new meta production; these productions are added to the system, and GENEX is invoked. GENEX may make modifications if appropriate, asking for hints from the expert if necessary, and for the expert's approval for any modifications.

2.2 PLEX

PLEX is a planning expert which contains a model of goal interaction. If a trainer wishes to construct a new goal production during a training session, PLEX asks the trainer for the subgoal structure, and asks questions if necessary to determine the parameters and attributes of the subgoals. For example, if the trainer indicates that an action or sequence of actions is to be repeated, the interface asks questions to extract the information necessary to build a REPEAT node (see below for details of REPEAT nodes). This information includes both the type of the action to be repeated (which may itself involve repetition), and the condition for termination of the repetition.

The interface then translates the trainer's input into an OPS5 production right-hand side, and adds the new goal production to the system's production set. By rehearsing the left-hand side (which consists of the goal to be expanded and its parameters), PLEX causes this new production to enter the conflict set, and be fired by OPS5. In addition, the current state of working memory, along with the conflict set, is sent to GENEX as the left-hand side of a new meta production. The right-hand side is the newly constructed goal production.

2.3 GENEX

The generalization component, GENEX, generalizes the meta productions along the lines of Whitehill [10].

In constructing new meta productions, the expert is given some flexibility in giving names to subgoals; these names will subsequently occur as elements of the meta productions, but PLEX and GENEX initially attach no other semantic significance to them. Initially the names of goals are restricted to a predicate and parameters, in a form such as:

(GOAL ^type build)
(GOAL-PARAM ^type pile ^value edge).

GENEX may subsequently generalise these expressions.

3.0 OPS5 MODIFICATIONS

The production system we are using is OPS5 [1], modified as described below to permit meta productions to make the conflict resolution decisions.

The original OPS5 system does not have the capability to access its conflict set as data for the productions. Our version of OPS5 allows rules to test the conflict set for a particular instantiation of a production. Furthermore, a rule can cause

set for a particular instantiation of a production. Furthermore, a rule can cause an instantiation of an action production to fire. The mechanism to realize this is as follows:

1. WM is assumed to contain goals of the following form which specify possible elements of the action system's conflict set:
```
(EXECUTE ^pname --- ^pid --
   ^ownid <n> ^father ---
   ^status --- ^brother ---)
```
together with one or more parameters:
```
(EXECUTE-PARAM ^class --- ^att ---
   ^value --- ^ownid <n> )
```

2. After any change to the task domain elements in WM, OPS5 automatically links WM elements of the above form with instantiations in the conflict set which have class-attribute-value triples corresponding to the instantiation information in the goal's parameters.

3. An OPS5 user function fire-production is provided that causes a particular instantiation in the conflict set to fire.

4. The system distinguishes among four kinds of productions: executive productions, meta productions, goal productions, and action productions; we have adopted the convention that the names of all action productions begin with 'a', the names of all goal productions begin with 'g', the names of all meta productions begin with 'm', and the names of all executive productions begin with 'x'. The conflict-resolution process has been modified to choose executive productions over meta productions. An executive production which is something like:
```
( p x1 (EXECUTE ^pname <p> ^pid <n>)
   -->
      (call fire-production <n>))
```
is used to fire action productions. OPS5's normal conflict-resolution process is used to choose among the executive productions.

Normally, when a production is added to OPS5, it is not possible for it to enter the conflict set immediately, as the production-matching network is modified by changes to working memory, not to production memory. However, our version has been altered to allow a new production to be instantiated immediately.

3.1 System Structure

With this capability, the tasks were structured in the following manner in order to take advantage of the access to the conflict set.

Goal trees in working memory contain two types of nodes: those which represent abstract goals, which are treated in a purely syntactic fashion; and nodes which express the goal of firing a particular production to affect the domain. The various goal types are represented in the WM by OPS5 elements of the following form:
```
(GOAL <type> <id> <son>
      <brother> <father>)
(GOAL-PARAM <type>
      <value> <id>)
```
where the goal type can be: EXECUTE, REPEAT, SEQ(uence), AND, or OR. The type-value pairs specify information which the particular type of goal needs, such as the termination information for REPEAT goals, or which instantiation to fire for EXECUTE goals. The son, brother, and father values are integers which point to the goals which are below, adjacent to, and above a given goal node in the tree.

The action productions manipulate the domain, and cannot access the goal tree. In the modified OPS5, an action production is not allowed to fire by itself, but instead must be activated by the presence in the goal tree of an EXECUTE goal which requests the firing. The action productions are a complete set of low-level domain manipulations.

The domain-dependent productions and the control productions that manipulate the goal tree are implemented as goal productions and meta productions, respectively. Meta and goal productions may access the goal tree or the domain information, and also the conflict set. Thus, goals can be set depending on which actions are fireable and which goals seem attainable or unattainable.

Domain-independent rules that contain information relevant to the structuring of the goal tree are implemented as executive productions. This information may be high-level, along the lines of Wilensky [11], or merely housekeeping information. An example of the former is detecting goal overlap, which could be represented in OPS5 as:
```
(p x25
  (EXECUTE ^pname <p> ^ownid <id1>)
  (EXECUTE ^pname <p> ^ownid {<> <id1>})
 -->
  (make GOAL ^type bring-into-cs
             ^pname <p>))
```

This production detects the presence of two different instances of the EXECUTE

goal, which are both attempting to execute production <p>, but are different by virtue of their differing 'ownid' fields. Instantiation information is also included in EXECUTE goals, but is not shown for simplicity and space.

An example of a housekeeping executive production is:

```
(p x4
   (OR ^status active ^ownid <x>)
   (<<EXECUTE GOAL AND OR>> ^status done
                            ^father <x>)
-->
   (modify 1 ^status done) )
```

This passes goal satisfaction data back up to an OR node.

Iteration in the goal tree is handled by REPEAT nodes. A REPEAT node requires the goal immediately below it to be repeatedly satisfied until some condition is met. This is implemented by deleting the subtree under the goal under the REPEAT, and rebuilding it as long as the condition is not met.

The deletion of the subtree is done by a set of productions which include the following two executive productions:

```
(p x10
   (delete-this ^id <id1>)
   (GOAL ^ownid <id1> ^son <s>
         ^brother <b>)
  -->
   (remove 1)
   (remove 2)
   (make delete-this ^id <s>)
   (make delete-this ^id <b>) )
(p x11
   (delete-this ^id nil)
  -->
   (remove 1) )
```

3.1.1 Handling Of REPEAT Nodes -

A REPEAT node is a node which specifies that a particular sequence of actions, or a particular goal, should be repeatedly satisfied until a given condition is met.

Every REPEAT node contains a pointer to the goal subtree which is to be repeated, and a pointer to a subtree which represents the condition for terminating the repetition. In addition, the REPEAT node may contain fields indicating the status of the REPEAT goal ('done' or 'active'), and the standard pointers to the father and brother goals.

The portion of the goal tree that is under the REPEAT goal can be removed after each iteration, and regenerated. This would be done to allow each iteration to be accomplished by different means, if necessary. This removal is handled at the production level.

The termination condition of a REPEAT node is represented by a subtree composed of AND, OR, and NOT nodes. The leaf nodes test whether a particular instantiation of a production is in the conflict set. This tree represents the logical relation of those instantiations that must be (or not be) in the conflict set at the completion of the goal subtree under the REPEAT node. Every node in the logical tree has a 'dormant' status. This prevents the execution of the logical tree during the execution of the goal tree. At the completion of the goal tree, the condition represented by the logical tree is tested. If the condition is true, the REPEAT node is marked 'done'. Thus the REPEAT node corresponds to a 'REPEAT-UNTIL' construct.

For the purpose of efficiency, the testing of the logical condition is not handled by productions. This would involve many productions, and consume many cycles. Instead, when a REPEAT condition is first tested, the tree is traversed without evaluation, and converted into a corresponding s-expression by the system. This s-expression is then evaluated to determine whether the repetition continues or not. This s-expression is then saved in a list, and is used whenever this REPEAT node is completed again.

The following example illustrates the above features.

```
(p g-warm-up-room
  (goal ^type warm-up-room ^id <id1>
        ^son nil ^status active)
  (goalnumber ^no <g>)
 -->
  (make repeat ^father <id1> ^status
    active ^ownid (compute <g>+1)
    ^son (compute <g>+2)
    ^terminate (compute <g>+3))
  (make goal ^type raise-one-degree
    ^status active
    ^father (compute <g>+1)
    ^ownid (compute <g>+2))
  (make execute
    ^pname a-temperature-comfortable
    ^ownid (compute <g>+3) ^father
    (compute <g>+1) ^status dormant)
  (modify 1 ^son (compute <g>+1))
  (modify 2 ^no (compute <g>+3)))
```

When the goal of warming up a room is established in the goal tree, the above goal production establishes that this can be accomplished by repeatedly raising the temperature one degree until the action production 'a-temperature-comfortable' enters the conflict set. This production might appear as:

```
(p a-temperature-comfortable
  (temperature ^value { > 67 < 73 })
  --> )
```

Note that this production has a null right-hand side. It serves merely to sense conditions. If the trainer states a condition for which no sensing production exists, PLEX asks the trainer to define the property by creating a new sensing production.

4.0 AN EXAMPLE

An example was constructed to get some feel for the kind of goal and meta productions which would be useful in practice. The example used was a nondeterministic production system which solved a jigsaw puzzle similar to that in [9]. When left to run with no goal or meta guidance, the system repeats the same sequence of productions, and accomplishes nothing. However, when goal productions of the form described above were used to implement a simple strategy, the system proceeds directly to the solution. These goal productions were not generated by PLEX and GENEX, but added directly by the trainer.

The strategy specified by the trainer was simply to find a piece with an edge matching one already in the puzzle, then place this piece in the puzzle. This advice was transformed into a goal tree in several steps.

First, the trainer stated that this goal could be accomplished by first finding the piece with the matching edge, and then placing it into the puzzle. The system constructed a goal production to create a SEQ node; a SEQ node specifies a sequence of subgoals necessary to solve the parent goal.

Next, the trainer defined what was meant by finding a piece with a matching edge. The statement of this definition was that "there must be a piece in the puzzle with a certain edge value, and a piece in the heap of pieces not in the puzzle with the same edge value." This definition was turned into an action production that would match such a piece. This action production had no right-hand side, and thus served as a "sensing" production. This type of production is used by the system to embody definitions and conditions.

Then the second subgoal of putting the piece in the puzzle was clarified by the trainer, who declared that this consisted of a specific sequence of actions chosen from the set of action productions. With this information, the system proceeded to follow the optimal path to solution of the puzzle. Other strategies, such as putting all the pieces of one color in the puzzle at a time, could also be implemented. With these rules, the system is able to solve new puzzles without additional training.

REFERENCES

1. Forgy, Charles L, "OPS5 Users' Manual", Report CMU-CS-81-135, Carnegie-Mellon University, July 1981.

2. Georgeff, M D, "A Framework for Control in Production Systems", Proc. IJCAI-6, 1979.

3. Kowalski, Robert A, "Algorithm = Logic + Control", CACM, July 1979.

4. Langley, Pat, Bradshaw, Gary L, and Simon, Herbert A, "BACON.5: The Discovery of Conservation Laws," Proc. IJCAI-6, 1979.

5. Mitchell, Tom M, Utgoff, Paul E, Nudel, Bernard, and Banerji, Ranan, "Learning Problem-Solving Heuristics Through Practice," Proc. IJCAI-6, 1979.

6. Reboh, Rene, "Using a Matcher to make an Expert Consultation System Behave Intelligently," Proc. AAAI Conference, 1980.

7. Stolfo, Salvatore J, and Harrison, Malcolm C, "Automatic Discovery of Heuristics for Nondeterministic Programs", Proc. IJCAI-6, 1979.

8. Stolfo, Salvatore J, and Harrison, Malcolm C, "Automatic Discovery of Heuristics for Nondeterministic Programs" (full version), Technical Report 7, Courant Institute, 1979.

9. Stolfo, Salvatore J, "Automatic Discovery of Heuristics for Non-deterministic Programs from Sample Execution Traces", PhD Thesis, New York University, 1979.

10. Whitehill, Stephen B, "Self-Correcting Generalization," Proc.AAAI Conference, 1980.

11. Wilensky, Robert, "Meta-Planning: Representing and Using Knowledge about Planning in Problem-Solving and Natural Language Understanding," Cognitive Science, 5, 1981.

12. Winston, Patrick H, "Learning Structural Descriptions from Examples", in "The Psychology of Computer Vision", edited by Winston, McGraw-Hill, 1975.